Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Klient-server aplikace pro výměnu šifrovaných dat

Plzeň 2020 Štěpán Červenka

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 6. května 2020

Štěpán Červenka

Abstract

The subject of this thesis is an implementation of a client-server application in Java which serves as a framework verifying algorithms for encrypting, integrity checking, and a digital signature created by users of the program. The first part of the paper describes how to implement a network application and cryptographic algorithms in Java. In the second part there is introduced the protocol of the server and how are algorithms verified on the server. In the last chapter, the work explains, how a user can use the client for his own cryptographic algorithm verification.

Abstrakt

Práce se zabývá vývojem klient-server aplikace v programovacím jazyku Java, která slouží jako framework ověřující uživatelsky vytvořené kryptografické algoritmy pro šifrování, kontrolu integrity a digitální podpis. V první části práce popisuje možnosti implementace síťových aplikací a kryptografických algoritmů v jazyku Java. Ve druhé části je představen konkrétní protokol a validace uživatelsky vytvořených algoritmů v serverové části aplikace. Na závěr práce vysvětluje, jakým způsobem může uživatel rozšířit klientskou část aplikace pro ověření vlastních kryptografických algoritmů.

Obsah

1 Úvod **3**

2 Síťová část **4**

2.1 Rozdíl TCP a UDP teoreticky 4

2.1.1 Protokoly TCP a UDP v kontextu referenčního modelu TCP/IP 4

2.1.2 Základní rozdíly mezi protokoly 4

2.2 Implementace protokolu TCP v jazyce Java 6

2.2.1 Třída ServerSocket 6

2.2.2 Třída Socket 7

2.2.3 Demo aplikace 8

2.3 Implementace protokolu UDP v jazyce Java 10

2.3.1 Třída DatagramSocket 10

2.3.2 Třída DatagramPacket 12

2.3.3 Demo aplikace 12

2.4 Knihovna java.io 14

2.4.1 Proudy 14

2.4.2 Výhody a nevýhody druhů proudů v kontextu klient-server aplikace 15

2.5 Rozšíření základní TCP aplikace 18

2.5.1 Přijímání více zpráv od jednoho klienta 19

2.5.2 Obsluha více klientů 19

3 Kryptografická část **21**

3.1 Základní popis 21

3.1.1 Úvod 21

3.1.2 Definice pojmů 21

3.1.3 Cíle 22

3.2 Šifrování 23

3.2.1 Symetrické šifrování 23

3.2.2 Asymetrické šifrování 24

3.2.3 Implementace šifrování v jazyce Java 25

3.3 Integrita dat 28

3.3.1 Hashování 28

3.3.2 Implementace hashování v jazyce Java 28

3.3.3 Digitální podpis 29

3.3.4 Implementace digitálního podpisu v jazyce Java . . . 31

4 Implementace serverové části 33

4.1 Kostra serveru 33

4.1.1 Připojování klientů 33

4.1.2 Logování a konfigurace 34

4.1.3 Propojení do funkčního celku 36

4.2 Protokol serveru 37

4.2.1 Funkcionalita serveru 38

4.2.2 Popis a struktura požadavků 39

4.2.3 Popis a struktura odpovědi 43

4.3 Zpracování požadavku serverem 45

4.3.1 Přihlášování klienta na server 45

4.3.2 Zpracování kryptografických požadavků 47

4.3.3 Vykonání kryptografických požadavků 50

4.3.4 Správa klíčů 53

5 Implementace klientské části 56

5.1 Rozhraní pro komunikaci se serverem 56

5.1.1 Spojení se serverem 56

5.1.2 Rozšířená funkcionalita 58

5.2 Testy 59

5.2.1 Testy jako nejvyšší vrstva klientské části programu . 59

5.2.2 Testy ověřující funkčnost programu 60

5.2.3 Odhalené problémy v průběhu testování 60

5.3 Validátor 62

5.3.1 Princip fungování 62

5.3.2 Implementace validátoru 63

6 Závěr 65

Literatura 66

A Uživatelská dokumentace 69

1 Úvod

Cílem této bakalářské práce je vytvoření frameworku pro snadné vytváření a validaci kryptografických algoritmů. Ve většině semestrálních prací předmětu KIV/BIT studenti implementují zvolený, běžně používaný, kryptografický algoritmus. Validace správnosti šifrovacího algoritmu zpravidla probíhá proti stejnému šifrovacímu algoritmu důvěryhodné třetí strany (kryptografické knihovně). Hlavní nevýhodou většiny kryptografických aplikací s jednoduchým uživatelským rozhraní je nemožnost automatizace testů a díky tomu není možné efektivně otestovat implementovaný algoritmus na testovací množině většího rozsahu.

Vytvořený framework poskytne jednoduchou platformu, pomocí které je možné implementovat a ověřit přesně zadanou množinu algoritmů a která především umožní automatizaci testů. Rozdělení frameworku na klientskou a serverovou část přinese administrátorům serveru plnou kontrolu nad množinou kontrolních algoritmů a jejich implementací.

V teoretické části je popsána problematika síťových aplikací i kryptografických algoritmů. Kryptografická část pokrývá symetrické i asymetrické šifrování, integritu dat (včetně digitálního podpisu), tedy všechny zásadní kryptografické techniky používané v informačních technologiích.

Praktickou část práce popisují poslední dvě kapitoly. Na základě tohoto popisu implementace je možné rozšiřovat nebo upravovat funkcionalitu. Poslední kapitola je pak věnována klientské části, která slouží částečně jako uživatelská příručka a představí (primárně studentům) doporučené použití frameworku.

2 Síťová část

2.1 Rozdíl TCP a UDP teoreticky

2.1.1 Protokoly TCP a UDP v kontextu referenčního modelu TCP/IP

Protokoly *TCP* a *UDP* jsou protokoly transportní vrstvy v referenčním modelu *TCP/IP*. Transportní vrstva se vyznačuje tím, že poskytuje přenos dat mezi dvěma konkrétními aplikacemi nezávisle na typu nebo struktuře sítě, po které budou data odesílána. Transportní vrstva pouze předpokládá, že spojení mezi zařízeními bylo zajištěno nižšími vrstvami. Pro adresaci komunikujících aplikací jsou využívány tzv. *porty*, které v rámci jednoho protokolu jednoznačně identifikují konkrétní aplikaci na jednom zařízení. Cílový a zdrojový port společně s cílovou a zdrojovou IP adresou a názvem transportního protokolu definuje konkrétní datový tok mezi zařízeními [20].

2.1.2 Základní rozdíly mezi protokoly

Protokol TCP obecně posílá více metadat, která zajišťují některé speciální vlastnosti, které protokol UDP nemá. Zjednodušeně by se dalo říct, že zprávy odeslané protokolem TCP:

1. Se neztratí.
2. Dorazí bez chyb.
3. Dorazí ve stejném pořadí, jako byly odeslány.

Spolehlivost

Protokol TCP je spolehlivý protokol. Je garantováno, že data odeslaná tímto protokolem dorazí k příjemci. Pokud dojde je ztrátě dat, nebo jejich poškození, dojde ke znovuodeslání dat. Protokol UDP je nespolehlivý. Data odeslaná tímto protokolem nemusí dorazit, nebo mohou dorazit poškozená. Spolehlivost souvisí s prvními dvěma body výše uvedeného seznamu [11].

Spojovanost

S třetím bodem ze seznamu souvisí pojem spojovanost. Protokol TCP označujeme jako spojovaný protokol, protože vytváření spojení mezi dvěma

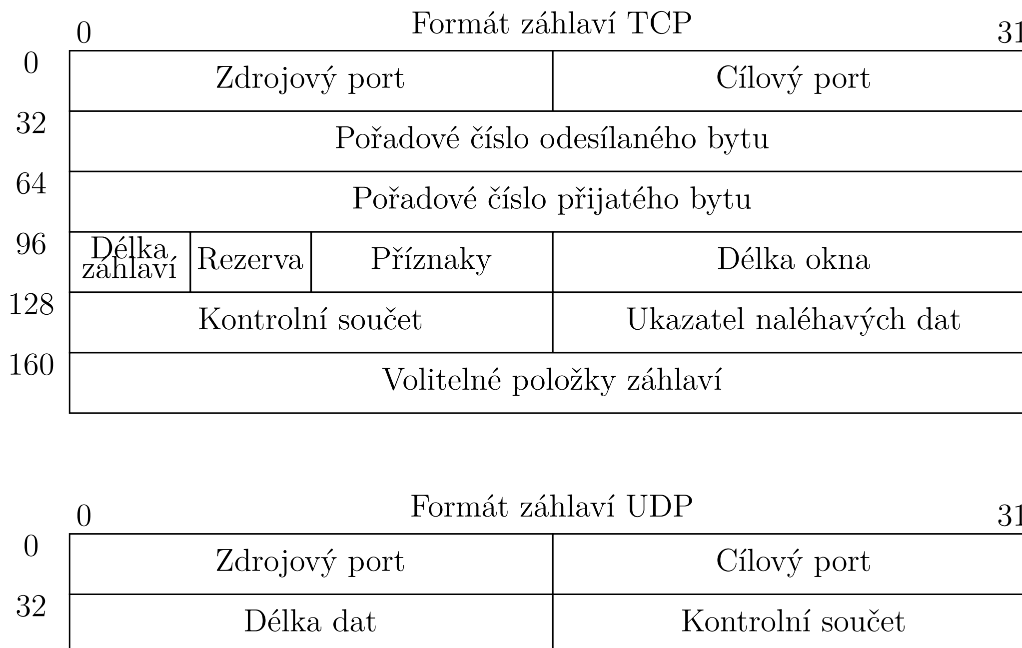
zařízeními před zahájením komunikace. Protokol UDP spojení nenavazuje a rovnou odesílá data [11].

Řízení toku dat

Mechanismus řízení toku dat využívaný v protokolu TCP zajišťuje, že odesílatel nezahltí příjemce přílišným množstvím dat díky implementaci vyrovnávací paměti [11].

Rychlost

TCP pro zajištění výše vypsaných služeb musí odesílat větší množství metadat (viz obrázek 2.1). Velkou výhodou protokolu UDP je tedy jeho rychlost [11].



Obrázek 2.1: Rozdíl hlaviček TCP a UDP

Shrnutí

Z uvedených výhod a nevýhod vyplývá, že protokol TCP využíváme hlavně v případě, kdy nutně nepotřebujeme odezvu v reálném čase a výhody poskytované protokolem TCP tedy pomalejší rychlost převáží. Příkladem takovéhoto použití může být například hypertextový protokol HTTP, nebo protokol pro přenos souborů FTP. UDP naopak využijeme při živém

vysílání multimediálního obsahu, online hrách nebo při překladu doménových jmen na adresy pomocí protokolu DNS [11].

2.2 Implementace protokolu TCP v jazyce Java

K tomu, abychom komunikovali pomocí protokolu TCP v jazyce Java, využijeme dvě třídy z knihovny *java.net*. Tou první je třída *ServerSocket*, která slouží převážně k definici spojení na straně serveru. Druhou důležitou třídu je třída *Socket*, která využívá knihovny *java.io* k samotné výměně zpráv.

2.2.1 Třída ServerSocket

Třída *ServerSocket* [17] je v celé klient-server aplikaci zastoupena typicky jedinou instancí, která je obvykle vytvořena při spuštění serverové části aplikace. Základní funkce této třídy jsou:

1. Vytvoření serverového socketu.
2. Svázání (bind) s konkrétním portem.
3. Poslouchání (listen) na portu.
4. Vytváření nových spojení.

Konstruktor třídy

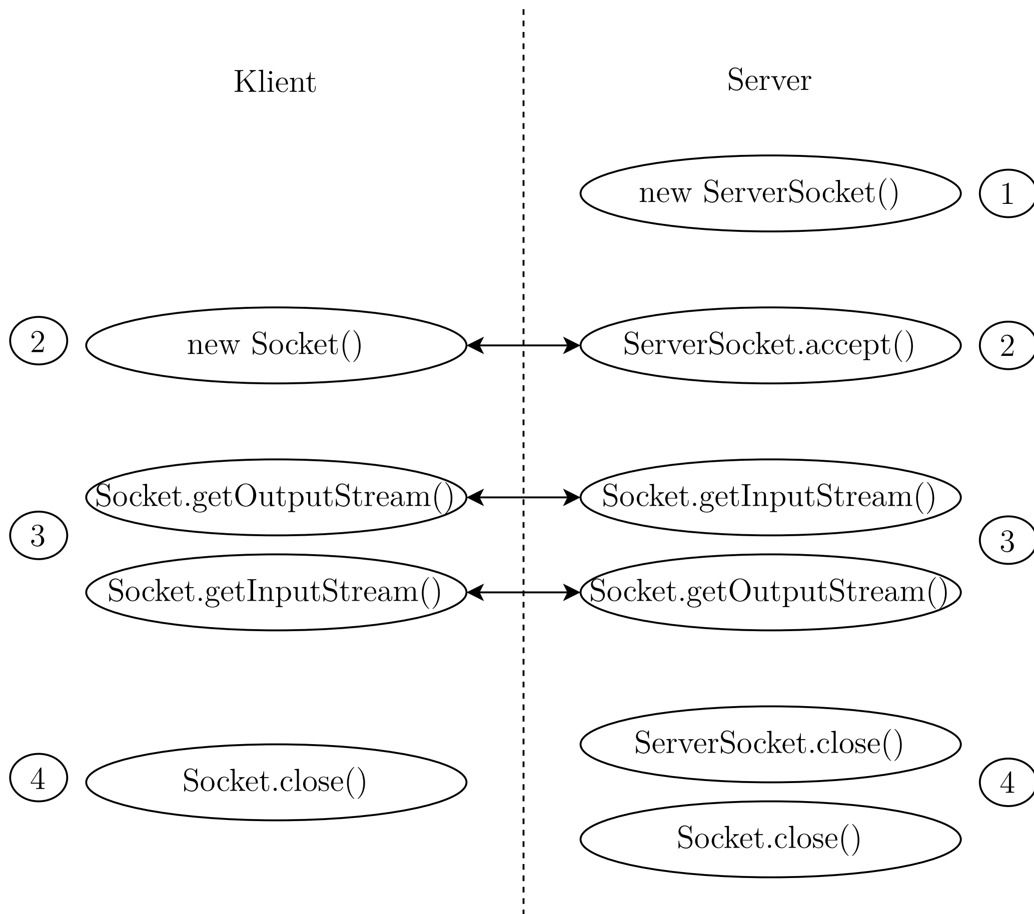
Konstruktory třídy lze rozdělit na dvě základní kategorie. Typicky je v hlavičce konstruktoru uveden port, na který chceme socket svázat. V takovém případě konstruktor ihned splní první dvě funkce z výše uvedeného seznamu. Instanci třídy lze ovšem vytvořit i bez uvedení portu. V takovém případě je IP adresu a port třeba dodatečně uvést pomocí metody *bind()*.

Metody třídy

Důležitou metodou třídy *ServerSocket* je metoda *accept()*. Ta totiž plní zbylé dvě výše vypsané základní funkce třídy. Metoda *accept()* je blokující[1] metoda, která poslouchá na zvoleném portu a v případě nového spojení

[1] Uspí vlákno, dokud nenastane požadování I/O operace. V našem případě je to připojení klienta.

vytváření instanci třídy *Socket*, která bude sloužit ke komunikaci s nově připojeným klientem. Dalšími metodami třídy jsou například výše zmíněná metoda *bind()*, nebo metody nastavující důležité detaily komunikace, jako jsou příznaky (např. *SO_REUSEADDR*[2]), nebo nastavení velikosti bufferu obdržených zpráv. Opomenout nesmíme ani metodu *close()*, která socket uzavře.



Obrázek 2.2: Diagram volání metod TCP protokolu v jazyce Java

2.2.2 Třída Socket

Socket je z definice koncovým bodem dvoustranné komunikace mezi dvěma programy v síti. Každý z dvojice socketů dokáže odeslat zprávu druhému socketu, nebo analogicky zprávu přijmout. V rámci implementace protokolu

[2]Tento příznak dovolí v určitých případech použít adresu a port, které jsou "obsazeny"jiným socketem.

TCP v jazyce Java, bude instance třídy *Socket* [19] držena serverem i klientem. Mezi základní funkcionalitu třídy *Socket* patří:

1. Vytvoření socketu na straně klienta.
2. Připojení se k serveru.
3. Odeslání zprávy.
4. Přijetí zprávy.

Konstruktor třídy

Vytvoření instance třídy *Socket* se diametrálně odlišuje pro klientskou i serverovou část aplikace. V případě založení na klientu je využit konstruktor třídy. Analogicky k třídě *ServerSocket* je možné založit *Socket* explicitně s uvedením IP adresy a portu serveru. V takovém případě se klientský socket rovnou pokusí připojit. V případě neuvedení IP adresy a portu serveru při vytváření *Socketu* je potřeba později zavolat metodu *connect()*, která připojení provede. Vytvoření *Socketu* na serveru provede třída *ServerSocket* zavoláním metody *accept()* po připojení klienta.

Metody třídy

Hlavní funkcionalitou třídy je přijímání a odesílání zpráv. K tomu slouží odkazy na třídy *InputStream* a *OutputStream* z knihovny *java.io*. *OutputStream* z prvního socketu dovoluje posílat proud bytů, které k němu vztažený *InputStream* z druhého socketu přijímá (naznačeno na obr. 2.2). Dále jsou zde opět zastoupeny metody na zjišťování a nastavování příznaků síťové komunikace nebo metoda *close()*.

2.2.3 Demo aplikace

V minulých podkapitolách jsme si představili teoretický základ implementace protokolu TCP v jazyce Java. Nyní si tento základ zkusíme demonstrovat na praktické ukázce.

```java
import java.net.*;
import java.io.*;

public class Server{
    public void start(int port) {
        ServerSocket serverSocket = new ServerSocket(port);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out = new PrintWriter(
            clientSocket.getOutPutStream());
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                clientSocket.getInputStream()));
        out.println(in.readLine().toUpperCase());
        clientSocket.close();
        serverSocket.close();
        }
}
```

Ukázka kódu 2.1: Jednoduchý TCP Server

Abychom se vyhnuli jakýmkoliv smyčkám a udrželi program co nejjednodušší, omezíme se na situaci, kdy server dokáže přijmout pouze jednu textovou zprávy od jednoho klienta. Tuto zprávu zprávu poté převede na velká písmena a odešle zpátky. Poté se server ukončí (viz ukázka kódu 2.1).

Práce klientské části aplikace je ještě o něco jednodušší. Klient se pokusí připojit, odeslat zprávu, a vypíše přijatou zprávu do konzole (viz ukázka kódu 2.2).

Chování programu

Struktura programu se drží diagramu uvedeném na obrázku 2.2. Jako první se musí spustit server, který založí socket a zablokuje se na metodě *accept()*, která čeká, než se připojí klient. Klient se zavoláním konstruktoru třídy *Socket* připojí k serverové části a odtud pokračuje program asynchronně. Oba dva programy poté pomocí objektů *InputStream* a *OutputStream* inicializují třídy, které dovolují posílat a přijímat text (viz podkapitola 2.4). Poté dojde již k výše zmiňované výměně zpráv[3] Na konci programů dojde k uzravření socketů.

[3]Synchronizace je zaručena metodami *in.readLine()*. Na nich se programy zablokují v případě, že jsou rychlejší než jejich protějšek. V opačném případě dojde k přečtení zprávy z bufferu.

```java
import java.net.*;
import java.io.*;

public class Client{
    public void start(String ip, int port, String message) {
        Socket clientSocket = new Socket(ip, port);
        PrintWriter out = new PrintWriter(
            clientSocket.getOutPutStream());
        BufferedReader in = new BufferedReader(
            newInputStreamReader(
                clientSocket.getInputStream()));
        out.println(message);
        System.out.println(in.readLine());
        clientSocket.close();
        }
}
```

Ukázka kódu 2.2: Jednoduchý TCP Klient

2.3 Implementace protokolu UDP v jazyce Java

Obdobně jako v případě protokolu TCP i komunikaci pomocí protokolu UDP v jazyce Java obsluhují převážně dvě základní třídy. Třída *DatagramSocket* je jakýmsi spojením funkcionality tříd *ServerSocket* a *Socket* z protokolu TCP. Druhá důležitá třída *DatagramPacket* nahrazuje knihovnu *java.io*, která v případě implementace protokolu UDP není implicitně využita.

2.3.1 Třída DatagramSocket

Instanci třídy *DatagramSocket* [15] musí vytvořit serverová i klientská část aplikace. Účelem této třídy je:

1. Vytvoření síťového socketu.
2. Svazání socketu s konkrétním portem.
3. Odeslání packetu.
4. Přijetí packetu.

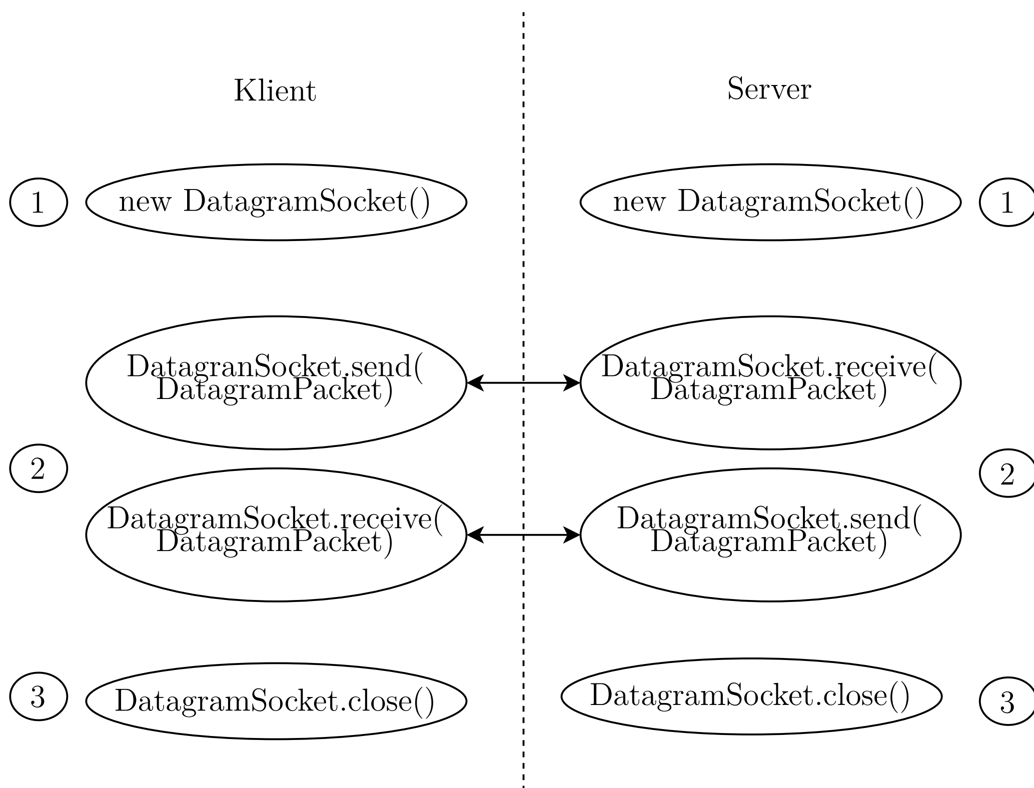
Konstruktor třídy

Návrh přetížení konstruktoru třídy *DatagramSocket* je velmi podobný jako v případě třídy *ServerSocket* v protokolu TCP. Opět máme dostupné dvě základní možnosti. Můžeme socket pouze vytvořit a jeho svázání s portem nechat na později, nebo můžeme vytvořit socket s definovaným portem,

jehož svázání proběhne v rámci konstruktoru. Novinkou oproti třídě *ServerSocket* je ale možnost svázání na nespecifikovaný dostupný port, čehož je využito typicky v případě vytváření této třídy na straně klienta[4].

Metody třídy

Hlavní metody třídy se jmenují *send()* a *receive()* a jejich úkolem je posílat a přijímat packety mezi serverem a klientem. Mechanismus posílání packetů je obdobný jako v případě protokolu TCP. Metoda *send()* odešle packet reprezentovaný třídou *DatagramPacket* (viz následující podkapitola 2.3.2), který přijme odpovídající blokující metoda *receive()*. Ve třídě nechybí ani metody na zjišťování a nastavování příznaků síťové komunikace, metoda *bind()*, kterou využijeme ke svázání socketu k portu, nebo metoda *close()* pro uzavření socketu.



Obrázek 2.3: Diagram volání metod UDP protokolu v jazyce Java

[4]Port, který byl použit je sdílen se serverovou částí v rámci metainformací, které obsahuje třída *DatagramPacket*.

2.3.2 Třída DatagramPacket

Třída *DatagramPacket* [14] je v protokolu UDP využívána pro přenos informací mezi serverem a klientem. Její funkcionalita by se dala shrnout následovně:

1. Přenos posílaných dat po síti.
2. Přenos IP adresy a portu odesílatele.

Konstruktor třídy

Konstruktor třídy se liší pro případy, kdy chceme packet odesílat nebo přijímat. V obou případech je třeba uvést odkaz na pole bytů sloužící jako buffer dat pro odesílání nebo přijímání. K bufferu je možné uvést offset[5] a délku v případě, kdy chceme využít pouze část pole. V případě, že chceme packet odeslat, je třeba ještě specifikovat IP adresu a port příjemce.

Metody třídy

Metody třídy jsou omezeny a zjišťování nebo nastavování hodnot parametrů objektu. Nejčastěji využívané metody jsou metody *getAddress()* a *getPort()* pro zjištění IP adresy a portu protistrany. Tyto metody jsou zásadní hlavně na straně serveru při přijetí první zprávy od klienta, protože jsou často jediným způsobem, jak zjistit adresu a port, na který má server odpovědět.

2.3.3 Demo aplikace

Stejně jako v případě protokolu TCP se z důvodu zachování jednoduchosti omezíme na situaci, kdy server i klient dokáží odeslat a přijmout pouze jeden packet.

Cíl programu zůstane také stejný. Server přijme od klienta zprávy, převede jí na velká písmena a odešle zpět. Klient přijatou zprávu vypíše.

[5] Nebudou využity data od začátku pole, ale od specifikovaného indexu.

```java
import java.net.*;

public class Server{
    public void start(int port) {
        DatagramSocket socket = new DatagramSocket(port);
        byte[] buffer = new byte[256];
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
        socket.receive(packet);

        InetAddress clientAddress = packet.getAddress();
        int clientPort = packet.getPort();
        String received = new String(
            packet.getData(), 0, packet.getLength());
        byte[] toSend = received.toUpperCase().getBytes();

        packet = new DatagramPacket(
            toSend, toSend.length, clientAddress, clientPort);
        socket.send(packet);

        socket.close();
        }
}
```

Ukázka kódu 2.3: Jednoduchý UDP Server

Chování programu

Postupovat budeme opět dle uvedeného diagramu na obrázku 2.3. V první části kódu obou programů (viz ukázka kódu 2.3 resp. 2.4) dojde k inicializaci třídy *DatagramSocket*, v případě serveru uvedeme port.

Odesílání a přijímání zpráv v rámci protokolu UDP v jazyce Java je striktně navázáno na použití pole bytů jako bufferu. Abychom odeslali textovou informaci z klientské části aplikace, využijeme knihovní metody *getBytes()*, která převede datový typ *String* na právě potřebné pole bytů. To pomocí metody *send()* odešleme. Na serveru si mezitím připravíme prázdný buffer, do kterého se uloží přijatá zpráva metodou *receive()*.

Na serveru poté dojde ke zjištění posílaných dat a informací o klientovi metodami *getData()*, *getAddress()* a *getPort()*. Přijatou zprávu převede na textovou reprezentaci, dle zadání tuto textovou reprezentaci převede na velká písmena a zpětně převede na pole bytů, které odešle. Klient tuto zprávu načte do připraveného bufferu a vytiskne do konzole. Poté dojde k zavření socketů.

```
import java.net.*;

public class Client{
    public void start(InetAddress address, int port, String message) {
        DatagramSocket socket = new DatagramSocket();
        byte[] toSend = message.getBytes();
        DatagramPacket packet = new DatagramPacket(
            toSend, toSend.length, address, port);
        socket.send(packet);

        byte[] buffer = new byte[256];
        packet = new DatagramPacket(buffer, buffer.length);
        socket.receive(packet);

        String received = new String(
            packet.getData(), 0, packet.getLength());
        System.out.println(received);

        socket.close();
        }
}
```

Ukázka kódu 2.4: Jednoduchý UDP Klient

2.4 Knihovna java.io

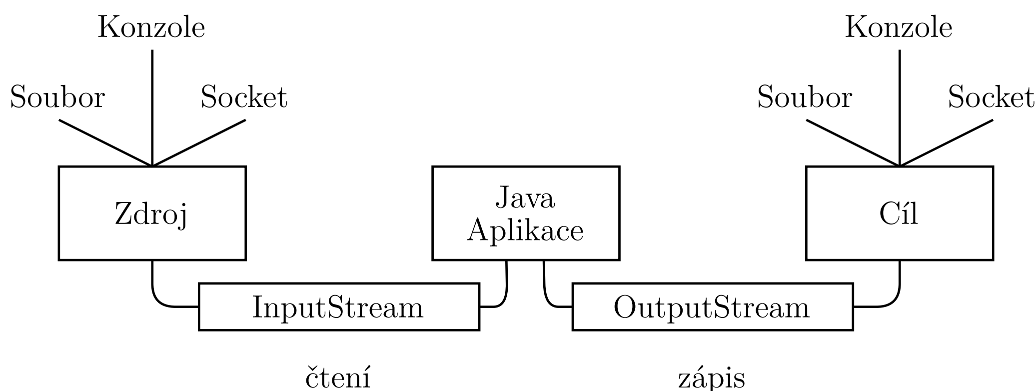
Knihovna *java.io* je úzce navázána na implementaci protokolu TCP. Právě touto knihovnou je totiž implementován přenos dat mezi serverem a klientem. Pro TCP aplikaci v jazyce Java je tedy naprosto zásadní, jakým způsobem je tento přenos implementován.

2.4.1 Proudy

Knihovna *java.io* je založena na konceptu tzv. proudů (anglicky stream). Jde o teoreticky neomezeně dlouhou sekvenci dat, která například narozdíl od konceptu pole neobsahuje indexy. Data lze tedy číst pouze v pořadí, ve kterém byla do proudu zapsána.

Proudy můžeme rozdělit na dva základní druhy (viz obrázek 2.4). Aplikace napsaná v jazyce Java může z nějakého zdroje (např. z konzole nebo socketu) pomocí proudu data číst. Takový proud poté nazýváme *InputStream*. V opačném případě, kdy aplikace na nějaké místo zapisuje, používá obvykle *OutputStream* [5].

Oba druhy proudů jsou v jazyce Java zastoupeny abstraktními třídami *InputStream* a *OutputStream*, které jsou poté konkrétně implementovány jejich potomky.



Obrázek 2.4: Obecné schéma proudů InputStream a OutputStream

Znakové proudy

Proudy jsou z definice v programovacím jazyce Java orientovány po jednotlivých bytech. V případě, že chceme místo bytů zapisovat nebo číst znaky využijeme abstraktní třídy *Reader* a *Writer*. Tyto třídy mohou existovat zcela nezávisle na výše zmíněných proudech, nicméně zpravidla jejich implementace obsahují konstruktor s odkazy na *InputStream* nebo *OutputStream*. Jsou proto vhodným nástrojem k implementaci znakového proudu a v síťové komunikace pomocí nich můžeme implementovat znakově orientovaný protokol. Stačí v konstruktoru *Readeru* nebo *Writeru* uvést odkaz na *InputStream* nebo *OutputStream*, který vrací třída *Socket* [6, 12].

2.4.2 Výhody a nevýhody druhů proudů v kontextu klient-server aplikace

V této podkapitole porovnáme tři druhy implemetace proudů v síťových aplikacích v jazyce Java. Jedná se vždy o konkrétní potomoky tříd *InputStream*, *OutputStream*, *Reader*, nebo *Writer*. Jsou to:

1. Posílání bitů.
2. Posílání znaků.
3. Posílání objektů.

Posílání informací v síťových může být samozřejmě implementováno kterýmkoliv druhem proudu. Vybrány byly tři nejtypičtější, které mohou být použité v největším množstvím případů.

Posílání bitů

V této podkapitole uvažujeme situaci, kdy posílaným datovým typem mezi serverem a klientem je proud bitů. Jedná se o nejvíce nízkoúrovňové řešení. Pro jednolivé úkony by byly vytvořeny speciální značky jako komunikačního nástroje, kterým by klient specifikoval, kterou akci po serveru vyžaduje.

K posílání textu by byla použita metoda *getBytes()* pro převod textu na pole bytů v jazyce Java. Na druhou stranu k posílání obecných dat[6] by samozřejmě nebyl třeba žádný převod.

Implementace proudu bitů by v jazyce Java mohla být realizována pomocí tříd *DataInputStream* a *DataOutputStream*. Samotný proud bitů by pak byl přenášen v rámci pole bytů (viz ukázka kódu 2.5).

Mezi základní výhody a nevýhody takovéto implementace patří:

+ Vysoká přenositelnost mezi platformami
+ Odesíláno malé množství dat
– Nutnost definice vlastních značek
– Špatná rozšiřitelnost

Mezi hlavní výhody takovéto implementace patří snadná přenositelnost na jinou platformu. Proud bitů je totiž plně nezávislý na jakémkoliv programovacím jazyku. Další výhodou takovéhoto přístupu je nízká redundance odesílaných dat, z čehož plyne posílání pouze nezbytného množství informací.

Nevýhodou ovšem je, že interpretace takovýchto dat už je složitější. Typicky je třeba definovat vlastní unikátní značky, které mimo jiné činí rozšíření programu (např. jiným programátorem) složitější, protože je třeba tyto definované značky respektovat.

```
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
out.write(new byte[]);

DataInputStream in = new DataInputStream(
socket.getInputStream());
byte[] message = new byte[256];
in.readFully(message, 0, message.length);
```

Ukázka kódu 2.5: Implementace odesílání bitů

[6] Například klíče pro účely šifrování, nebo jiné obecné struktur.

Posílání znaků

Posílání znaků se výhodami a nevýhodami nijak zvlášť neliší od posílání bitů. Souvisí to především s tím, že převod mezi znaky a bity je relativně přímočarý. Přesto můžeme registrovat určité rozdíly.

Proud znaků v jazyce Java lze implementovat například pomocí formátovaného vstupu a výstupu. Třídy *PrintWriter* a *BufferedReader* jsou k tomu vhodné (viz ukázka kódu 2.6).

Vybrané výhody a nevýhody:

+ Stále slušná přenositelnost

+ Odesíláno rozumné množství dat

– Nutnost definice vlastních značek, nicméně tentokrát čitelnějších

– Špatná rozšiřitelnost

– Obtížnější posílání čistých dat

Při použití známého kódování znaků by přenositelnost mezi platformami neměla ve srovnání s posíláním bitů nijak utrpět. Rovněž množství odesílaných dat by mělo při rozumně definovaném protokolu zůstat řádově stejné. Naopak je předpoklad, že to malé množství dat posílané navíc bude ve prospěch čitelnosti jednotlivých značek.

Na druhou stranu rozšíření programu bude ulehčeno pouze v malé míře. Naopak novou nevýhodou tohoto konceptu přenosu dat může být matoucí převod textu na obecná data v závislosti na zvoleném kódování znaků. Posílání obecných dat se kvůli tomu stává obtížnější.

```
PrintWriter out = new PrintWriter(socket.getOutputStream());
out.println(new String());

BufferedReader in = new BufferedReader(new InputStreamReader(
socket.getInputStream()));
String received = in.readLine();
```

Ukázka kódu 2.6: Implementace odesílání znaků

Posílání objektů

Poslední zkoumanou implementací proudu v síťovém programování bude možnost posílat celé objekty. Jedná se o jediné vysokoúrovňové řešení, ve kterém můžeme využít všechny výhody objektově orientovaného programování.

V Javě je takovýto přenos realizován pomocí tříd *ObjectInputStream* a *ObjectOutputStream* (viz ukázka kódu 2.7). Aby příjemce zjistil třídu, jejíž instanci získal, může využít klauzuli *instanceof*.

Výhody a nevýhody budou tentokrát diametrálně odlišné:

+ Možnost posílat libovolné datové typy
+ Snadná rozšiřitelnost programu
− Špatná přenositelnost mezi platformami
− Velká redundance dat

Největší výhodou posílání Java objektů je ta skutečnost, že objektem může být široká kombinace datových typu, datových struktur, souborů apod., které bychom pomocí textu nebo bitů vyjadřovali jen obtížně. Zároveň převod těchto datových typu plně delegujeme na interpret programovacího jazyka. Z toho plyne i mnohem snadnější rozšiřitelnost programu, díky absenci unikátních značek vytvořených programátorem.

Přímou nevýhodou, která z tohoto přístupu plyne, je špatná rozšiřitelnost mezi jiné programovací jazyky. Pokud bychom chtěli k serveru využívající posílání objektů připojit klient napsaný v jiném jazyce, museli bychom implementovat serializaci objektů na proud dat, nebo na serveru přidat podporu jiného typu protokolu. Další nevýhodou je posílání nadbytečného množství dat. Je ovšem otázkou, jestli je tato nevýhoda vzhledem k rychlosti dnešních sítí, skutečným problémem.

```
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
out.writeObject(new Object());

ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
Object o = in.readObject();
```

Ukázka kódu 2.7: Implementace odesílání objektů

2.5 Rozšíření základní TCP aplikace

Až do teď jsme se ve všech praktických ukázkách kódu omezovali na situaci, kdy se k serveru může připojit pouze jeden klient, který může navíc odeslat pouze jednu zprávu. V této kapitole navrhneme řešení, jak obsluhovat více klientů s neomezeným množstvím zpráv. Budeme tedy aplikaci rozšiřovat výhradně na straně serveru.

2.5.1 Přijímání více zpráv od jednoho klienta

Problém, který nám potenciálně brání přijímat více zpráv je ten, že pro každou poslanou zprávu od klienta je třeba na serveru zavolat nějakou formu metody *read()*. Protože v drtivé většině případů nevíme, kolikrát klient nějakou zprávu odešle, je na místě metodu *read()* volat ve smyčce. Díky tomu, že je metoda *read()* blokující, nebude smyčka odebírat výpočetní výkon dalším procesům.

Praktický příklad

V následujícím příkladu rozšíříme jednoduchý program, který jsme použili na ukázku základních metod TCP serveru (viz ukázka kódu 2.1).

```
import java.net.*;
import java.io.*;

public class Server{
    public void start(int port) {
        ...
        String input;
        while(true) {
            input = in.readLine();
            if(input.equals("quit") {
                out.println("quiting");
                break;
            }
            out.println(input.toUpperCase());
        }
        clientSocket.close();
        serverSocket.close();
        }
}
```

Ukázka kódu 2.8: Přijímání zpráv ve smyčce

Základem je nekonečná smyčka, která přijímá neomezené množství zpráv od jednoho klienta. Ve smyčce provádí server stejnou operace převodu textu na velká písmena, jako v ukázkách v předcházejících kapitolách. Ovšem v případě, že klient odešle příkaz *quit* dojde k ukončení nekonečné smyčky a uzavření socketů.

2.5.2 Obsluha více klientů

Dalším omezením, které v této podkapitole překonáme, je možnost obsluhovat více klientů. Omezením je způsobeno tím, že na jednom vláknu může být aktivní vždy pouze jedna blokující metoda. Je tedy třeba přijímání nových klientů (metoda *accept()*) a přijímání zpráv od jednotlivých klientů (varianta metody *read()*) rozdělit do jednotlivých vláken programu.

Praktický příklad

V tomto příkladu si již nevystačíme s pouhým rozšířením jedné metody, protože z logiky věci musí dojít k rozdělení implementace do minimálně dvou částí. Jedna část bude přijímat nové klienty a druhá část je bude obsluhovat.

```
import java.net.*;

public class Server{
    public void start(int port) {
        ServerSocket socket = new ServerSocket(port);
        while(true) {
            new ClientHandler(socket.accept()).start();
        }
    }
}
```

Ukázka kódu 2.9: Přijímání nových klientů

Přijímání nových klientů (viz ukázka kódu 2.9) běží na hlavním vláknu programu. Po spuštění metody dojde k inicializaci nové instanci třídy *ServerSocket*, která poslouchá na zvoleném portu a deleguje obsluhu klienta novému vláknu.

```
import java.net.*;
import java.io.*;

public class ClientHandler extends Thread{
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        PrintWriter out = new PrintWriter(
            socket.getOutPutStream());
        BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));

        while(true) {
        ...
        }

        socket.close();
    }
}
```

Ukázka kódu 2.10: Obsluha klientů ve vlastním vláknu

Vlákno je implementováno jako potomek třídy *Thread*, jak je v jazyce Java obvyklé (viz ukázka kódu 2.10). Kód logiky obsluhy vložíme do metody *run()*, kterou překryjeme původní implementaci. Pokud nyní zavoláme metodu *start()* nové třídy, je metoda *run()* spuštěna v novém vláknu. Tímto způsobem je možné čekat na zprávy více klientů najednou.

3 Kryptografická část

3.1 Základní popis

3.1.1 Úvod

Kryptografie se zabývá matematickými metodami, jak utajit smysl nebo existenci určité informace. Kryptografické techniky byly využívány již ve starověku převážně pro válečné nebo diplomatické účely, jejichž účelem bylo tehdy hlavně skrýt zprávu, nebo její význam, před nepřítelem. Nástup informačních a komunikačních technologií však dramaticky zvýšil možnost kopírovat a měnit informace, proto bylo potřeba nově zajistit i bezpečnost těchto informací v elektronické podobě. Tato bezpečnost by měla být zajištěna nezávisle na fyzickém médiu [23].

3.1.2 Definice pojmů

V následujícím textu budeme zacházet s řadou pojmů, které je vhodné před použitím definovat.

Šifrovací systém je systém, který pro danou množinu klíčů používá šifrovací a dešifrovací algoritmy k transformaci prostého textu na text šifrovaný a naopak [25].

Prostý text (plaintext) je text (nebo data) v jazyce, kterému běžně rozumí široká skupina lidí [25].

Šifrovaný text (ciphertext) je text (nebo data) v tajném jazyce, kterému rozumí pouze úzká skupina lidí se specifickou znalostí, obvykle po dešifrování daného textu [25].

Symetrický šifrovací systém je šifrovací systém, který používá stejný klíč pro šifrování i dešifrování dat [25].

Asymetrický šifrovací systém je šifrovací systém, který využívá dva rozdílné klíče pro šifrování a dešifrování [25].

Klíč je element, který definuje některé konkrétní kroky šifrovací a dešifrovacího algoritmu [25].

Veřejný klíč je klíč, který využívají asymetrické kryptosystémy. Klíč je známý široké skupině lidí a v naší práci se využívá výhradně pro šifrování.

Soukromý klíč je klíč, který využívají asymetrické kryptosystémy. Klíč je známý pouze úzké autorizované skupině lidí a v naší práci se používá výhradně pro dešifrování.

Hashovací funkce je výpočetně efektivní zobrazení dat (binárních řetězců) libovolné délky na data pevné délky [23].

Digitální podpis zprávy jsou data, která vznikla transformací původní zprávy za použití asymetrické kryptografie a jsou k původní zprávě připojena. Digitální podpis zajišťuje jak autentizaci odesílatele, tak validitu odesílaných dat [23].

3.1.3 Cíle

V následující podkapitole jsou popsány cíle, které se dnešní kryptografie snaží naplnit. [23]

Důvěrnost je služba, která zabraňuje přístupu k informacím všem, kteří nejsou autorizování tyto informace získat. Příklad naplnění tohoto cíle v rámci naší práce mohou být zašifrovaná data, která dokáže přečíst pouze entita, která zná algoritmus a klíč k jejich dešifrování.

Integrita dat zabraňuje neoprávněné modifikaci dat. K zajištění integrity dat je potřeba mechanismus, který zajistí, že data, která byla původně odeslaná jsou stejná jako data, která byla přijata. Hashovací funkce a digitální podpis jsou příklady technik sloužící k zajištění integrity dat, které se vyskytují v této práci.

Autentizace je proces ověření identity uživatele nebo zprávy kterou poslal. V práci s tímto pojmem souvisí asymetrické šifrování a digitální podpis, tedy techniky, které využívají veřejný a soukromý klíč.

Nepopíratelnost zajišťuje, že autor nemůže popřít předchozí závazky nebo akce. Nepopíratelnost může opět zajistit asymetrické šifrování nebo digitální podpis, které jsou v rámci práce implementovány.

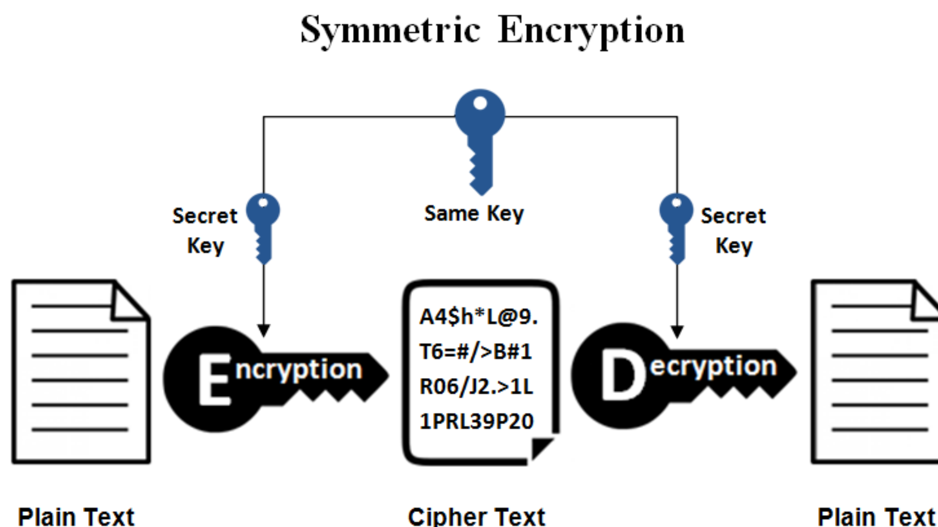
3.2 Šifrování

3.2.1 Symetrické šifrování

Symetrické šifry se vyznačují tím, že používají jeden klíč pro šifrování i dešifrování (viz obrázek 3.1). Algoritmy symetrického šifrování můžeme rozdělit do dvou základních kategorií:

1. Proudové
2. Blokové

Zatímco proudové šifrovací systémy šifrují každý bit prostého textu zvlášť, blokové šifrovací systémy šifrují najednou vždy určitý blok dat, jehož velikost je předem určená.



Obrázek 3.1: Schéma symetrického šifrování

Zdroj obrázku: https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences

Blokové šifry

Většina blokových šifer je založena na principu Feistelovy sítě. Tato implementace má zásádní výhodu v tom, že mechanismus šifrování i dešifrování je velice podobný. Algoritmus je založený na postupném provedení tzv. rund, což je aplikace stále stejných operací. Zástupcem blokových šifer je například šifra AES [10].

Proudové šifry

Prodouvé šifry potřebují ke svému fungování pseudonáhodný proud bitů. Na prostý text a tento proud dat je poté zpravidla aplikovaná logická funkce XOR, jejíž výsledkem je šifrovaný text.

Proudové šifry mají oproti blokovým několik nevýhod. Příkladem takovéto nevýhody je skutečnost, že jeden bit šifrovaného textu je zašifrován pomocí právě jednoho bitu prostého textu, což značně zjednodušuje prolomení šifry. Další nevýhodou proudových šifer je snadná modifikace šifrovaného textu. Mezi již zašifrované bity je možné vložit bity vlastní, které je obtížné detekovat. Příkladem proudové šifry je RC4. [10].

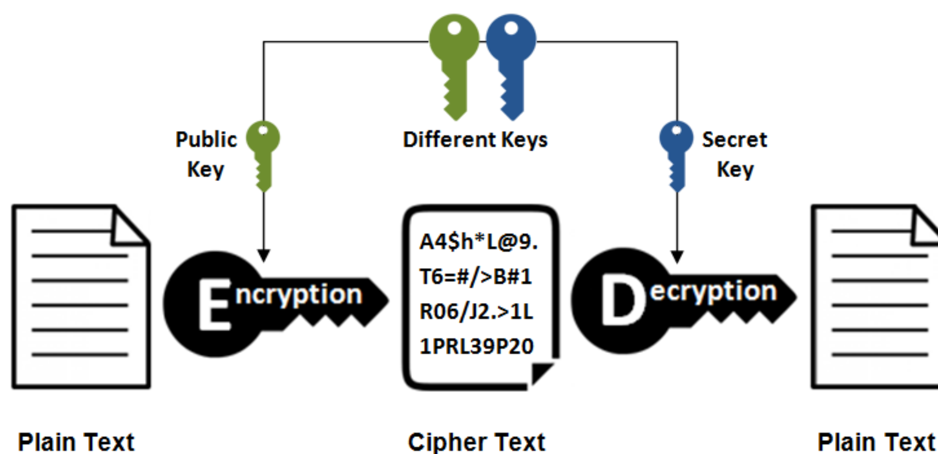
3.2.2 Asymetrické šifrování

Jak již bylo v předchozích kapitolách naznačeno, asymetrické šifrovací systémy narozdíl od symetrických pracují se dvěma klíči - veřejným a soukromým (viz obrázek 3.2). Veřejný klíč je zpravidla běžně dostupný a slouží k zašifrování zprávy. Jediný, kdo je schopný zprávu dešifrovat je vlastník soukromého klíče, což zajišťuje důvěrnost zprávy.

Porovnání se symetrickým šifrováním

Hlavním rozdílem mezi symetrickým a asymetrickým šifrování je v distribuci klíčů. Sdílet veřejný klíč, jak již jeho název napovídá, je totiž možné veřejně, což je zpravidla mnohem jednodušší než tajné sdílení jednoho klíče pro symetrickou kryptografii.

Naopak jednou z nevýhod asymetrického šifrování je vyšší výpočetní náročnost na provedení algoritmu. Z tohoto důvodu jsou asymetrické šifrovací systémy při aplikaci na větší objem dat běžně využívány v kombinaci se symetrickou kryptografií. Data jsou v tomto případě zašifrována symetrickou šifrou, na jejíž klíč je použita šifra asymetrická [23].

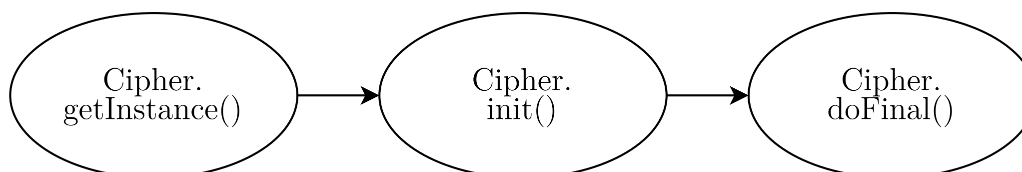


Obrázek 3.2: Schéma asymetrického šifrování

Zdroj obrázku: https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences

3.2.3 Implementace šifrování v jazyce Java

Veškeré kryptografické nástroje jsou v jazyce Java implementovány v knihovně *javax.crypto*.



Obrázek 3.3: Schéma použití třídy Cipher

Šifrování a dešifrování poté zajišťuje třída *Cipher* (viz obrázek 3.3) [13]. Celý proces získání šifrovaného textu (nebo prostého v případě dešifrování) má následující tři kroky.

Získání instance

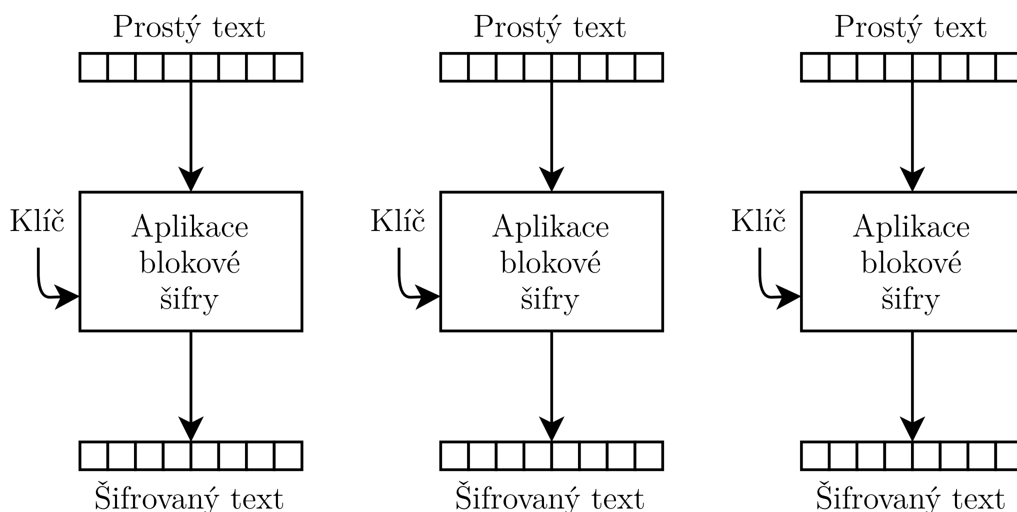
Konstruktor třídy Cipher je chráněný[1] a k získání instance slouží tovární metoda *getInstance()*. Ta má jako parametr textový řetěz obsahující

[1] Klíčové slovo `protected`.

tři následující parametry[2].

1. **Název algoritmu** označuje definovanou zkratkou jeden z algoritmů podporovaný touto knihovnou.

2. **Režim blokových šifer** ovlivňuje, jakým způsobem jsou bloky šifrovány. Nejjednodušším režimem blokových šifer je režim *ECB* (viz obrázek 3.4), kdy je blok šifrovaného textu získán pouze zašifrováním prostého textu, bez aplikace jakékoliv další operace[3]. V ostatních režimech (například CBC obrázek 3.5) šifer je vstup nebo výstup algoritmu upraven. Obvykle jsou tato data zkombinována logickou funkcí XOR se vstupem nebo výstupem předchozího bloku. V těchto režimech je také využíván tzv. inicializační vektor, který zjednodušeně řečeno plní roli předchozího bloku pro blok první [1].

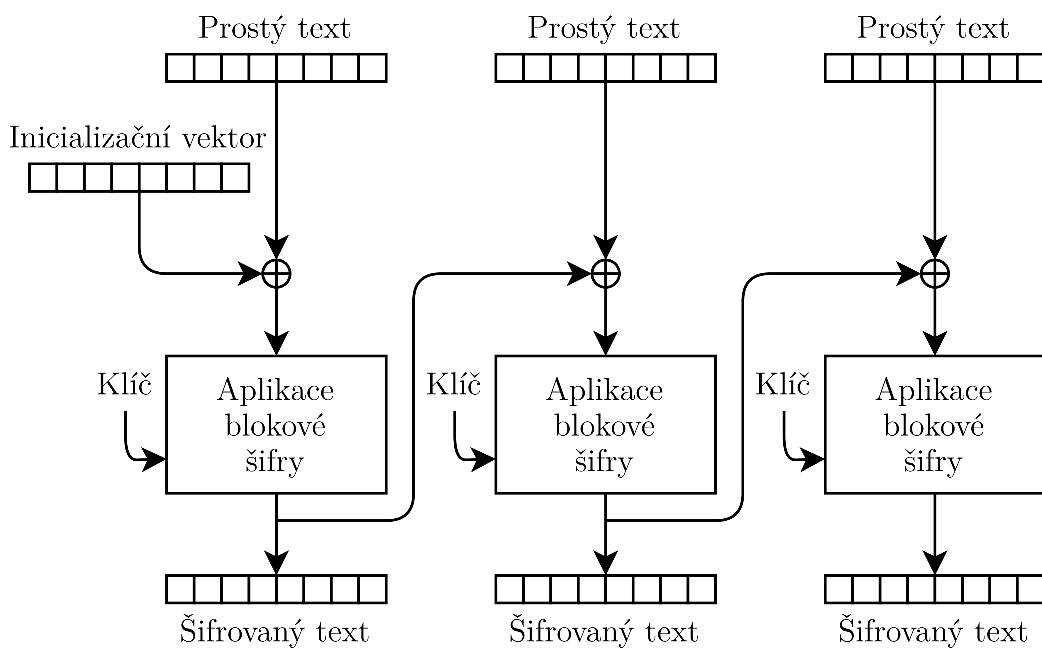


Obrázek 3.4: Schéma šifrování dle režimu ECB

3. **Padding** je následující operace. Prostý text musí být před aplikací blokové šifry rozdělen na bloky dat, jeho délka proto musí být dělitelná délkou bloku dané šifry. Pro případy, kdy tato podmínka není splněna, může být na konec prostého textu vloženo potřebné množství bitů tak, aby prostý text bylo na bloky možné rozdělit. Způsob, jakým jsou tyto bity vkládány je pak poslední parametrem k získání instance třídy *Cipher*.

[2]K vytvoření instance stačí zadat název algoritmu. V takovém případě se použije implicitní režim blokových šifer a padding

[3]Šifra RSA, která není bloková, musí být inicializována vždy módem ECB



Obrázek 3.5: Schéma šifrování dle režimu CBC

Inicializace

Inicializace objektu třídy *Cipher* se provádí metodou *init()*. Parametry, které do metody uvedeme, typicky jsou:

1. **Mód operace** specifukuje, k čemu chceme inicializovanou šifru použít. Nejčastějším příkladem použití jsou módy *ENCRYPT_MODE* a *DECRYPT_MODE* sloužící k zašífrování, respektive dešifrování textu.

2. **Klíč** je datového typu *byte[]* a musí mít samozřejmě validní formát definovaný v rámci daného algoritmu.

3. **Inicializační vektor** uvádíme samozřejmě jen v případech použití jiného režimu blokových šifer, než je *ECB*.

Objekt je možno inicializovat dalšími parametry. Tyto parametry jsou však nepovinné a pro tuto práci zbytečné.

Provedení operace

K provedení konečné operace slouží metoda *doFinal()*, do které ve většině případů ještě uvedeme data, se kterými chceme operace provést[4]. Návratovou

[4] Tedy prostý text v případě šifrování a šifrovaný text v případě dešifrování

hodnotou této funkci jsou data po provedení příslušného algoritmu. Data jsou datového typu *byte[]*.

3.3 Integrita dat

Integritou dat máme na mysli situaci, kdy s určitou mírou jistoty můžeme konstatovat, že přijatá data nebyla modifikována nebo poškozena. Tuto funkci zpravidla plní hashovací funkce. Digitální podpis pak do celého procesu zavádí prvek autentizace. V následujícím textu se budeme zabývat pouze kryptografickými technikami zajištění integrity dat. Integrita dat však může být zajištěna i jinými technikami, například metodou CRC [21].

3.3.1 Hashování

Hashovací funkce slouží k převedení dat libovolné délky na data pevné, předem určené, délky, který označujeme jako *hash* (česky otisk). Takto definované zobrazení bude z principu obsahovat kolize, protože množina možných vstupů je řádově mnohem větší, než množina výstupů [4].

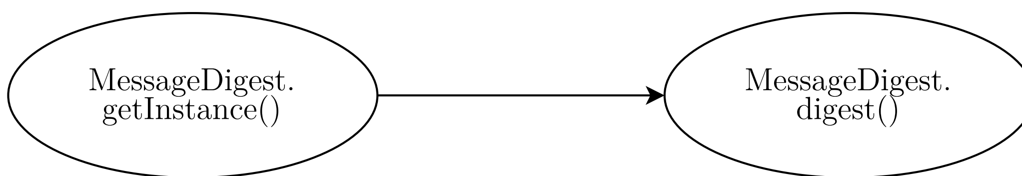
Hashovací funkce by měla splňovat několik základních požadavků:

1. Výpočetní efektivita.
2. Rovnoměrné rozložení vstupů.
3. Výpočetně náročné odhalení vstupu na základě výstupu.
4. Výpočetně náročné odhalení stejného výstupu pro dva různé vstupy.
5. Malá změna ve vstupu způsobí velkou změnu na výstupu.

Zejména poslední podmínka je podstatná pro využití hashovacích funkcí k zachování integrity dat. V modelové situaci může například odesílatel poslat příjemci zprávu a její otisk. Příjemce po přijetí může ověřit validitu zprávu tím, že znovu vytvoří její otisk. Pokud se zpráva změnila, výsledný otisk by měl být jiný již na první pohled [7, 23].

3.3.2 Implementace hashování v jazyce Java

Schéma hashování (viz obrázek 3.6) v jazyce Java bude podobné jako v případě šifrování. Vzhledem k menšímu množství parametrů se však bude jednat o výrazně jednodušší proces.



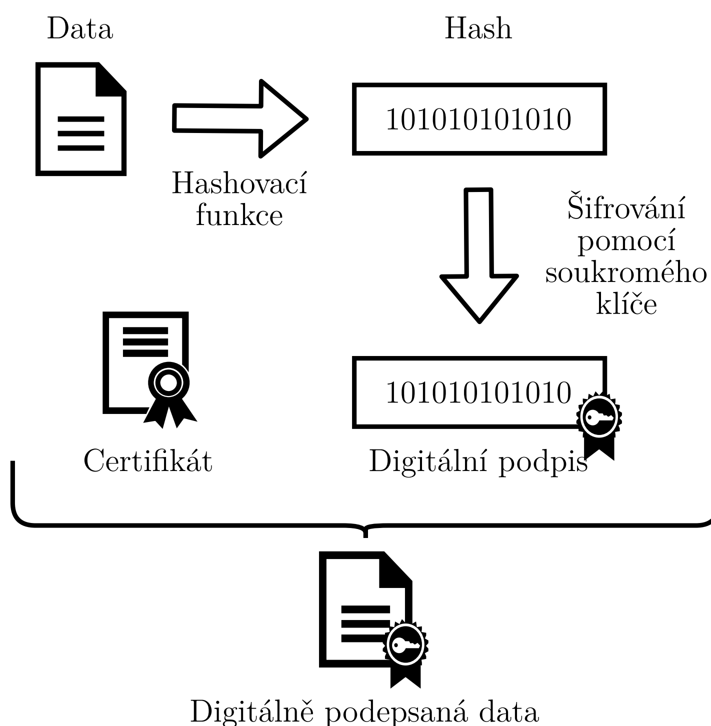
Obrázek 3.6: Schéma použití třídy MessageDigest

K hashování slouží v knihovně *javax.crypto* třída *MessageDigest* [16]. Instanci získáme znovu tovární metodou *getInstance()*, tentokrát ovšem jako argument uvedeme pouze název algoritmu.

Metoda *digest()* je poté ekvivalentní k šifrovací metodě *doFinal()* a zajistí samotné provedení algoritmu.

3.3.3 Digitální podpis

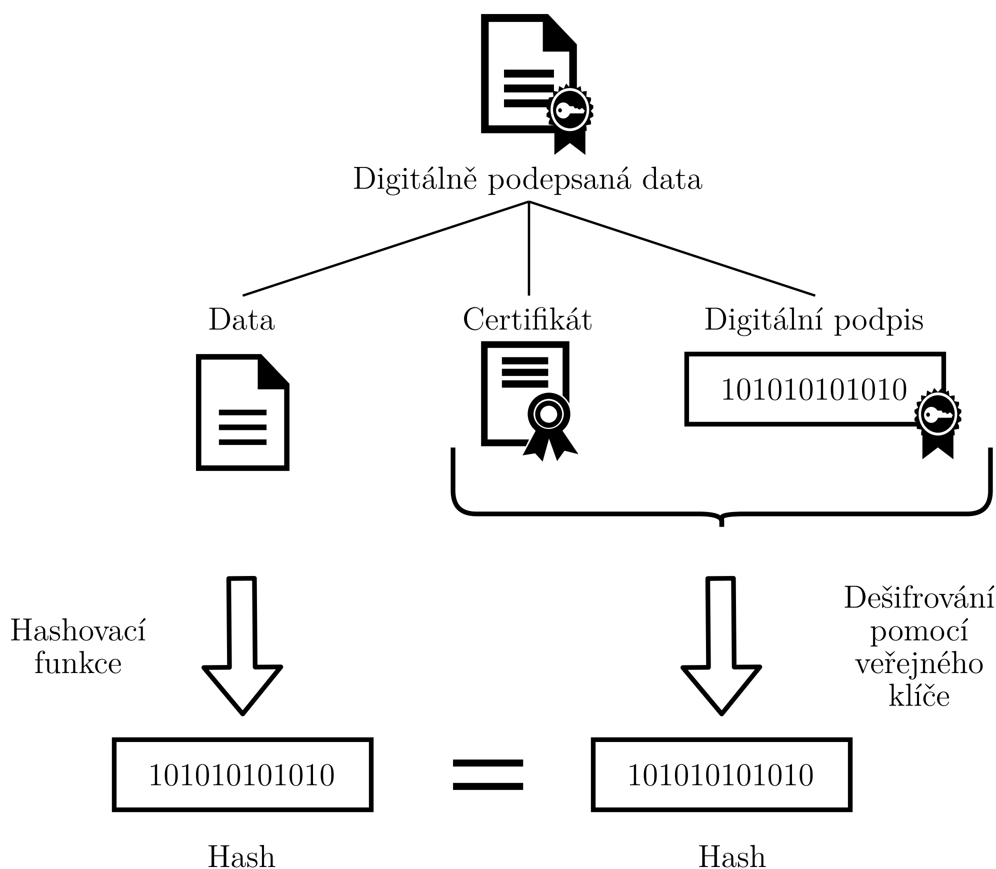
Digitální podpis kombinuje prvky hashování a asymetrické kryptografie do jednoho funkčního celku. Celý proces se skládá z dvou základních operací.



Obrázek 3.7: Schéma podepsání

Podepsání zpráv

Podepsání zprávy (viz obrázek 3.7) je operace, kterou provádí odesílatel. Algoritmus nejprve zprávu zahashuje. Poté je otisk ještě zašifrován privátním klíčem[5] odesílatele. V praxi jsou podepsaná data ještě opatřena certifikátem od nezávislé autority, který zajišťuje nepopíratelnost autora odeslaných dat [2, 3].



Obrázek 3.8: Schéma ověřování digitálního podpisu

Ověření podpisu

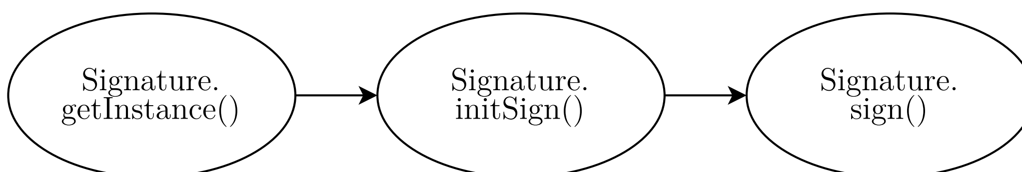
Ověření podpisu (viz obrázek 3.8) probíhá na straně příjemce zprávy. Uživatel porovnává dva otisky zpráv. První získá opětovným zahashováním původní zprávy. K opatření druhého otisku dešifruje digitální podpis odesílatele. Pokud se oba otisky rovnají, je podpis validní [2].

[5] V rámci této práce se technicky provádí operace dešifrování

3.3.4 Implementace digitálního podpisu v jazyce Java

Analogií k třídám *Cipher* a *MessageDigest* z knihovna *javax.crypto* bude tentokrát třída *Signature* [18]. Ovšem vzhledem k tomu, že mechanismus podepisování a ověřování digitálního podpisu se poměrně liší, je oddělená do určité míry i implementace v rámci této třídy.

Podepisování



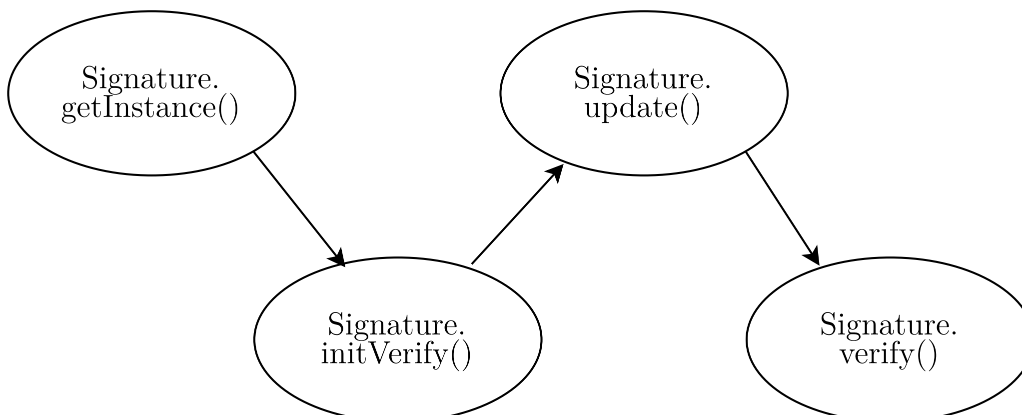
Obrázek 3.9: Schéma podepisování pomocí třídy Signature

Pokud chceme využít třídu *Signature* k podepisování (viz obrázek 3.9 nejprve standardně zavoláme tovární metodu *getInstance()*, kam uvedeme název algoritmu pro hashování a šifrování.

Inicializace proběhně podobně jako v případě šifrování pomocí metody *initSign()*, kam zadáme i privátní klíč.

Samotné podepsání provedeme metodou *sign()*, do které jako parametr vstupuje původní zpráva.

Ověřování



Obrázek 3.10: Schéma ověřování podpisu pomocí třídy Signature

Pro ověření zprávy nejprve závoláme opět metodu *getInstance()*, se stejnými parametry jako v případě podepisování.

Poté následuje metoda *initVerify()*, kam vložíme veřejný klíč.

Dále je třeba zavolat metodu *update* pro zadání původní zprávy.

Ověření pak proběhne pomocí metody *verify()*, která si jako parametr bere samotný digitální podpis zprávy odesílatele.

4 Implementace serverové části

4.1 Kostra serveru

V kapitole 2.5 byla uvedena jednoduchá implementace základní obsluhy více klientů. V této kapitole bylo také vysvětleno, proč je použití vláken a nekonečných smyček při implementaci serverové části klient-server aplikace v programovacím jazyce Java nevyhnutelné. Následující kapitoly tedy budou navazovat na ukázky kódu (2.8, 2.9 a 2.10) uvedené v této kapitole.

4.1.1 Připojování klientů

Implementaci metody *run()*

Implementace vláken v jazyce Java je v zásadě možná dvěma způsoby. První možností je vytvořit potomka třídy *Thread*, druhou pak implementovat rozhraní *Runnable*.

Obě možnosti se ve své funkčnosti příliš neliší, protože pokaždé bude prakticky celá implementace spočívat v tom, že překryjeme, respektive implementujeme, metodu *run()*.

V našem případě dává přece jen o něco větší smysl implementovat rozhraní *Runnable*. To zajistí větší možnost rozšiřitelnosti, protože v jazyce Java je možné implementovat libovolné množství rozhraní, ale dědit lze jen od jedné třídy.

Třída *ExecutorService*

I v případě samotného spuštění vlákna máme v programovacím jazyce Java na výběr. Nejjednodušší cestou, jak vlákno spustit, je metoda *start()*, která na novém vláknu zavolá metodu *run()*. V takovém případě máme však jen omezenou míru kontroly nad daným vláknem.

Jako praktičtější varianta se jeví použití třídy *ExecutorService*. Tato možnost nám umožní specifikovat maximální množství vytvořených vláken[1], čímž zabráníme přetěžování serveru.

V případě, že se klient pokusí připojit k plnému serveru, je zařazen do fronty a čeká na obsloužení předchozích vláken.

[1] Což je v našem případě ekvivalentní k počtu připojených klientů.

4.1.2 Logování a konfigurace

Mezi základní funkce klient-server aplikací patří logování a konfigurace. V našem případě se nejedná o podstatnou část programu, nedává tedy smysl porovnávat již existující řešení, nebo dokonce implementovat svoje vlastní. V obou případech se spolehneme na existující implementace v rámci knihoven jazyku Java.

Logování

Pro logování využijeme třídu *Logger* z knihovny *java.util.logging*. Pro specifikaci výstupu využijeme třídu *FileHandler*, která mimo jiné umožňuje volbu logování do více souborů pro větší projekty. My tuto možnost nevyužijeme a jako parametr zvolíme soubor zadaný v konfiguračním souboru (viz další podkapitola). Na závěr je třeba zvolit ještě formátování výstupu. My pro jednoduchost použijeme třídu *SimpleFormatter*. Nyní stačí jen tyto třídy propojit (viz ukázka kódu 4.1).

```
import java.util.logging.*;

public class Server{

   private Logger logger;

   private void setupLogger() {
        this.logger = Logger.getLogger("ServerLog");

        String logFileName = reader.getString("log_file_name");
        FileHandler handler = new FileHandler(logFileName, true);
        this.logger.addHandler(handler);

        SimpleFormatter formatter = new SimpleFormatter();
        handler.setFormatter(formatter);
    }

    ...
}
```

Ukázka kódu 4.1: Ukázka použití třídy Logger

Konfigurace

Implementaci konfigurace je vhodné oddělit od hlavní třídy serveru vytvořením třídy vlastní. To zajistí větší přehlednost. Zároveň je konfigurace potenciálním zdrojem možných výjimek, které si díky této implementaci projeví už v nově vytvořené třídě a pro uživatele bude snadnější nastalý problém identifikovat a opravit.

Nová třída nazvaná *ConfigReader* bude obsahovat referenci na instanci třídy *Properties*, která poskytuje jednoduché rozhraní pro tabulku typu klíč

- hodnota. Do této instance načteme pomocí třídy *FileInputStream* text z konfiguračního souboru. Konstruktor třídy bude vypadat následovně (viz ukázka kódu 4.2).

```
import java.util.Properties;
import java.io.*;

public class ConfigReader {

    private Properties properties;

    public ConfigReader(String filename) {
        this.properties = new Properties();
        InputStream is = null;

        try {
            is = new FileInputStream(filename);
            properties.load(is);
        } catch (FileNotFoundException e) {
            System.err.println("File named " + filename + " not found.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    ...
}
```

Ukázka kódu 4.2: Konstruktor třídy ConfigReader

Poslední věc, která zbývá, je delegace získání hodnot z konfiguračního souboru z třídy *ConfigReader* na třídu *Properties*. Třída *Properties* obecně pracuje s hodnotami datového typu *String*. Protože v konfiguračním souboru se budou vyskytovat i číselné hodnoty, poskytneme i rozhraní pro hodnoty datového typu *int* (viz ukázka kódu 4.3). Koverzi textového řetězce na číslo nebudeme nijak ošetřovat, úpravu konfiguračního souboru považujeme za pokročilou akci, není proto třeba vytvářet uživatelsky přívětivé řešení.

```
import java.util.Properties;
import java.io.*;

public class ConfigReader {
    ...

    public String getString(String key) {
        return properties.getProperty(key);
    }

    public int getInt(String key) {
        String val = properties.getProperty(key);
        return Integer.parseInt(val);
    }
}
```

Ukázka kódu 4.3: Metody třídy ConfigReader

4.1.3 Propojení do funkčního celku

Vztah mezi vlákny

V kontextu našeho programu musí existovat obousměrná reference[2] mezi hlavní serverovou třídou (v našem programu pojmenovaná jako třída *Server*) a třídou implementující rozhraní *Runnable*, která obsluhuje jednotlivé klienty (pojmenovaná *ClientHandling*). Demonstrovat si to můžeme na jednoduchém případu při přihlašování uživatele, kdy potřebujeme zkontrolovat, že přezdívka uživatele je v rámci celého serveru unikátní. V takovém případě musí mít třída *Server* přehled o všech přezdívkách uživatelů a zároveň musí třída *ClientHandling* k tomuto přehledu přístup.

Výsledek

Zkombinování jednotlivých prvků zmíněných v předchozích kapitolách je možné ukázat na metodě *start* (viz ukázka kódu 4.4), která server spustí. Proměnné *port* a *maxClientCount* jsou nahrané z konfiguračního souboru v konstruktoru třídy *Server*.

Na této ukázce kódu je naznačen princip fungování logování i konfigurace. Zároveň je možné pozorovat rozšíření základní aplikace z kapitoly 2.5, využití třídy *ExecutorService* a obousměrná reference naznačená v konstruktoru třídy *ClientHandling* a pomocí seznamu *users* implementované datovou strukturou *ArrayList*.

[2]Tzn. třída A má referenci na třídu B a naopak třída B na třídu A

```java
import java.util.logging.*;

public class Server{
    ...

    private List<ClientHandling> users = new ArrayList<>();
    private int maxClientCount;
    private int port;

    ...

    private void start() throws IOException {

        maxClientCount = getReader().getInt("max_client_count");
        port = getReader().getInt("port");

        getLogger().info("Server starts on port " + port + ".");

        try (ServerSocket listener = new ServerSocket(port)) {
            ExecutorService pool =
            Executors.newFixedThreadPool(maxClientCount);
            while (true) {
                ClientHandling handling =
                new ClientHandling(listener.accept(), this);
                users.add(handling);

                getLogger().info("User connected on port "
                + handling.socket.getPort() + ".");
                broadcastOnlineClients(); //vypise do logu pocet uzivatelu
                pool.execute(handling);
            }
        }
    }

    ...
}
```

Ukázka kódu 4.4: Metoda run

4.2 Protokol serveru

V kapitole 2.4.2 jsme si představili výhody a nevýhody jednotlivých proudů v kontextu klient-server aplikace v rámci programovacího jazyku Java.

Nejvhodnějším způsobem komunikace mezi serverem a klientem v našem případě je možnost odesílat celé objekty. Vzhledem k zádání práce neočekáváme přenos na jinou platformu, server i klient je programován v jazyce Java. Zároveň na základě zádání můžeme očekávat různorodost přenášených dat. Bude se jednat například o specifikaci požadavků, obecná data pro posílání klíčů, prostého a šifrovaného textu, logická hodnota ano/ne například pro implementaci digitálního podpisu, nebo vypsání důvodu, proč daný požadavek nebyl zpracován.

Protokol je navržený tak, že klient vždy vyšle požadavek, na který server zareaguje danou odpovědí. Jedná se o výhradní způsob výměny dat, server tedy žádnou komunikaci neinicializuje, ani nereaguje na nic jiného, než je validní požadavek.

4.2.1 Funkcionalita serveru

Před popisem samotné implementace je vhodné si určit, jakou funkcionalitu bude server vůbec vykonávat. Tyto požadavky můžeme rozdělit do dvou velkých skupin a jednoho samostatného požadavku stojící mimo tuto hierarchii.

Kryptografické požadavky

První ze dvou skupin jsou požadavky na využití kryptografie. Konkrétně se jedná o:

1. Požadavek na šifrování.
2. Požadavek na digitální podpis.
3. Požadavek na aplikaci hashovací funkce.
4. Požadavek na ověření validity klíčů pro asymetrické šifrování.

Šifrování a digitální podpis probíhá vždy s veřejným klíčem, které je nejprve nutné na server nahrát (viz následující podkapitola). Hashovací funkce ze své definice nepotřebují ke svému provozu jakýkoliv klíč, jejich použití je proto nejjednodušší. Mechanismus ověření validity dvojice klíču asymetrické kryptografie je od ostatních požadavků odlišný, protože nepracuje s konkrétními daty, ale pouze s dvojicí klíčů, což se také projeví na implementaci tohoto požadavku. Klíče pro ověření, narozdíl od požadavků na šifrování a digitální podpis, předem nenahráváme, ale uvádíme je jako součást požadavku, protože chceme předejít ukládání soukromých klíčů na potenciálně nechráněný disk.

Požadavky na manipulaci s klíči

Konkrétními implementaci na manipulaci s klíči jsou:

1. Požadavek na nahrání klíče.
2. Požadavek na stáhnutí klíče.

3. Požadavek na odstranění klíče.

Pro každého klienta je na serverovém disku vytvořena složka, kam se ukládají nahrané klíče, které se používají pro šifrování a digitální podpis. Každý typ klíče (uvedený ve výčtovém typu *KeyType* při posílání požadavku) může být obsažen ve složce nanejvýš jednou. Klíč je poté možný přepsat nahráním nového klíče stejného typu, nebo ho odstranit pomocí určeného požadavku.

Požadavek na přihlášení

Mimo obě kategorie stojí požadavek na přihlášení, které zajišťuje identifikaci. Na základě uživatelského jména je vytvořena složka pro klíče. Zároveň je toto jméno využíváno v logovacích záznamech.

4.2.2 Popis a struktura požadavků

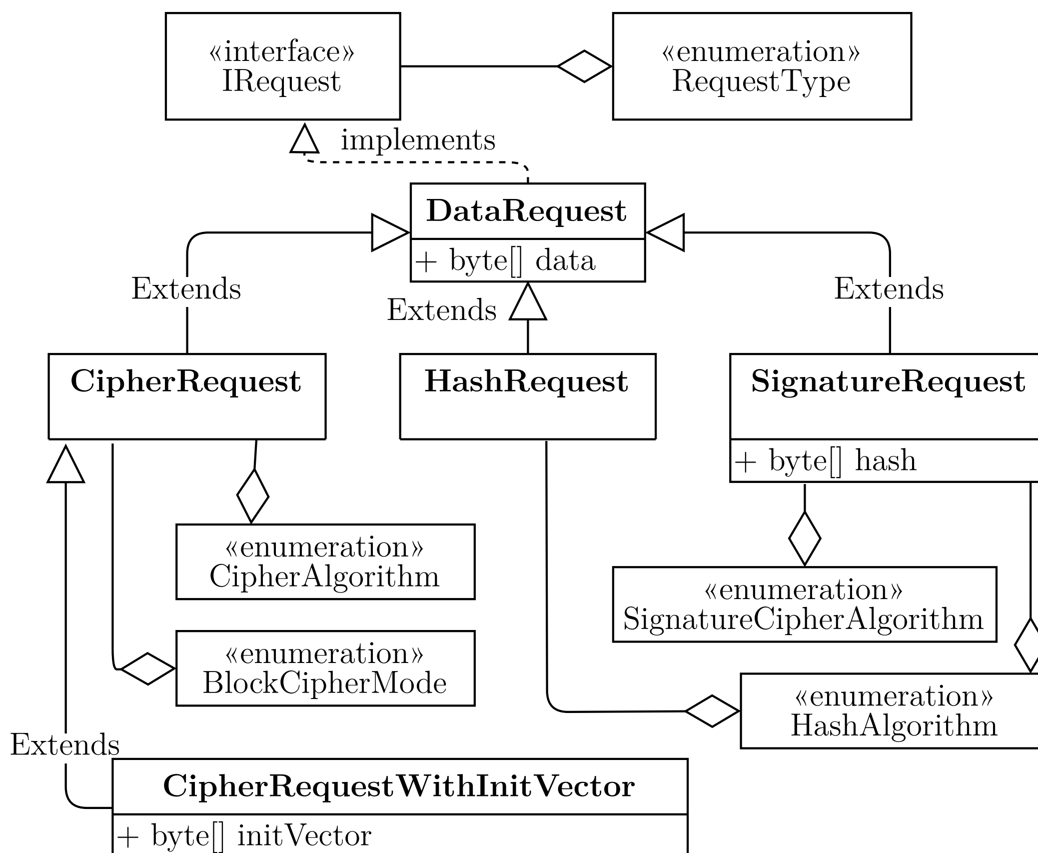
Každý konkrétní požadavek je reprezentován instancí rozhraní *IRequest*. Samotná třída, jejíž instancí je daný požadavek, pak musí implementovat metodu *getRequestType()*, která je v tomto rozhraní definována.

Požadavky jsou pak nadále členěny dle jejich implementace. S výjimkou požadavku na ověření validity klíčů se jedná o stejné rozdělení jako v předchozí podkapitole. První skupinou jsou požadavky na zpracování nějakého typu dat (viz obrázek 4.1), které definují většinu kryptografických požadavků. Požadavky na manipulaci s klíči jsou zastoupeny v druhé skupině (viz obrázek 4.2). Požadavky, pro které nebyl nalezen nějaký společný prvek s ostatními požadavky, jsou uvedeny zvlášť (viz obrázek 4.3). Jedná se po o požadavek na přihlášení na server a požadavek na ověření algoritmu pro generování dvojice klíčů pro asymetrické šifrování.

Požadavky na zpracování dat

Požadavky na zpracování dat jsou z hlediska účelu serveru jeho nejdůležitější součástí. Rozhraní *IRequest* implementuje abstraktní třída *DataRequest* (viz obrázek 4.1), která je zodpovědná právě za manipulaci s daty. Od této třídy dále dědí již tři konkrétní třídy reprezentující jednotlivé požadavky.

CipherRequest posíláme serveru v případě, že chceme šifrovat data. Do požadavku je třeba specifikovat algoritmus, kterým chceme šifrovat, reprezentovaný výčtovým typem (enumem) *CipherAlgorithm*, ve kterém jsou specifikované všechny algoritmy, které server podporuje. Druhým argumentem



Obrázek 4.1: Struktura požadavků na zpracování dat

je výčtový typ *BlockCipherMode*, který definuje režim blokových šifer. Pokud posíláme požadavek bez uvedení inicializačního vektoru, musíme za režim z logiky věci zvolit mód *ECB* (viz podpkapitola 3.2.3), který jako jediný inicializační vektor nevyžaduje. Server odesílá zpět na klienta zašifrovaná data.

CipherRequestWithInitVector rozšiřuje třídu *CipherRequest* o inicializační vektor. Z toho vyplývá, že za parametr režim blokových šifer nesmí být zvolený mód *ECB*.

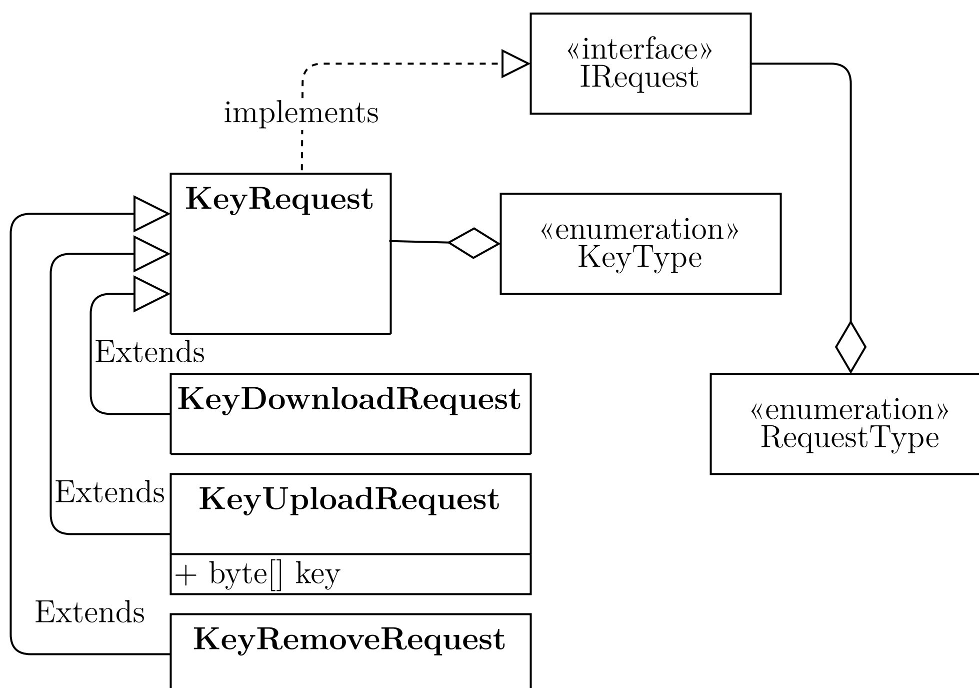
HashRequest reprezentuje požadavek na vytvoření otisku. Jediný parametr kromě dat, který třída vyžaduje, je zvolení algoritmu reprezentovaného výčtovým typem *HashAlgorithm*. Server vrátí klientu otisk zprávy.

SignatureRequest slouží k vyslání požadavku na ověření digitálního podpisu. Tato třída potřebuje kromě dat také otisk, podle kterého bude kontrolu provádět. Znovu je také potřebovat definovat hashovací funkci a algoritmus

použitý pro asymetrickou kryptografii. Zatímco množina algoritmů na vytvoření otisku bude stejná jako u třídy *HashRequest*, množina šifrovacích algoritmů bude oproti třídě *CipherRequest* odlišná. Pro ukázku, algoritmus *DES* je možné využít pouze pro šifrování, protože se jedná o symetrickou šifru. Pro algoritmus *DSA*[3] naopak neexistuje standard, který by podporoval šifrování a dešifrování, tudíž je vhodný pouze pro digitální podpis [9]. Pomocí algoritmu *RSA* je možné šifrovat i vytvářet digitální podpis. Návratovou hodnotou je narozdíl od předchozích požadavků logická hodnota ano/ne.

Požadavky na manipulaci s klíči

Analogicky k požadavkům pro zpracování dat, i v tomto případě máme definovanou abstraktní třídu *KeyRequest* (viz obrázek 4.2), která implementuje specifikace typu klíče, což je nutné u všech třech konkrétních tříd.



Obrázek 4.2: Struktura požadavků pro manipulaci s klíči

KeyDownloadRequest stáhne klíč ze serveru na klienta. Server tedy zpět na klienta posílá data.

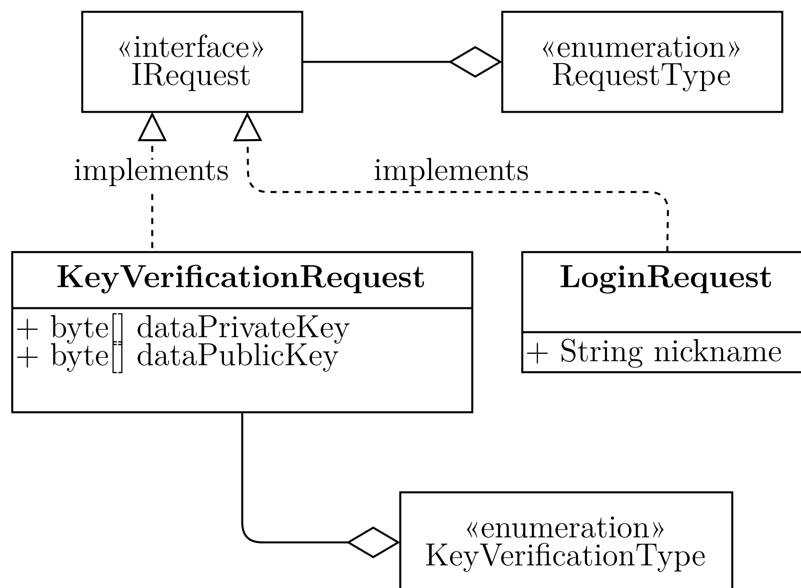
[3]Algoritmus DSA není v rámci této práce implementovaný. Vypsán je pro vysvětlení, že množina algoritmů pro šifrování a digitální podpis je různá.

KeyUploadRequest používáme k nahrání klíče na server. Server neposílá zpět návratovou hodnotu, pouze potvrdí, že byla akce úspěšná.

KeyRemoveRequest odstraní klíč ze serveru. Server opět neposílá zpět žádnou návratovou hodnotu, pouze potvrzuje úspěch akce.

Ostatní požadavky

Posledními dvěma požadavky, které server podporuje, jsou verifikace dvojice klíčů a požadavek na přihlášení (viz obrázek 4.3). Jedná se o jediné dvě konkrétní třídy, které přímo implementují rozhraní *IRequest*.



Obrázek 4.3: Struktura ostatních požadavků

KeyVerificationRequest ověřuje platnost veřejného a soukromého klíče pro asynchronní šifrování. K tomu využívá výčtový typ *KeyVerificationType*, ve kterém specifikujeme, které klíče chceme ověřovat. Jedná se o jinou množinu hodnot, než jsou hodnoty ve výčtovém typu *KeyType* použité pro manipulaci s klíči. Server vrací logickou hodnotu ano/ne.

LoginRequest je možné použít k přihlášení se na server. Tato operace je standardně vykonána během vytváření instance klientské třídy *ClientManager* (viz kapitola 5.1.2). Návratová hodnota opět neexistuje, server jen potvrzuje úspěšné provedení požadavku.

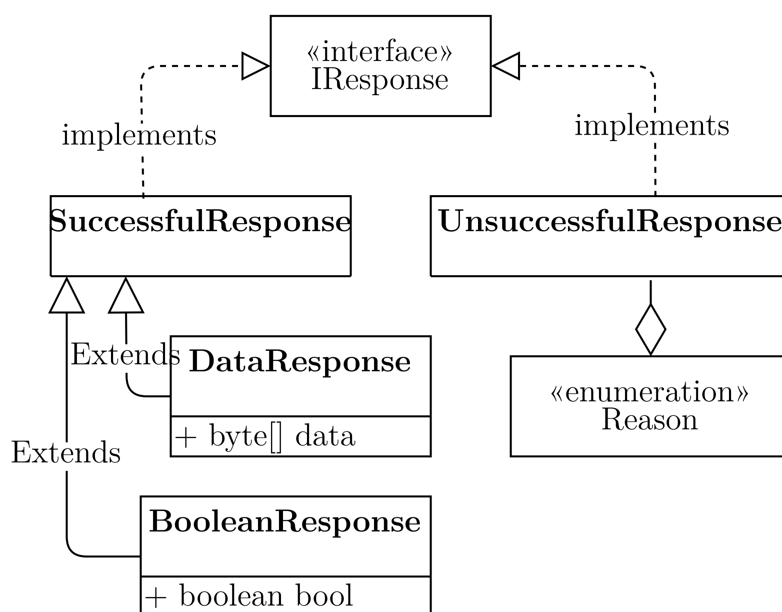
4.2.3 Popis a struktura odpovědi

V minulé kapitole jsme si naznačili obvyklé návratové hodnoty pro jednotlivé požadavky. Tyto návratové hodnoty však byly představeny pouze obecně pro vytvoření si základní představy mechanismu fungování daného požadavku. Nyní si uvedeme jejich konkrétní implementaci.

Typy odpovědí

Základem celé struktury je opět rozhraní, které je pojmenované *IResponse* a všechny konkrétní třídy představující odpověď jej implementují. Rozhraní definuje metodu *boolean isSuccessful()*, která zjišťuje úspěch zpracování požadavku.

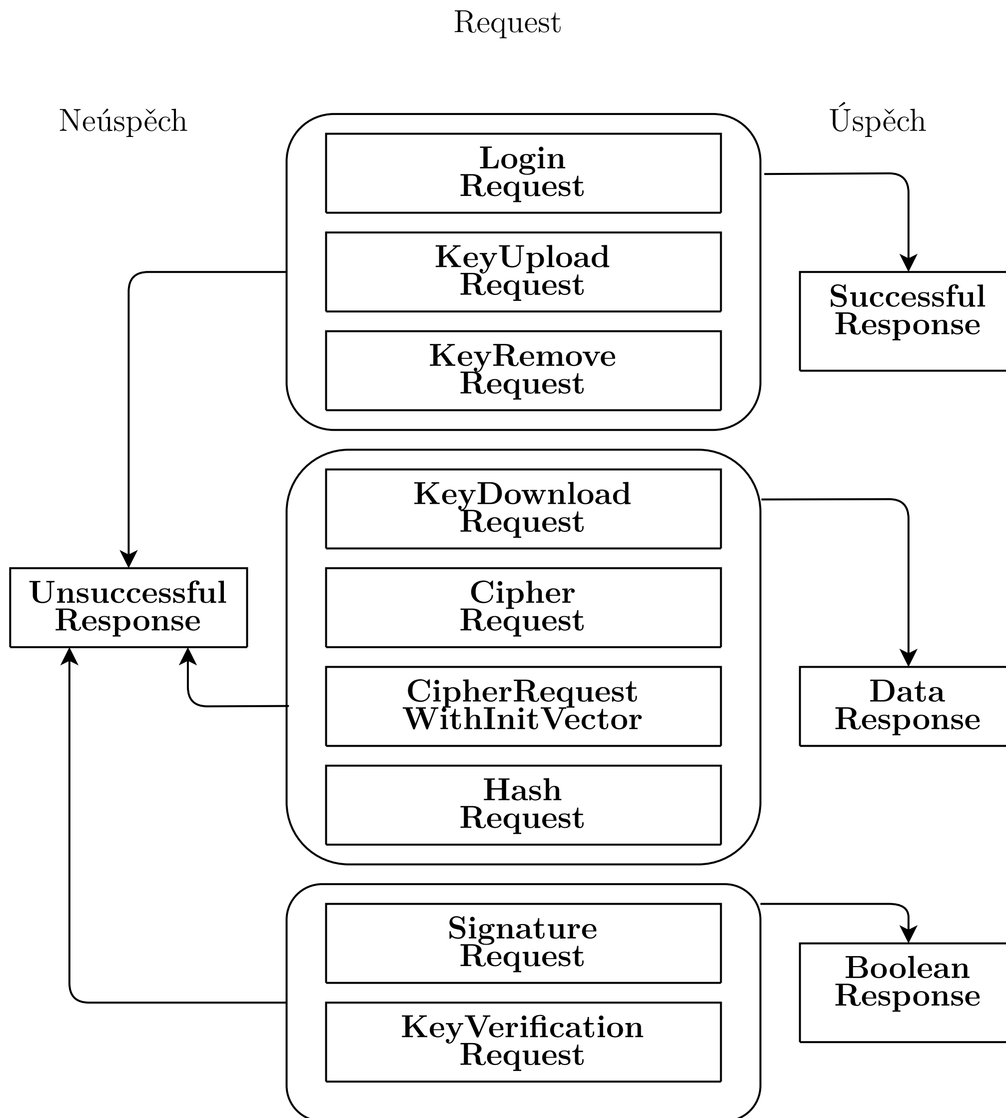
Server dále může obecně na požadavek reagovat jednou ze čtyř druhů odpovědí (viz obrázek 4.4). Základní členění je podle návratové hodnoty metody *boolean isSuccessful()*. Třída *SuccessfulResponse* vrací hodnotu *true*, třída *UnsuccessfulResponse* hodnotu *false*. U některých odpovědí dojde rovněž k předání návratové hodnoty.



Obrázek 4.4: Struktura odpovědí

SuccessfulResponse je jedna ze dvou tříd, které přímo implementují rozhraní *IResponse*. Používáme ho v případě, kdy chceme dát klientu najevo, že jeho požadavek byl úspěšně zpracován, ale nemáme žádnou návratovou

hodnotu, kterou bychom mu předali. Třídou *SuccessfulResponse* reaguje server výhradně na požadavky *LoginRequest*, *KeyUploadRequest* a *KeyRemoveRequest* (viz obrázek 4.5).



Obrázek 4.5: Vztah požadavků a odpovědí

DataResponse rozšiřuje třídu *SuccessfulResponse* o návratovou hodnotu ve formě datového typu *byte[]*. Používán je při úspěšném zpracování požadavku, který vyžaduje jako odpověď jakákoliv obecná data. Třída *DataResponse* je odpovědí na požadavky *KeyDownloadRequest*, *CipherRequest*, *CipherRequestWithInitVector* a *HashRequest* (viz obrázek 4.5).

BooleanResponse analogicky k třídě *DataResponse* přidává k třídě *SuccessfulResponse* návratovou hodnotu datového typu *boolean*. Praktický rozdíl proti třídě *SuccessfulResponse* spočívá v tom, že tato třída přidává jeden možný stav odpovědi - návratovou hodnotu *false*. Tato odpověď je použita v případě ověřování. Požadavek byl zpracován až do konce, ale samotné ověření získaných hodnot dopadlo negativně. Třída *BooleanResponse* je reakcí na požadavky *SignatureRequest* a *KeyVerificationRequest* (viz obrázek 4.5).

UnsuccessfulResponse je poslána serverem v případě, že požadavek nebyl úspěšně zpracován. Důvod, proč byla akce neúspěšná, je uveden ve výčtovém typu *Response*. Třída *UnsuccessfulRequest* je v případě neúspěchu akce možné odeslat jakémukoliv typu požadavku (viz obrázek 4.5).

4.3 Zpracování požadavku serverem

V této podkapitole si naznačíme popis procesu zpracování požadavku serverem. Po připojení klienta na server se spustí metoda *run()* (viz podkapitola 4.1.1). Jedná se tedy o vstupní bod celého vlákna, ze kterého je poté program dále větven.

4.3.1 Přihlášování klienta na server

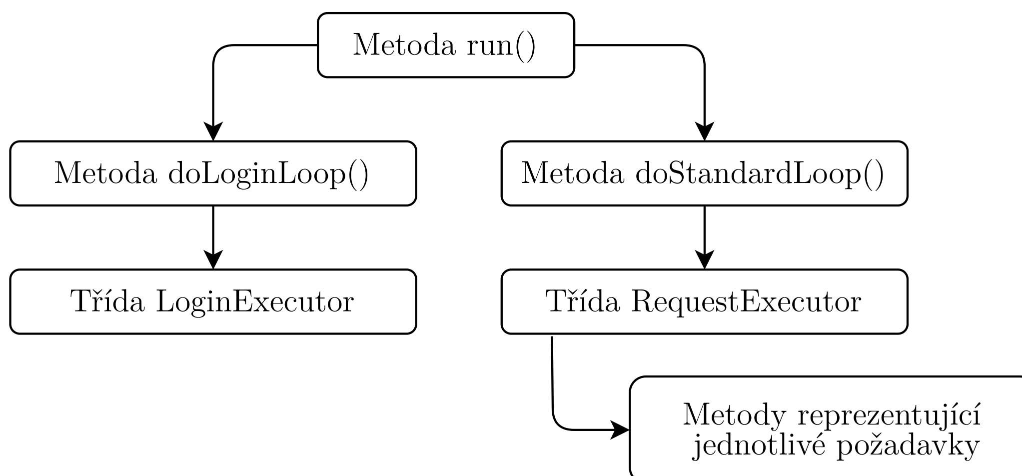
Server rozeznává dva stavy klienta, které jsou reprezentovány vlastními metodami s nekonečnými smyčkami (viz obrázek 4.6). Jsou to:

1. Stav před přihlášením reprezentovaný metodou *doLoginLoop()*.
2. Stav po přihlášení reprezentovaný metodou *doStandardLoop()*.

Smyčka před přihlášením

Smyčka před přihlášením (metoda *doLoginLoop()*) přijímá pouze požadavky třídy *LoginRequest*. V každém jiném případě odesílá na server odpověď *UnsuccessfulResponse*. Požadavek na přihlášení je následovně předán třídě *LoginExecutor*, která rozhodne o validitě přihlášení. Tato třída zkoumá dvě podmínky:

1. Přihlašovací jméno obsahuje pouze tisknutelné znaky.
2. Jiný uživatel se stejným přihlašovacím jménem není přihlášen na serveru.



Obrázek 4.6: Schéma nekonečných smyček

Pokud přihlášení proběhne úspěšně, je klientu odeslána odpověď reprezentována třídou *SuccessfulResponse*, jméno se zapíše do tomu příslušné proměnné a spustí se druhá smyčka.

Smyčka po přihlášení

Po přihlášení je klient oprávněn posílat i ostatní požadavky. Metoda *doStandardLoop()* nejprve ověří, že byl přijat validní objekt[4]. Poté ja zavolána statická metoda *execute()* třídy *RequestExecutor*, která požadavek zpracuje.

Základem zpracování požadavku třídou *RequestExecutor* je konstrukce *switch*, která pomocí klauzule *instanceof* určí typ požadavku. Pro každý typ požadavku je vytvořena vlastní metoda, která požadavek dále zpracovává. Struktura metod se liší dle typu požadavku definovaného v podkapitole 4.2.1. Protože požadavek na přihlášení byl už zpracován třídou *LoginExecutor* zbývají nám dva typy požadavků:

1. Kryptografické požadavky.

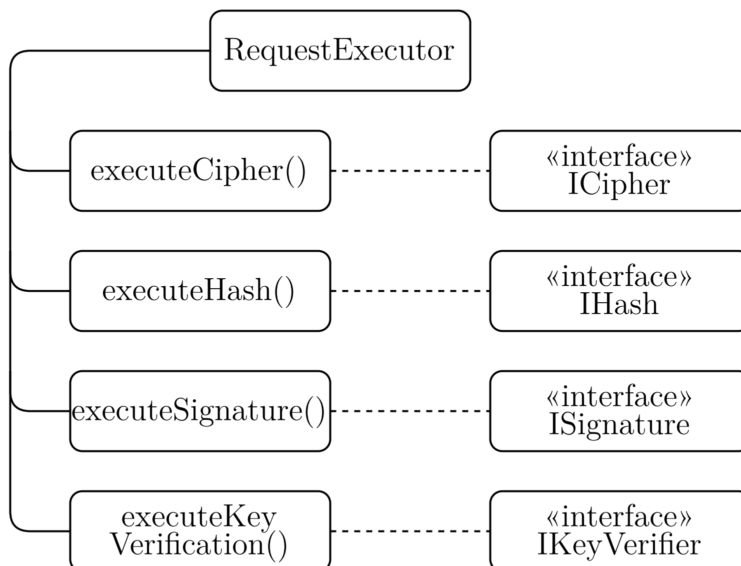
2. Požadavky na manipulaci s klíči.

Třída *RequestExecutor* v případě neúspěšných požadavků převádí vyhozené výjimky z nižších vrstev na výčtový typ *REASON*, který je součástí třídy *UnsuccessfulResponse*.

[4]Objekt se nerovná hodnotě *null*.

4.3.2 Zpracování kryptografických požadavků

Tato část programu je dekomponována následujícím způsobem. Pro každý ze čtyř kryptografických požadavků (viz podkapitola 4.2.1) je vytvořena vlastní metoda v třídě *RequestExecutor* a vlastní rozhraní v balíku (package) *server.decrypt* (viz obrázek 4.7).



Obrázek 4.7: Kryptografické metody třídy RequestExecutor

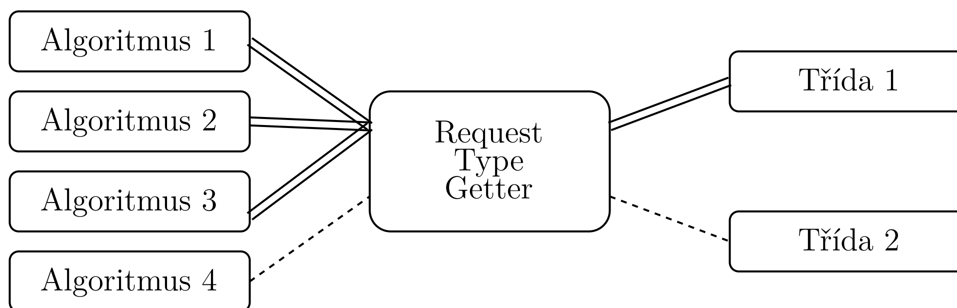
Výčtový typ požadavku (viz obrázek 4.1) a konkrétní třída implementující dané rozhraní reprezentují konkrétní algoritmus, který bude použit. Obě tyto reprezentace konkrétních algoritmů jsou propojeny pomocí třídy *RequestTypeGetter*, která obsahuje pro každý typ požadavku metodu s konstrukcí *switch*(viz ukázka kódu 4.5), které dle daného výčtového typu navrátí zpět konkrétní třídu, ve které je obsažena implementace daného algoritmu.

```
public class RequestTypeGetter {

    static ICipher getCipher(CipherAlgorithm algorithm, String clientName) {
        switch(algorithm) {
            case DES:
            case AES:
            case TRIPLE_DES:
            case BLOWFISH:
                return new JavaBlockCipher(algorithm, clientName);
            case RSA:
                return new JavaRSA(clientName);
            case CAESAR:
                return new CaesarCipher(clientName);
            case KNAPSACK:
                return new KnapsackCipher(clientName);
            default:
                return null;
        }
    }
    ...
}
```

Ukázka kódu 4.5: Příklad metody třídy RequestTypeGetter

Jak již je v kódu naznačeno, nemusí platit, že jeden algoritmus je reprezentovaný právě jednou třídou (viz obrázek 4.8). V případě toho programu například nedává smysl, abychom pro každou blokovou šifru, kde pouze předáváme parametry třídě *Cipher* knihovny *javax.crypto* vytvářeli vlastní třídu. Konkrétní algoritmus je tedy specifikován jako parametr konstruktoru třídy.



Obrázek 4.8: Naznačení principu třídy RequestTypeGetter

Z toho vyplývá, že pro přidání další implementace již existujícího požadavku (například nového algoritmu pro hashování) je potřeba vykonat následující kroky:

1. Vytvořit třídu implementující dané rozhraní v balíku *server.decrypt*, která provede požadovanou akci.

2. Vytvořit záznam v odpovídajícím výčtovém typu.

3. Propojit daný záznam s danou třídou v odpovídající metodě třídy *RequestTypeGetter*.

Přidání nového požadavku by pravděpodobně vyžadovalo více akcí, kterého by závisely na podobě tohoto požadavku. Nutná podmínka toho vytvoření by ovšem vždy byla:

1. Vytvořit požadavek implementující rozhraní. *IRequest*

V případě, že by se navíc jednalo o kryptografický požadavek a programátor by respektoval již vytvořenou strukturu (mohlo by se jednat například o kombinaci symetrického a asymetrického šifrování naznačenou v podkapitole 3.2.2), vytvoření by zahrnovalo ještě následující kroky:

2. Vytvoření nového rozhraní v balíku *server.decrypt*.
3. Vytvoření nové metody v třídě *RequestExecutor* podle již vytvořených tříd.
4. Přidání metody do konstrukce *switch* v metodě *execute()* třídy *RequestExecutor*.
5. Vytvoření nové metody v třídě *RequestTypeGetter*, která zachovává strukturu stávajících metod (viz ukázka kódu 4.6).
6. Opakovat postup pro přidání implementace do již existujícího požadavku pro každý algoritmus, který bude nový požadavek podporovat.

```
public class RequestExecutor {

    ...

    private static IResponse executeHash(HashRequest request) {

        IResponse response;
        IHash hash = RequestTypeGetter.getHash(request.getAlgorithm());
        assert hash != null;

        try {
            byte[] rvalue = hash.execute(request.getData());
            response = new DataResponse(rvalue);
        } catch (Exception e) {
            response = new UnsuccessfulResponse(
            Reason.ALGORITHM_ERROR);
        }

        return response;
    }
    ...
}
```

Ukázka kódu 4.6: Příklad metody třídy RequestExecutor

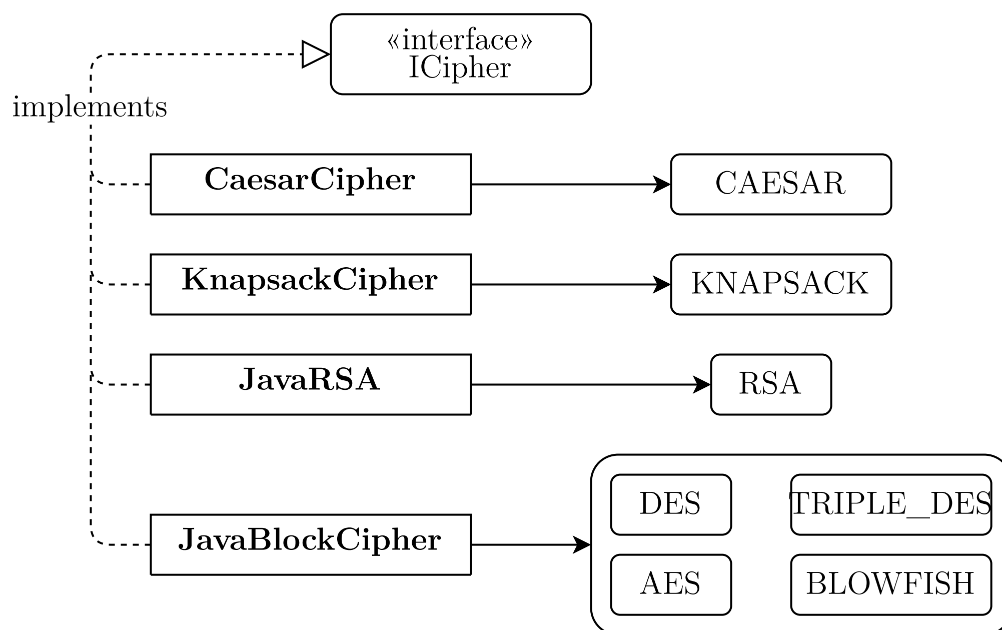
4.3.3 Vykonání kryptografických požadavků

V předchozí podkapitole bylo ukázáno, jakým způsobem server zpracuje požadavek a deleguje vykonání na konkrétní třídu implementující rozhraní z balíku *server.decrypt*. V této podkapitole bude představena konkrétní implementace těchto tříd.

Požadavek na šifrování

Sjednocujícím prvkem pro všechny algoritmy, které implementují šifrování je rozhraní *ICipher*. Každá třída, která toto rozhraní implementuje, musí obsahovat metody *executeWithVector()* a *executeWithoutVector()* v závislosti na uvedení nebo neuvedení inicializačního vektoru. V případě, že šifra nepodporuje inicializační vektor, dojde k vyhození (throw) tomu odpovídající výjimky.

Podporované šifry jsou uvedeny na obrázku 4.9.



Obrázek 4.9: Rozhraní ICipher

Třída CaesarCipher vykonává jednoduchou *Caesarovu šifru*, která každé malé písmeno nahradí jiným písmenem vzdálené o definovaný posun. Tato šifra je určena pro vyzkoušení si programu na jednoduché symetrické šifře.

Třída KnapsackCipher slouží pro šifrování pomocí *Zavazadlového algoritmu*. Tato třída slouží ke stejnému účelu jako *Caesarova šifra* pro asyme-

trickou kryptografii. V rámci návodu uvedeného v souboru přiloženém ke zdrojovému kódu *tutorial2.txt* je implementována třída *KnapsackAlgorithm*. Totožná třída se nachází v serverové části a třída *KnapsackCipher* na ní pouze deleguje činnosti.

Třída JavaRSA šifruje pomocí algoritmu *RSA*. K tomu je využita třída *Cipher* a postup uvedený v podkapitole 3.2.3. Algoritmus *RSA* musí být od odstatních, jazykem Java podporovaných, algoritmů oddělen z důvodu jiné konverze klíče z datového typu *byte[]* na objekt *Key*, který je potřebný pro spuštění algoritmu.

Třída JavaBlockCipher implementuje ostatní šifrovací algoritmy, které knihovna *javax.crypto* podporuje. Konkrétně se jedná o:

1. *DES*
2. *AES*
3. *TRIPLE_DES*
4. *BLOWFISH*

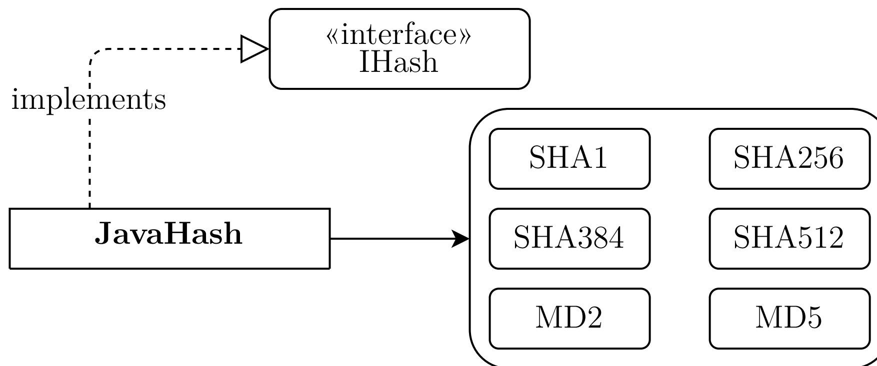
Třída *JavaBlockCipher* opět pouze deleguje parametry na třídu *Cipher*, která vykoná samotné šifrování.

Požadavek na hashování

Základem pro implementaci hashování je opět rozhraní *IHash*, které obsahuje metodu *execute()*. Toto rozhraní ale tentokrát implementuje pouze jediná třída (viz obrázek 4.10).

Třída JavaHash deleguje samotné provedení algoritmu na třídu *MessageDigest* z knihovny *javax.crypto*, k čemuž využije postup uvedený v podpkapitole 3.3.2. Podporované algoritmy jsou:

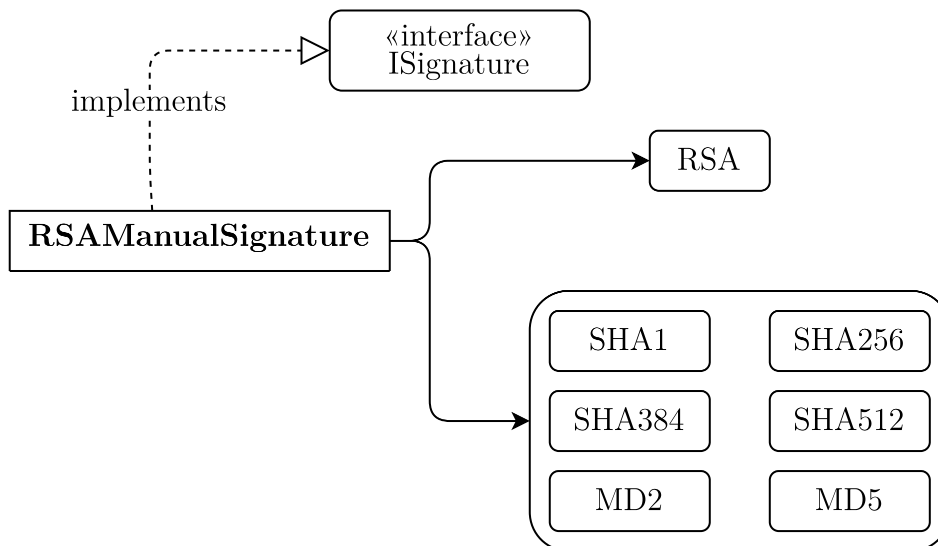
1. *SHA1*
2. *SHA256*
3. *SHA384*
4. *SHA512*
5. *MD2*
6. *MD5*



Obrázek 4.10: Rozhraní IHash

Požadavek na ověření digitálního podpisu

Rozhraní *ISignature* definuje metodu *verify()*, kterou konkrétní třídy ověřující digitální podpis musí implementovat. Toto rozhraní je v programu znovu implementováno pouze jedinou konkrétní třídou (viz obrázek 4.11).

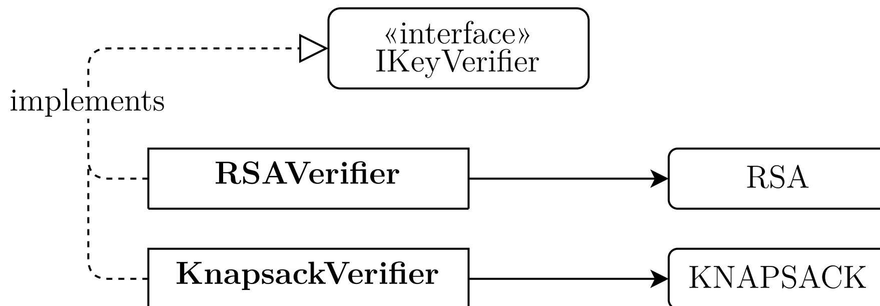


Obrázek 4.11: Rozhraní ISignature

Třída RSAManualSignature nevyužívá třídu *Signature* z knihovny *javax.crypto*. Důvodem je využití paddingu *PKCS*, které sice zajišťuje větší bezpečnost, algoritmus je však poté mnohem hůře zreprodukovatelný. Vzhledem k tomu, že primárním účelem práce je framework pro studenty učící se kryptografické algoritmy, jsou využity již existující algoritmy pro asymetrickou kryptografii a hashování, na které není aplikovaný žádný padding a které jsou aplikovány dle schématu na obrázku 3.7 [8].

Požadavek na ověření validity klíčů pro asymetrickou kryptografii

Posledním rozhraním z balíku *server.decrypt* je rozhraní *IKeyVerifier* definující metodu *verify()*. Toto rozhraní implementují dvě třídy (viz obrázek 4.12).



Obrázek 4.12: Rozhraní IKeyVerifier

Třída RSAVerifier ověřuje validitu klíčů pro algoritmus *RSA*. Třída vygeneruje náhodný blok dat, které se pokusí pomocí zvolených klíčů zašifrovat a dešifrovat. Pokud se výsledek rovná původnímu bloku dat, je akce považována za úspěšnou.

Třída KnapsackVerifier validuje klíče *Zavazadlového algoritmu*. Princip fungování je stejný jako v případě třídy *RSAVerifier*. Třída *KnapsackVerifier* rovněž deleguje činnosti na třídu *KnapsackAlgorithm*, v níž je definován celý *Zavazadlový algoritmus*.

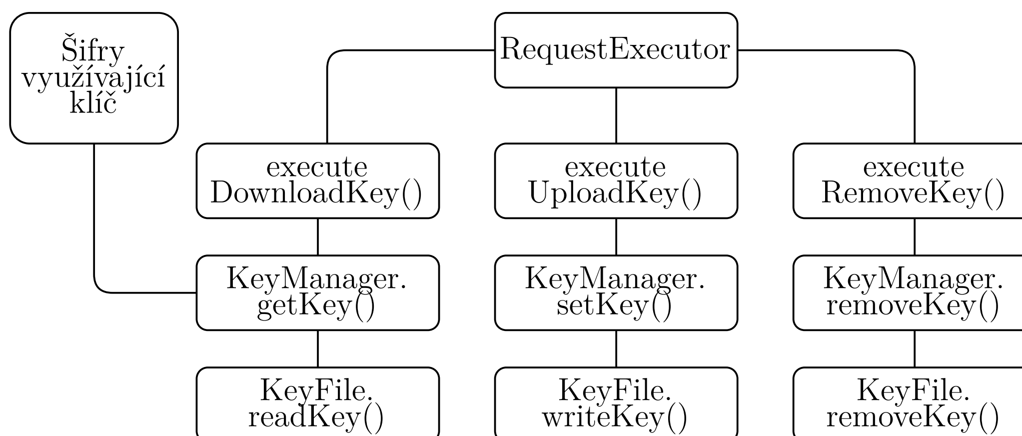
4.3.4 Správa klíčů

Druhou kategorií požadavků jsou požadavky na manipulaci s klíči. V kontextu třídy *RequestExecutor*. Metody opět korespondují s typy požadavků uvedených v podkapitole 4.2.1. Celé schéma je na obrázku 4.13.

Celý proces je rozdělen do tří vrstev, pomocí kterých je dekomponován celý problém. Pro snadnější pochopení bude celý systém vysvětlený od nejnižší vrstvy, kterou představuje třída *KeyFile*.

Třída KeyFile

Třída *KeyFile* je koncipována jako nejnižší vrstva správy klíčů. Třída je navržena tak, aby byla možna přenést na klienta, nebo do jakéhokoliv jiného programu, kde by bylo potřeba ukládat klíče do souborů. Třída má dvě základní funkce, jsou to:



Obrázek 4.13: Schéma správy klíčů

1. Fyzický zápis nebo čtení na disk pomocí knihovny *java.io*, které zahrnuje operace typu ověření existence složky, do které chceme zapisovat, ověření práv k souboru a samozřejmě samotnou operaci čtení, zápis nebo smazání.
2. Přidání značky na začátek a konec souboru, která obsahuje název algoritmu.

Třída má tři základní paremtry a inicializuje se dvěma způsoby:

1. V konstruktoru třídy jsou uvedeny všechny tři parametry, tedy cesta k souboru, název algoritmu a data v tisknutelné podobě. V takovém případě je třída *KeyFile* připravena zapsat uvedená data.
2. Uvedeny jsou pouze dva parametry, konkrétně cesta k souboru a název algoritmu. V tomto případě je možné zavolat metody pro získání nebo odstranění klíče.

Třída KeyManager

Třída *KeyManager* využívá metod třídy *KeyFile* a vkládá do nich parametry z kontextu tohoto programu. Třída má opět dvě základní funkce:

1. Převod dat mezi binární a tisknutelnou podobou pomocí kódování *Base64*.
2. Zjištění cesty k souboru pomocí parametru uvedeného v konfiguraci programu.

Tuto třídu kromě metod zpracovávající požadavky pro manipulaci s klíči z třídy *RequestExecutor* využívají rovněž kryptografické třídy pracující s klíči.

Metody třídy RequestExecutor

Tyto metody jsou analogií ke kryptografickým metodám z téže třídy. Jejich účelem je delegace požadavku na třídu *KeyManager* a v případě neúspěchu převod výjimek na výčtový typ *REASON* třídy *UnsuccessfulResponse.*

5 Implementace klientské části

Při popisu klientské části programu je třeba mít na paměti skutečnost, že program je koncipován primárně pro studenty, kteří budou do klientské části programovat implementaci některých podporovaných šifer. Z tohoto důvodu klientská část programu obsahuje pouze rozhraní pro komunikaci se serverem, testy a validátor. Studentům bude poté předložen program bez většiny současných testů, přičemž předpokládáme, že studenti doprogramují šifry ve formě jiných testů.

5.1 Rozhraní pro komunikaci se serverem

Rozhraní pro komunikace se serverem navazuje na podkapitolu 2.2.3, ve které je představena jednoduchá TCP aplikace a podkapitolu 2.4.2, kde je naznačeno použití objektového protokolu.

Vzhledem ke skutečnosti, že rozšíření TCP aplikace je možné zejména na straně serveru, zůstalo komunikační rozhraní klienta velice jednoduché. I tak lze problematiku rozdělit do dvou vrstev:

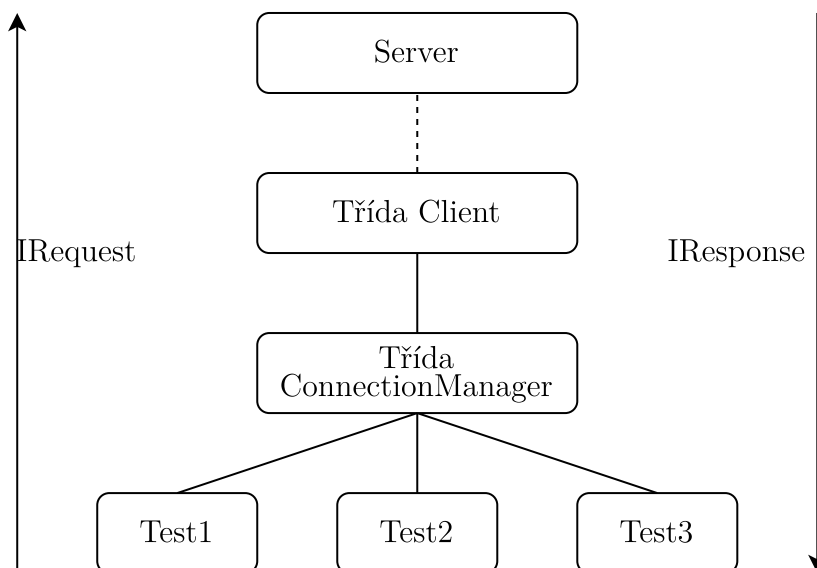
1. Nezbytná funkcionalita pro síťovou komunikaci (třída *Client*).
2. Rozšířená funkcionalita spojená zejména s přihlašováním na server (třída *ConnectionManager*).

Celé schéma komunikace se serverem je zobrazeno na obrázku 5.1. Hlavní funkci klientské části programu plní jednotlivé jednotkové testy.

5.1.1 Spojení se serverem

Nezbytnou funkcionalitu pro síťovou komunikaci představující nižší z vrstev rozhraní pro komunikaci na straně klienta implementuje třída *Client* z balíku *client.main*. Tato třída dokáže:

1. Vytvářet a ukončovat spojení se serverem.
2. Posílat požadavky a přijímat odpovědi.



Obrázek 5.1: Schéma rozhraní pro komunikace se serverem

Vytváření a ukončení prakticky kopíruje jednoduchou aplikaci uvedenou v teoretické části práce (viz podkapitola 2.2.3).

Odesílání a příjímání objektů je rozšířeno o konverzi odpovědi na rozhraní *IResponse* (viz ukázka kódu 5.1). Výjimka *IOException* představuje výpadek spojení, zatímco výjimka *IllegalResponseException* znamená, že server vrátil neplatný objekt. O zpracování výjimek se stará druhá vrstva rozhraní.

```java
import java.net.*;
import java.io.*;

public class Client{
    private ObjectOutputStream out;
    private ObjectInputStream in;

    ...

    public IResponse sendObject(IRequest request) throws
    IOException, IllegalResponseException {
        out.writeObject(request);

        Object obj = null;
        try {
            obj = in.readObject();
        } catch (ClassNotFoundException e) {
            throw new IllegalResponseException(
            "Server has returned unknown class.");
        }

        if(obj instanceof IResponse) {
            return (IResponse) obj;
        } else {
            throw new IllegalResponseException(
            "Server hasn't returned the Response object.");
        }
    }
}
```

Ukázka kódu 5.1: Odesílání požadavků a přijímání odpovědí

5.1.2 Rozšířená funkcionalita

Druhá vrstva rozhraní je reprezentována třídou *ConnectionManager*, která využívá metod třídy *Client*, a ke kterým přidává tyto tři funkce:

1. Automatické poslání požadavku na přihlášení na server.
2. Vygenerování uživatelského jména, pokud si nižší vrstva nevyžádá vlastní jméno.
3. Převod výjimek, které vyhazuje třída *Client*, na výčtový typ *Reason* třídy *UnsuccessfulResponse*.

Třída má zároveň přednastavenou implicitní adresu serveru a port, které ovšem vyšší třída[1] může přepsat tím, že je uvede do konstruktoru třídy. V kombinaci s možností si zvolit nebo vygenerovat uživatelské jméno, lze třída inicializovat čtyřmi způsoby.

[1] Tedy třída obsahující jednotkový test.

Požadavek na přihlášení je vyslán automaticky po úspěšném připojení se k serveru. Uživatelské jméno je generováno jako náhodný alfanumerický řetězec dlouhý osm znaků.

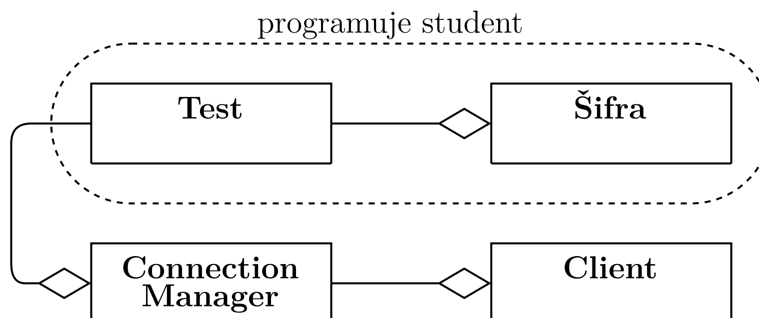
5.2 Testy

Na testy v tomto programu lze nahlížet ze dvou úhlů pohledu. První úhel pohledu už byl naznačen v předcházející podkapitole, kde testy představovaly vyšší vrstvu vůči rozhraní pro komunikaci se serverem. To znamená, že testy v tomto případě nahrazují uživatelské rozhraní nebo spuštění programu s parametry příkazové řádky, které by byly pro účel této práce nepraktické. Druhý úhel pohledu na testy je standardní účel jednotkových testů, tedy ověření funkčnosti programu.

5.2.1 Testy jako nejvyšší vrstva klientské části programu

V testech je koncentrována hlavní funkcionalita klientské části, protože právě v nich definujeme jaké požadavky s jakými parametry odešleme na server a jak se zachováme k návratové hodnotě.

Typické zamýšlené užití programu je takové, že student vytvoří šifru s určitým volně definovaným rozhraním, kterou spustí a ověří právě pomocí testu (viz obrázek 5.2).



Obrázek 5.2: UML diagram klienta bez použití validátoru

Implementace algoritmu DES

Pro ověření funkčnosti tohoto návrhu byla v balíku *client.cipher* implementována šifra *DES* stejným způsobem. Testovací třída *DESTest* se nachází

v testovacím balíku *Test*. Třída kromě ověření funkčnosti řešení zároveň naznačuje předpokládané užití programu. Šifra byla naprogramována dle standardu Národního institutu standardů a technologií při ministerstvu obchodu USA [22].

5.2.2 Testy ověřující funkčnost programu

Druhým pohledem na testy je ověření správného fungování převážně serverové části. Podobně jako v případě testovací třídy *DESTest* i nyní kromě validity řešení dáváme příklad toho, jakým způsobem je možné odeslat data na server. V programu byly testovány následující dvě situace:

1. Server navrátí třídu *SuccessfulResponse* se správnými parametry při odeslání validního požadavku.
2. Server navrátí třídu *UnsuccessfulResponse* se správnou hodnotou výčtového typu *Reason*, pokud zadáme nevalidní parametry požadavku.

Návratová hodnota serveru je ověřována proti použití knihovny *javax.crypto* na straně klienta. V takovém případě sice testujeme dvě stejné implementace šifer proti sobě, předpokládáme ovšem, že knihovna *javax.crypto* je natolik odladěná, že není třeba jí znovu testovat. Toto řešení tak ověří funkčnost celého procesu odeslání požadavku, jeho zpracování a správné dosazení do totožné serverové třídy.

V případě testování hodnoty výčtového typu *Reason* jsou testovány pouze ty scénáře, které způsobíme zvolením neplatných parametrů. Scénáře, které vyžadují nějakou externí akci, jako je testování hodnoty při výpadku spojení, nebo při nedostatečných právech na zápis souboru nejsou testována, protože v kontextu jednoho testu nemusí mít vždy stejný výsledek.

5.2.3 Odhalené problémy v průběhu testování

Problémy, které byly nalezeny při testování souvisí zejména s knihovnou *javax.crypto* a byly dány především tím, že knihovna je primárně určena k použití kryptografie v praxi, což účelu tohoto programu může někdy škodit.

Se symetrickými šiframi a hashovacími funkcemi nebyl nalezen žádný problém. Výstupy těchto algoritmů odpovídaly náhodně vybraným výsledkům algoritmů třetích stran, včetně vlastní implementace šifry *DES* zmíněné v předchozích podkapitolách.

Až výstupy z algoritmů, které využívají asymetrickou kryptografii začaly vracet jiné hodnoty, než bylo původně očekáváno. To se týká jak třídy *Signature*, tak použití algoritmu *RSA* v rámci třídy *Cipher*.

Problém s třídou Signature

Očekávané chování třídy Signature bylo takové, že návratová hodnota po provedení algoritmu bude stejná, jako při použití algoritmů na asymetrické šifrování a hashování zvlášť aplikovaných dle schématu na obrázku 3.7.

Bohužel se nepodařilo návratovou hodnotu třídy *Signature* zreprodukovat ani při použití všech kombinací parametrů, se kterými lze třídy *Cipher* a *MessageDigest* spustit, ani při použití algoritmů na asymetrické šifrování a hashování třetích stran.

Z tohoto důvodu algoritmus implementovaný v serverové části aplikace dosazuje zadané hodnoty do tříd *MessageDigest* a *Cipher* a nevyužívá třídu *Signature*.

Problémy s algoritmem RSA

I třída *Cipher*, pokud byl vybrán algoritmus *RSA*, vykazovala předem neočekávané chování. Obecně lze algoritmus *RSA* inicializovat se dvěma druhy paddingu (vysvětlení pojmu padding se nachází v podkapitole 3.2.3). Jedná se o:

1. Inicializace bez jakéhokoliv paddingu (zadáno *NoPadding*).
2. Inicializace s paddingem *PKCS1* (zadáno *PKCS1Padding*).

Třetím způsobem inicializace algoritmu *RSA* je neuvedení žádného paddingu, tedy spuštění šifry *Cipher* bez dalších parametrů. V takovém případě je překvapivě implicitně vybrán padding *PKCS1*, který přidává do prostého textu náhodné znaky. Tím se šifrování pomocí algoritmu *RSA* s tímto paddingem stává nedeterministické, protože výstupem může být pro stejný prostý text různý šifrovaný text. Padding je samozřejmě navržen tak, že v případě dešifrování šifrovaných textů je výstupem opět stejný prostý text. Ačkoliv tento padding zabezpečuje šifru, toto chování je pro náš program nežádoucí.

I žádný padding v algoritmu *RSA* má neočekávané konsekvence. Pomocí paddingu *PKCS1* je totiž do prostého textu zároveň vložena informace o délce prostého textu. Tato informace se tedy bez použití tohoto paddingu ztrácí. To znamená, že pokud necháme algoritmem *RSA* zašifrovat a následně dešifrovat prostý text o hodnotách jednotlivých bytů *[0, 0, 0, 100]*, výsledkem takové operace bude pouze jednoznakový text *[100]*, protože algoritmus nemá žádný způsob, jak zjistit původní velikost textu. Proto je nezbytné při jakémkoliv užití dešifrovacího algoritmu šifry *RSA* znát původní velikost textu a výsledek algoritmu rozšířit na požadovanou délku bytů.

Posledním nalezeným problém s šifrou *RSA* je skutečnost, že maximální možná velikost dat, kterou je možné šifrovat, je přímo úměrná velikosti klíče[2]. To komplikuje zejména algoritmus ověřování validity klíčů pro šifru *RSA*, kde neznáme velikost klíčů a potřebujeme generovat náhodný blok dat, na kterém bude otestována validita klíčů. Pokud vyjdeme z informace, že nejmenší oficiálně podporovaná délka *RSA* klíče je 512 bitů, velikost dat po dosazení do vzorečku pro výpočet maximální délky dat šifry RSA dle velikosti klíče činí 43 bitů. Maximální možná velikost náhodně generovaných dat pro účel ověření validity klíčů bude tedy 5 bytů, protože jakýkoliv delší blok už by nemuselo být možné klíčem dlouhým 512 bitů zašifrovat.

5.3 Validátor

Posledním prvkem klientské části programu je validátor, který rozšiřuje část testování algoritmů. Do této chvíle jsme předpokládali situaci, kdy student vytvoří své testy se svými klíči a daty, které poté posílá na server. V jednu chvíli se tedy testovalo jen jedno použití algoritmu s jedním prostým textem. Cílem validátoru je vytvořit rozhraní pro šifry, aby bylo možné testovat šifru s nezávislými daty a klíči. Pokud tedy test validátoru vyjde úspěšně, pravděpodobnost, že je šifra implementována správně je mnohem vyšší.

5.3.1 Princip fungování

Prvním krokem procesu validace je vygenerování náhodného klíče pro danou šifru. Následně je generován čím dál tím delší náhodný prostý text, který slouží jako vstup do lokálních (testovaných) algoritmů. Princip vyhodnocení správnosti výsledků lokálních algoritmů se liší v závislosti na typu algoritmu:

1. Pro symetrické šifry jsou data zašifrována pomocí lokálního algoritmu, stejná data jsou poslána na server. Test je platný, pokud je výsledek obou algoritmů stejný.

2. Pro asymetrické šifry je kromě šifrování testováno i dešifrování. Ke stejnému procesu jako v případě symetrických šifer je přidán krok, kdy se výsledná data dešifrují a výsledek je porovnáván s původními daty.

3. Hashovací funkce jsou testovány stejně jako symetrické šifry.

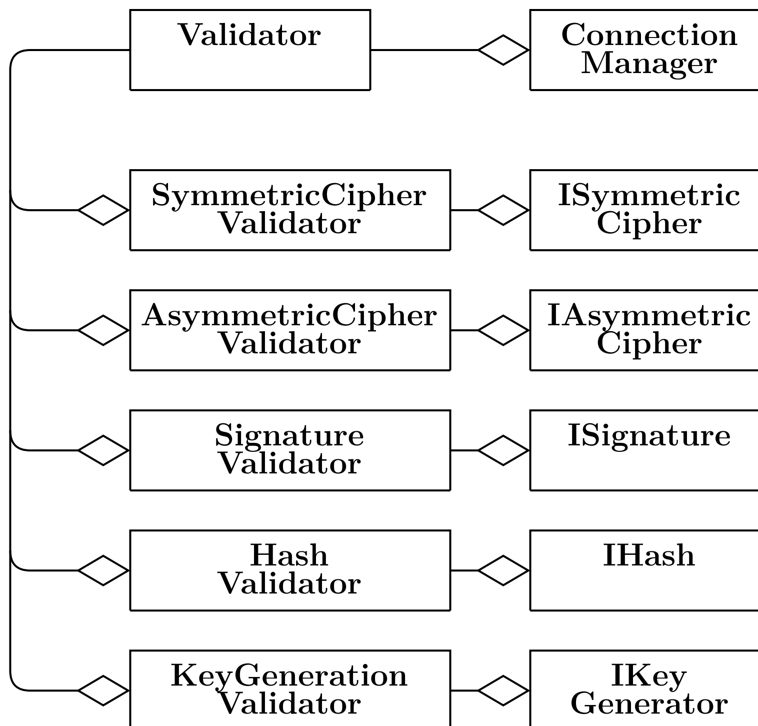
[2]Konkrétní vzoreček pro výpočet maximální velikosti šifrovaných dat je $\lfloor n/8 \rfloor - 11$, kde proměnná n znamená velikost klíče [24].

4. Pro algoritmus digitálního podpisu je testována část podepisování. Náhodná data jsou podepsána a výsledek zkontrolován na serveru.

5. Pro algoritmus ověření generování dvojice asymetrických klíčů je implementován proces podobný kontrole digitálního podpisu. Klíče jsou vygenerovány a zkontrolovány pomocí serveru.

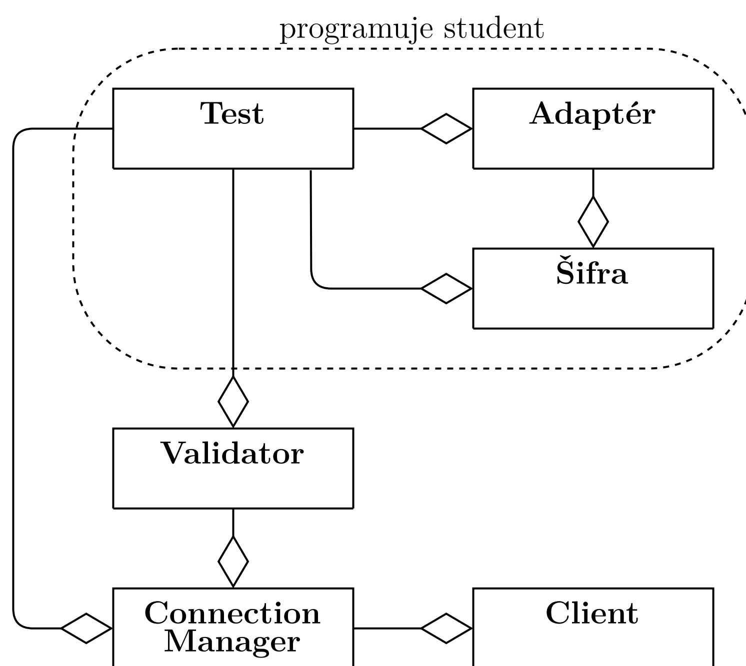
5.3.2 Implementace validátoru

Základem fungování validátoru je stejnojmenné třída *Validator*, která obsahuje přetíženou metodu *validate()*. Parametrem metody *validate()* je rozhraní představující implementaci šifry. Implementace validace je delegována dle rozhraní na příslušnou validační třídu, která pomocí třídy *ConnectionManager* odesílá požadavky na server. UML diagram celého systému je znázorněn na obrázku 5.4.



Obrázek 5.3: UML diagram klienta s užitím validátoru

Doporučeným způsobem implementace rozhraní je návrhový vzor adaptér. Adaptér bude implementovat zvolené rozhraní a delegovat jednotlivé požadavky na již naprogramovanou kryptografickou třídu. UML diagram celé klientské části po použití toho způsobu je uveden na obrázku 5.4.



Obrázek 5.4: UML diagram klienta s užitím validátoru

6 Závěr

Cílem práce byla implementace aplikace typu klient-server umožňující šifrování a přenos dat. Při implementaci byl kladen velký důraz na rozšiřitelnost a znovupoužitelnost. Tomu odpovídá granularita modulů a komunikace prostřednictvím rozhraní. Díky tomu lze snadno měnit jednotlivé moduly či přidávat další. Architektura typu klient-server přináší možnost komunikace s více klienty najednou.

V aktuální verzi programu, pracuje framework pouze s třídami z knihovny *javax.crypto*. Aplikace je však oddělena na vrstvy takovým způsobem, že dosáhnout přechodu na jinou knihovnu, případně na kombinaci více knihoven, je jednoduše realizovatelné.

Teoretické kapitoly, zabývající se síťovou částí práce, společně s navazující implementační praktickou příručkou vytváří dostatečný základ pro vytváření dalších síťových aplikací v jazyce Java.

Práce byla vytvořena v programovacím jazyce Java, proto byl zvolen protokol založený na posílání a přijímání objektů tohoto jazyku. Možným rozšířením této práce by mohla být podpora dalších forem komunikace např. REST API, která by zaručila větší přenositelnost pro jiné (např. mobilní) platformy a další programovací jazyky.

Vytvořený framework splňuje zadání v celém jeho rozsahu. Umožňuje výměnu dat šifrovaných symetrickým a asymetrickým algoritmem, obsahuje základní hashovací funkce a jejich spojením s asymetrickou kryptografií umožňuje simulaci digitálního podpisu a kontroly integrity dat. Největší jeho výhodou však je efektivnost, s jakou lze validovat a testovat jednotlivé existující či nově vytvořené moduly. Díky vhodně zvolené architektuře lze tento nástroj používat pro účely výuky a implementaci semestrálních prací v předmětu KIV/BIT.

Literatura

[1] *Block Ciphers and Modes of Operation* [online]. CommonLounge. Dostupné z: https://www.commonlounge.com/discussion/6747358d828a45c99f61f4c09ff2f371.

[2] *What Is a Digital Signature?* [online]. Sectigo. Dostupné z: https://www.instantssl.com/digital-signature.

[3] *What is a Digital Signature How Do I Create One?* [online]. DocuSign, 2019. Dostupné z: https://www.docusign.com/how-it-works/electronic-signature/digital-signature/digital-signature-faq.

[4] What are Hash Functions and How to choose a good Hash Function? *GeeksforGeekM*. August 2019. Dostupné z: https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/.

[5] Java IO - javatpoint. *www.javatpoint.com*. . Dostupné z: https://www.javatpoint.com/java-io.

[6] Character and Byte Streams. *https://docs.oracle.com/*. . Dostupné z: https://docs.oracle.com/javase/tutorial/i18n/text/stream.html.

[7] Hashovací funkce. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=7029.

[8] *Java^{TM} Cryptography Architecture Standard Algorithm Name Documentation* [online]. Oracle. Dostupné z: https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html.

[9] *Public Key Security* [online]. Oracle. Dostupné z: https://docs.oracle.com/cd/E13203_01/tuxedo/tux71/html/secovr10.htm.

[10] An Overview of Symmetric Encryption and the Key Lifecycle. *Cryptomathic*. March 2019. Dostupné z: https://www.cryptomathic.com/news-events/blog/an-overview-of-symmetric-encryption-and-the-key-lifecycle.

[11] TCP vs. UDP: Understanding the Difference. *Private Internet Access Blog*. December 2018. Dostupné z: https://www.privateinternetaccess.com/blog/tcp-vs-udp-understanding-the-difference/.

[12] Debnath. Understanding Byte Streams and Character Streams in Java. *Developer.com*. April 2018. Dostupné z:

https://www.developer.com/java/data/understanding-byte-streams-and-character-streams-in-java.html.

[13] *Class Cipher* [online]. Oracle, 2018. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html.

[14] *Class DatagramPacket* [online]. Oracle, 2018. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/net/DatagramPacket.html.

[15] *Class DatagramSocket* [online]. Oracle, 2018. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html.

[16] *Class MessageDigest* [online]. Oracle, 2018. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html.

[17] *Class ServerSocket* [online]. Oracle, 2018. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html.

[18] *Class Signature* [online]. Oracle, 2018. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/security/Signature.html.

[19] *Class Socket* [online]. Oracle, 2018. Java SE Documentation. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html.

[20] Kabelová Alena, D. L. *Velky pruvodce protokoly TCP/IP a systemem DNS*. Computer Press, 2008.

[21] Koopman, P. Tutorial: Checksum and CRC Data Integrity Techniques for Aviation. May 2012, s. 44. Dostupné z: https://users.ece.cmu.edu/~koopman/pubs/KoopmanCRCWebinar9May2012.pdf.

[22] M. Daley, W. *DATA ENCRYPTION STANDARD (DES)* [online]. U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, 1999. Dostupné z: https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf.

[23] Menezes, A. J. et al. *Handbook of applied cryptography*. CRC press, 1996.

[24] Mospan, Y. *Secure data in Android - Encrypting Large Data* [online]. ProAndroidDev. Dostupné z: `https://proandroiddev.com/secure-data-in-android-encrypting-large-data-dda256a55b36`.

[25] Van Tilborg, H. C. – Jajodia, S. *Encyclopedia of cryptography and security.* Springer Science & Business Media, 2014.

A Uživatelská dokumentace

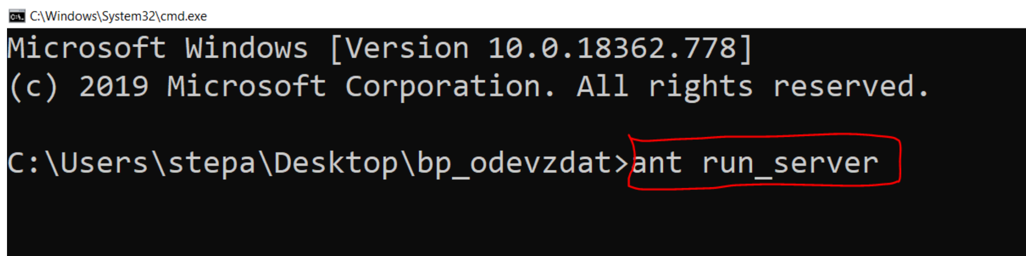
Soubor *readme.txt* v kořenové složce práce obsahuje popis všech adresářů práce. Práci je možné přeložit a spustit dvěma způsoby. Vzhledem ke konceptu programu, kdy je očekávána implementaci vlastních testů s využitím klientské části programu jako frameworku, tak doporučeným způsobem je program importovat do některého z vývojových prostředí (přiložený je projekt pro prostředí *IntelliJ IDEA*). Metoda *main* serveru se nachází v třídě *Server* v balíku *server.main*. Testy se nachází v balíku *test*. Program vyžaduje Javu verzi 8, která je součástí programu. Na verzi Javy 11 a výš program nefunguje.

Druhým způsobem je překlad pomocí nástroje *ant* (viz návod níže).

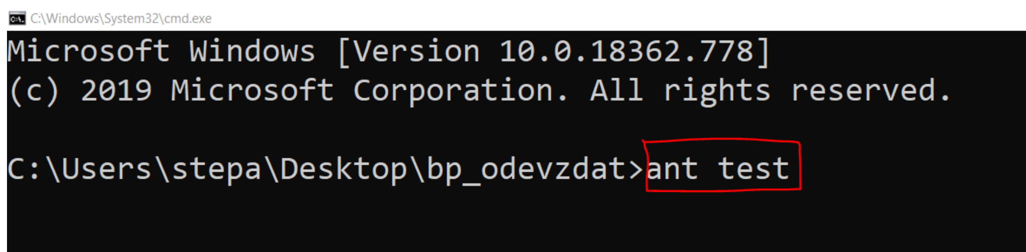
Spuštění nástrojem ant

1. Otevřeme dvě konzole v adresáři *ant source*.

2. V první konzoli spustíme server příkazem *ant run_server*.



3. V druhé konzoli spustíme testy příkazem *ant test*.



Dojde je spuštění implicitně vytvořených testů. Výstup by měl odpovídat obrázkům níže.

```
C:\Windows\System32\cmd.exe - ant run_server
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\stepa\Desktop\bp_odevzdat>ant run_server
Buildfile: C:\Users\stepa\Desktop\bp_odevzdat\build.xml

clean_server:
   [delete] Deleting directory C:\Users\stepa\Desktop\bp_odevzdat\build_server

compile_server:
    [mkdir] Created dir: C:\Users\stepa\Desktop\bp_odevzdat\build_server\classes
    [javac] C:\Users\stepa\Desktop\bp_odevzdat\build.xml:14: warning: 'includeantruntime' was not s
=last; set to false for repeatable builds
    [javac] Compiling 51 source files to C:\Users\stepa\Desktop\bp_odevzdat\build_server\classes

jar_server:
      [jar] Building jar: C:\Users\stepa\Desktop\bp_odevzdat\server.jar

run_server:
     [java] May 05, 2020 2:49:28 PM server.main.Server startServer.Main.Server is running
     [java] INFO: Server starts on port 6666.
     [java]
     [java] May 05, 2020 2:51:04 PM server.main.Server start
     [java] INFO: User connected on port 53957.
     [java] May 05, 2020 2:51:04 PM server.main.Server broadcastOnlineClients
     [java] INFO: Online users: (1/30).
     [java] May 05, 2020 2:51:04 PM server.main.Server logInfo
     [java] INFO: User RSATest successfully logged in.
```

```
    [junit] Test successful
    [junit] Test successful
    [junit] Test successful
    [junit] Encrypted data 1 bytes long same as server data.
    [junit] Decrypted data 1 bytes long same as original data.
    [junit] Encrypted data 2 bytes long same as server data.
    [junit] Decrypted data 2 bytes long same as original data.
    [junit] Encrypted data 4 bytes long same as server data.
    [junit] Decrypted data 4 bytes long same as original data.
    [junit] Encrypted data 8 bytes long same as server data.
    [junit] Decrypted data 8 bytes long same as original data.
    [junit] Encrypted data 16 bytes long same as server data.
    [junit] Decrypted data 16 bytes long same as original data.
    [junit] Encrypted data 32 bytes long same as server data.
    [junit] Decrypted data 32 bytes long same as original data.
    [junit] Encrypted data 64 bytes long same as server data.
    [junit] Decrypted data 64 bytes long same as original data.
    [junit] Encrypted data 128 bytes long same as server data.
    [junit] Decrypted data 128 bytes long same as original data.
    [junit] ------------- ---------------- ---------------

BUILD SUCCESSFUL
Total time: 16 seconds

C:\Users\stepa\Desktop\bp_odevzdat>
```

Návody

Součástí práce jsou také dva návody uvedené pro pochopení fungování programu. Návody je vhodné projít jako první, protože se jedná o nejjednodušší způsob, jak pochopit princip fungování programu.

Návod v souboru *tutorial1.txt* popisuje, jakým způsobem lze implementovat a otestovat vlastní symetrickou šifru na příkladu jednoduché Caesarovy šifry.

Druhým návodem sloužící obdobnému účelu pro asymetrické šifry je návod v souboru *tutorial2.txt*, který popisuje implementaci a testování Zavazadlového algoritmu.