STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 18: Informatika

Návrh a implementace internetového protokolu

Jan Štefanča
Hlavní město Praha

Praha 2025

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor č. 18: Informatika

Návrh a implementace internetového protokolu

Internet protocol design and implementation

Autoři: Jan Štefanča
Škola: Gymnázium, Praha 9, Litoměřická 726; Litoměřická 726/17, 190 00 Praha 9 – Prosek
Kraj: Hlavní město Praha
Konzultant: Bc. Jindřich Dvořák, Ing. Libor Polčák Ph.D.

Praha 2025

Prohlášení

Prohlašuji, že jsem svou práci SOČ vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

Prohlašuji, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné.

Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne 29. 1. 2025 Jan Štefanča

Poděkování

Děkuji za pomoc panu Ing. Liborovi Polčákovi Ph.D., který se mnou konzultoval moji implementaci protokolu a poskytl mi skvělý pramen informací, ze kterého jsem mohl čerpat. Dále děkuji panu Bc. Jindřichu Dvořákovi, který mi dal cenné rady k psaní této práce.

Anotace

Tato práce se zabývá návrhem a implementací vlastního komunikačního protokolu mezi klientem a serverem. Zaměřuje se na vysvětlení základních principů přenosu dat, struktury paketů a procesu handshake, který je klíčový pro navázání šifrovaného spojení. Hlavním cílem je vytvořit jednoduchý, modulární a bezpečný protokol, který je prakticky využitelný a snadno pochopitelný i pro čtenáře bez hlubokých teoretických znalostí. Text obsahuje příklady implementace v jazyce Java a ukazuje, jak různé vrstvy protokolu spolupracují při zajištění efektivní a bezpečné komunikace.

Klíčová slova

internet; internetový protokol; softwarový vývoj; protokol

Annotation

This thesis deals with the design and implementation of a custom communication protocol between client and server. The work focuses on explaining the basic principles of data transmission, packet structure and the handshake process, which is crucial for establishing an encrypted connection. The main goal is to create a simple, modular and secure protocol that is practical and easy to understand even for readers without deep theoretical knowledge. The text includes examples of Java implementations and shows how the different layers of the protocol work together to ensure efficient and secure communication.

Keywords

internet; internet protocol; software development; protocol

Obsah

Slovníček pojmů

- **implementace**: „*zavádění, zavedení něčeho, uvádění, uvedení něčeho do provozu*“ [1]
- **datagram**: „*označení pro základní jednotku, která je přepravována v počítačové síti s přepojováním paketů, kde není zajištěno jejich doručení, zachování pořadí ani eliminace duplicity*“ [2]
- **TCP segment**: dílek z celkového přenášeného objemu, který se skládá z dat a hlavičky. V hlavičce jsou kontrolní informace, které TCP využívá právě k zajištění správnosti přenosu [3].
- **datový proud**: sekvence dat, které můžou být potenciálně nekonečné. Dělíme ja na vstupní a výstupní. Výstupní proud slouží k zapisování a vstupní ke čtení. Pokud není vstupní proud uzavřený, není nemožné určit, kde je jeho konec [4].
- **hypertext**: „*počítačová aplikace využívající odkazů k vyhledávání a propojování různých informací v textu; takto zpracovaný text“ [1]*
- **třída**: „*představuje skupinu objektů, které nesou stejné vlastnosti*“ [5]
- **objekt/instance**: „*jeden konkrétní jedinec (reprezentant, entita) příslušné třídy*“ [5]
- **loopback**: „*logická smyčka elektrického signálu nebo datového toku z původního zařízení zpět ke zdroji bez dalšího zpracování nebo úprav.*“ [6]
- **thread pool**: „*udržuje více vláken, která čekají na přidělení úloh k souběžnému provádění dohlížejícím programem.*“ [7]

1 ÚVOD

Internet je nepostradatelnou součástí moderní společnosti a jeho význam neustále roste. Umožňuje nejen okamžitou komunikaci, online bankovnictví či práci na dálku, ale také podporuje digitalizaci mnoha služeb, které lidé využívají, aniž by si to vždy uvědomovali. Naprostá většina českých domácností je k internetu připojena a spoléhá na něj v každodenním životě. Dokonce i běžné úkony, jako je bezkontaktní platba kartou, vyžadují internetové připojení a fungování složitých síťových infrastruktur.

Přestože je internet tak zásadní, většina uživatelů nemá hlubší povědomí o tom, jak vlastně funguje. Lidé se na něj spoléhají, věří jeho spolehlivosti a bezpečnosti, ale málokdo si uvědomuje, jaké technologie umožňují jeho bezproblémový chod. Jedním z klíčových prvků, které stojí za veškerou komunikací na internetu, jsou síťové protokoly – soubory pravidel, díky nimž mohou zařízení mezi sebou efektivně a bezpečně přenášet data.

Tato práce má sloužit jako stručný teoretický úvod k přenosu dat po internetu a jako ukázka možnosti tvorby vlastního protokolu. První část práce se zaměřuje na vysvětlení základních principů komunikace po internetu a popisuje vybrané protokoly, s kterými se blíže běžní lidé dennodenně setkávají. Zároveň se věnuje šifrování, které je klíčové pro zabezpečení datového přenosu. Představí se základní principy šifrování a ochrany komunikace před neautorizovaným přístupem.

Praktická část se soustředí na návrh a implementaci vlastního protokolu v jazyce Java, ten bude implementovaný formou knihovny. Tento proces zahrnuje definici pravidel komunikace, návrh architektury protokolu a jeho následnou realizaci v programovacím jazyce. Čtenář tak získá nejen teoretické znalosti, ale také praktickou ukázku jejich využití při tvorbě vlastního protokolu.

Pro porozumění této práci se předpokládá, že čtenář má základní znalosti objektově orientovaného programování a je obeznámen s fundamentálními principy počítačových sítí. Tyto znalosti jsou nezbytné pro pochopení detailů návrhu protokolu a jeho implementace.

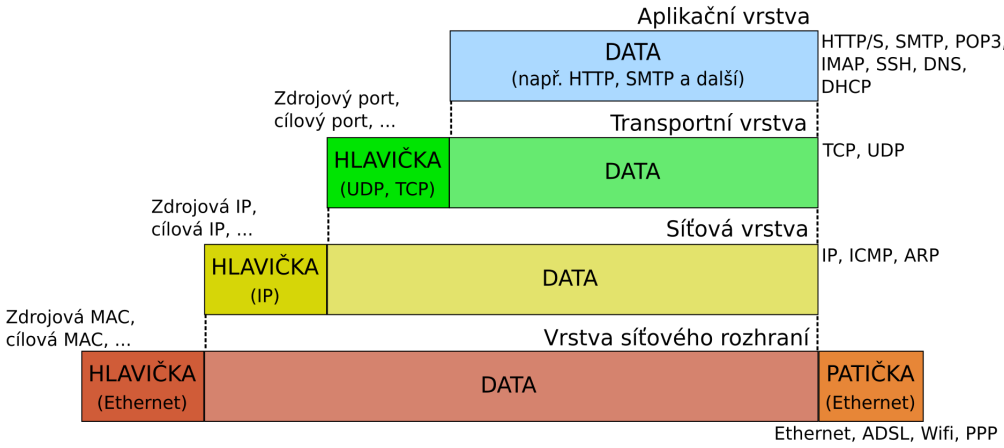
2 TEORETICKÝ ÚVOD K PŘENOSU DAT PO INTERNETU

Internet je *„celosvětová počítačová síť sloužící jako komunikační médium“* [1]. Přenos dat na něm lze nejjednodušeji představit jako tok binárních dat mezi různými místy, a to prostřednictvím kabelů a bezdrátových řešení (např. Wi-Fi). To však samo o sobě nestačí, proto existují protokoly. Jejich účelem je stanovit jasná pravidla, podle kterých se počítače řídí při přenosu, odesílání a přijímání dat [8, 9].

2.1 Internetové protokoly

Nejdůležitější rodinou protokolů na internetu jsou takzvané internetové protokoly, neboli TCP/IP rodina. Mezi funkce těchto protokolů patří přenos dat a vytvoření síťového spojení. Jedná se o skupinu pravidel, které určují syntaxi a význam jednotlivých zpráv na internetu [10].

Do této skupiny patří IP, TCP, UDP, HTTP, HTTPS, DNS a další. Protokoly mohou být nástavbami na jiné protokoly, například HTTP využívá TCP, který je nástavbou IP. V této ukázce lze protokoly rozdělit do vrstev a architektura TCP/IP právě s takovými vrstvami pracuje – aplikační, transportní, síťovou a síťového rozhraní (viz obrázek 1) [11]. Tato práce se zabývá převážně protokolem aplikačním.



Obr. 1: zapouzdření dat v sítích, TCP/IP [30]

2.1.1 IP

IP (*Internet Protocol*) je protokol, který používá adresní systém k určení místa, kam mají být data odesílána v počítačových sítích. Každé připojené zařízení má přiřazenou jedinečnou IP adresu, která funguje jako identifikátor v rámci této sítě. Při síťové komunikaci je IP adresa odesílatele a příjemce zahrnuta do každého datagramu. To umožňuje směrování těchto dat na správné místo. Systém adresování je klíčový pro efektivní a správné doručení informací v široce rozmanitých sítích, jako je internet [12, 13].

2.1.2 TCP

TCP (*Transmission Control Protocol*) je spolehlivá nástavba na IP, která umožňuje příjem dat ve stejném pořadí, v jakém byla odeslána. Data jsou posílána jako TCP segmenty, každý z nich je IP datagramem. Spolehlivost protokolu spočívá v detekci ztráty dat, chyb v přenosu a následných oprav znovuodesláním chybného segmentu. Je vhodný k odesílání jakéhokoli typu dat, kde je vyžadováno, aby nedošlo k jejich poškození (např. dokumenty či obrázky) [14]. Je na něm postaveno mnoho jiných protokolů, jakými jsou např. HTTP/1.0 a HTTP/2.0 [15, 16].

2.1.3 HTTP

HTTP (*Hypertext Transfer Protocol*), neboli hypertextový transferový protokol, je protokol na aplikační úrovni. Pro svou jednoduchost a rychlost je vhodný pro distribuci digitálních médií. Byl navržen a dodnes se využívá jako aplikační protokol *World Wide Webu* a jsou pomocí něj přenášeny webové stránky. Každá zpráva se skládá ze záhlaví a těla a rozlišuje se na žádost a odpověď [15, 17].

2.1.4 HTTPS

HTTPS (*Hypertext Transfer Protocol Secure*) je obdoba protokolu HTTP s přidanou bezpečnostní vrstvou. Využívá technologií TLS (*Transport Layer Security*), nebo SSL (*Secure Sockets Layer*), sloužící k navázání zabezpečeného připojení, při kterém se posílané zprávy šifrují. Jejich účelem je obrana před kyberútoky [18].

2.1.5 UDP

UDP (*User Datagram Protocol*) je protokol, který nezaručuje doručení datagramů, ani jejich pořadí. Je výhodný tehdy, kdy aplikace vyžaduje jednoduchost a minimální protokolový mechanismus [19].

Svojí jednoduchostí a rychlostí je v praxi vhodný pro využití v „reálném čase“, kdy je potřeba posílat data co nejrychleji a nevadí chyby v přenosu. Využívá se např. pro internetové volání, neboli VoIP (*Voice over Internet Protocol*), kde je vyžadováno nejmenší možné zpoždění zvuku [20].

2.2 Šifrovací algoritmy

Šifrování zabraňuje třetím osobám v přístupu k datům. Je hojně využíváno při komunikaci přes internet. Jedním z protokolů, který ho využívá je HTTPS. Provádí ho šifrovací algoritmy, ty slouží k šifrování a dešifrování dat pomocí klíčů. Dělí se na symetrické a asymetrické [21, 22].

2.2.1 Symetrické algoritmy

Symetrické algoritmy využívají jednoho klíče k zašifrování i dešifrování dat. Jsou výpočetně mnohem méně náročné než asymetrické algoritmy a disponují vysokou rychlostí. Největší slabinou těchto algoritmů je počáteční výměna klíčů. Pokud by byl klíč poslán po nešifrovaném spojení, mohl by ho útočník zachytit a veškerá komunikace by byla kompromitována [21]. Jedním z těchto algoritmů je AES (*Advanced Encryption Standard*), který využívá bloků po 128 bitech a dokáže zašifrovat velké objemy dat. Je vhodný například pro šifrování souborů [23].

2.2.2 Asymetrické algoritmy

Asymetrické algoritmy jsou založeny na tom, že místo jednoho klíče, pomocí kterého lze zprávu dešifrovat a zašifrovat, jsou klíče dva. K zašifrování slouží veřejný klíč a k dešifrování klíč soukromý [21]. Jedním z nejvíce používaných je algoritmus RSA, postavený na operacích s velkými náhodnými prvočísly. Ty umožňují tvorbu složitých matematických problémů, které jsou snadno proveditelné v jednom směru, ale obtížné v opačném, což zajišťuje bezpečnost šifrování. RSA algoritmus je pomalý a dovoluje šifrování dat pouze v relativně malém množství, z toho důvodu není vhodný pro zabezpečení dokumentů [24].

3 IMPLEMENTACE

Hlavním cílem této práce bylo vytvoření knihovny, umožňující přenos dat mezi dvěma zařízeními. Jelikož se jedná o relativně komplikovaný projekt, bylo by vhodné využití abstrakce a přehledného programovacího stylu. Proto jsem zvolil objektově orientovaný programovací jazyk Java. Oproti jiným jazykům (např. C++) má zabudované knihovny vhodné pro síťovou komunikace a jsem s ním seznámený do značného rozsahu.

Knihovna je rozdělena do vrstev (modulů), přičemž každá implementuje řešení konkrétní problematiky komunikace mezi dvěma zařízeními. Pomocí těchto vrstev je možné sestavit dva programy – jeden reprezentující žadatele (klienta) a druhý zprostředkovatele (server). Projekt je rozdělen na

- transportní vrstvu – určuje strukturu protokolu (*java.io*);
- bezpečnostní vrstvu – je určena k šifrování dat (*java.security* a *javax.crypto*);
- klientskou vrstvu – má úlohu komunikace se serverem (*java.net*) a
- serverovou vrstvu – komunikuje s klienty (*java.net*).

Transportní a bezpečnostní vrstvy představují implementační detaily protokolu. Klient a server umožňují komunikaci skrze protokol.

4 TRANSPORTNÍ VRSTVA

Transportní vrstva je jednou z nejdůležitějších částí tohoto protokolu, kterou využívá jak klient tak server, a to ve stejné podobě. Představuje ji třída `TransportLayer` a třídy jednotlivých druhů paketů. Její funkcí je přenos skrz datové proudy a zajištění, že data budou odeslána a přijata v požadované formě.

4.1 Proč je důležité určit délku balíčku?

Při čtení dat ze vstupního proudu není možné rozlišit jednotlivé posílané zprávy. To by mohlo mít zdánlivě jednoduché řešení – ohraničit posílané zprávy (viz příklad níže).

START prosím dvě plata vajec **STOP**

Tento přístup má však mnoho nevýhod. Například využitím rezervovaného slova „stop“ v posílaném textu, dojde k nepřečtení zbytku zprávy. Přeškrtnutá data v následujícím příkladu vyznačují data, která by byla opomenuta.

START Jiří si dá k snídani **STOP**~~rocentně 10 vajec STOP~~

Tato zpráva by byla přečtena pouze jako „Jiří si dá k snídani.“ Zvyšuje se tedy chybovost přenosu. Zároveň dochází i k jeho zpomalení, a to hlavně při vyšších objemech dat. Hledání ohraničení je potom poměrně náročné a doba zpracování se lineárně zvyšuje.

Tento problém lze vyřešit tím, že každá zpráva bude začínat informací o délce posílaných dat.

47 musím jít znovu koupit vejce, pojedu autostopem

47 představuje počet odesílaných znaků. Tento přístup zajišťuje, že je přečtený správný počet dat a nedojde tak k žádné ztrátě. Metoda je efektivní i pro přenos vysokého objemu dat.

V datovém proudu jsou informace interpretovány jako bajty. Jeden bajt vyjadřuje 8 bitů, těmi lze vyjádřit všechna čísla od 0-255[1]. Pro získání určitého počtu bajtů z proudu v Javě se využívá datový typ integer – 32 bitové číslo, 32 bitů se rovná 4 bajtům. Je možné z proudu přečíst tedy nejdříve 4 bajty a poté už zbytek zprávy.

```
byte[] header = readN(4); // přečte určitý počet bajtů (4)
int len = ByteUtil.byteArray4ToInt(header); // převede bajty na integer
byte[] data = readN(len);
Packet packet = new Packet(len, data); // vytvoření instance třídy Packet
```

Následně se vytvoří instance třídy Packet, která vyjadřuje posílaný balíček.

1 V Javě je však datový typ `byte` (bajt) zavedený jako číslo od -128 do 127. Tato implementace záporných čísel nevyužívá, jelikož délka nemůže být nikdy záporná. Nástroj na převod bajtů na číslo tak pracuje s bajtem, jako by byl vždy kladný.

4.2 Paket

Paket je balíček posílaných informací mezi klientem a serverem. Můj protokol rozlišuje 3 typy paketů:

- paket žádosti, který posílá klient serveru;
- paket odpovědi, který posílá server zpět klientovi a
- paket handshaku, který figuruje v procesu navázání zabezpečeného spojení (viz kapitola 5.1). Paket handshaku je specifický tím, že neposílá žádnou hlavičku, protože slouží k výměně dat o fixní délce.

Pakety žádosti a odpovědi se skládají ze dvou hlavních částí – hlavičky, která nese informace o délce těla, a těla, ve kterém se vyskytují posílaná data.

Nejvyšší možná kapacita těla paketu je omezena na maximální kladnou hodnotu datového typu `Integer` (2 147 483 647 bajtů, tedy přibližně 2 GB).

Paket je ve své základní podobě reprezentován jako pole bajtů, což umožňuje přímé uložení a manipulaci s daty. Tento přístup je ideální pro menší objemy dat, které lze snadno načíst a zpracovat přímo v operační paměti. Nicméně, pokud je potřeba přenášet nebo zpracovávat větší objemy dat (například obsah velkých souborů), je efektivnější využít datové proudy, které výrazně rozšiřují možnosti práce s daty. Výhodou spojení datových proudů je to, že není nutné celá data předem uložit do operační paměti. To šetří operační výkon a zvyšuje efektivitu aplikace.

Pro zajištění bezpečnosti komunikace je nutné posílaná data před odesláním šifrovat a po přijetí dešifrovat. Transportní vrstva proto využívá middleware – modul, který umožňuje provádět další zpracování dat. Po navázání zabezpečeného spojení (handshake) se tímto middlewarem stává bezpečnostní vrstva, která zajišťuje šifrování a dešifrování dat.

4.3 Struktura zpráv

Pokud je vhodné přidat k paketu ještě nějaké informace, je možné odesílanou zprávu strukturovat. Toho využívá např. HTTP. Ten rozlišuje žádost a odpověď. Ty s sebou nesou mimo těla s obsahem ještě další informace. Mezi ně patří verze protokolu, využitá metoda žádosti, žádaný dokument, status odpovědi, hlavičky a další [15].

```
GET /dokument HTTP/1.0
Název-hlavičky: hodnota-hlavičky

posílaná data
```

Na příkladě se nachází jednoduchá HTTP žádost typu `GET`, která se snaží získat dokument `/dokument` a uvádí, že využívá verze protokolu `HTTP/1.0`. Implementace této práce je protokolem HTTP v tomto směru inspirovaná. Každá paketa tak ve svém těle přenáší krom posílaných informací i další data.

4.3.1 Struktura žádosti

Každá žádost začíná verzí protokolu a cestou k dokumentu, který se snaží získat. Tyto dva údaje jsou oddělené mezerou.

```
JIP/1.0 /cesta/k/dokumentu
```

Poslání verze je důležité v případě, že novější verze protokolu strukturu zprávy pozmění. Protokol tak může poznat starší verzi a mít pro ni vybudovanou zpětnou podporu. Klient a server s rozdílnými verzemi tak nemusí být nutně problém. K umožnění posílání různých dokumentů jedním serverem, slouží cesta k dokumentu.

Každá žádost je pak schopna nést hlavičky a tělo, podobně jako u HTTP.

```
JIP/1.0 /cesta/k/dokumentu
Název-hlavičky: hodnota-hlavičky
Název-hlavičky2: hodnota-hlavičky

posílaná data
```

Tělo a hlavičky jsou však naprosto dobrovolné a je na serveru, jestli je bude přijímat. Nejdůležitější částí žádosti je tedy první řádek.

4.4 Struktura odpovědi

Odpověď je koncipována podobně jako žádost. Místo cesty k dokumentu, ale uvádí status.

```java
public enum StatusCode {
    OK(1),
    NOT_FOUND(2),
    DENIED(3),
    ERROR(4);

    private final int code;
    StatusCode(int code) {
        this.code = code;
    }

    ...
}
```

Kódy statusu jsou OK – vše proběhlo vpořádku, NOT FOUND – dokument nebyl nalezen, DENIED – zamítnuto a ERROR – chyba serveru. Využívání těchto kódů je dobrovolné a záleží na správci serveru, aby je implementoval.

Odpověď kód využije v číselné podobě. Hlavičky a posílaná data jsou opět dobrovolná. Celá odpověď pak vypadá takto:

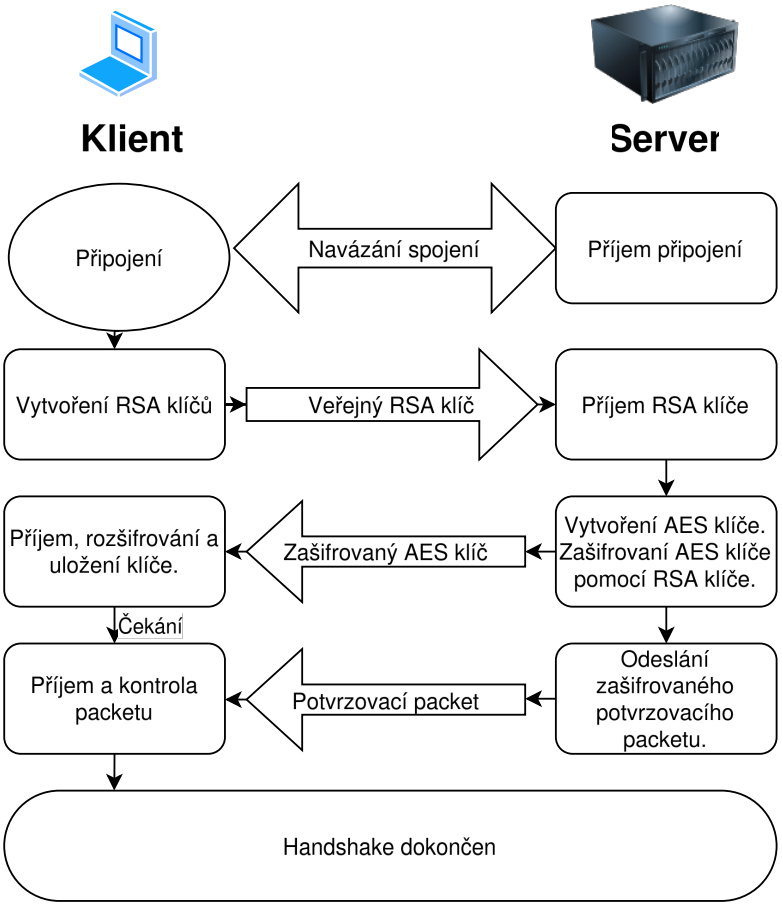
```
JIP/1.0 1
Název-hlavičky: hodnota-hlavičky
Název-hlavičky2: hodnota-hlavičky

posílaná data
```

5 BEZPEČNOSTNÍ VRSTVA

Bezpečnostní vrstva je klíčovou součástí systému zajišťující šifrování a dešifrování dat při přenosu mezi klientem a serverem. Zaručuje, že přenášená data jsou chráněna před neoprávněným přístupem a případnou manipulací třetími stranami. Implementace bezpečnostní vrstvy je vyjádřena třídou `SecurityLayer`, která funguje jako middleware transportní vrstvy a má dvě specializované podoby: `ServerSecurityLayer` pro server a `ClientSecurityLayer` pro klienta.

5.1 Handshake

Handshake je proces, při kterém dvě strany navazují zabezpečené spojení. Klient po připojení k serveru vytvoří pár asymetrických RSA klíčů a veřejný klíč zašle serveru. Server přijme veřejný klíč klienta a zašifruje pomocí něj symetrický AES klíč, který vytvoří. Klient obdrží tento klíč, dešifruje jej a využije pro další šifrovanou komunikaci se serverem. Server následně odešle potvrzovací paket, který klient dešifruje a zkontroluje. Pokud vše proběhlo v pořádku, je šifrované spojení navázáno a obě strany mohou bezpečně posílat šifrované zprávy.



Obr. 2: diagram procesu handshake

5.1.1 Problém tohoto přístupu

I když je popsaný proces výměny klíčů na první pohled jednoduchý a efektivní, obsahuje vážnou bezpečnostní slabinu. Pokud by se během fáze navazování spojení mezi klientem a serverem postavil útočník, mohl by zachytit komunikaci a vydávat se za server. Tomuto typu útoku se říká MITM (*man-in-the-middle*). To by znamenalo, že klient pošle svůj veřejný klíč nikoliv skutečnému serveru, ale útočníkovi. Ten by poté vytvořil vlastní pár klíčů a předal veřejný klíč serveru jako svou identitu. Tímto způsobem by útočník mohl dešifrovat všechny přenášené zprávy, pozměnit je a opět je zašifrovat, aniž by si to kterákoli ze stran uvědomila [25].

V praxi se tomuto problému předchází pomocí certifikačních autorit (CA). To jsou důvěryhodné organizace, které vydávají digitální certifikáty. Ty slouží právě k potvrzení, že server, se kterým klient navazuje spojení, je skutečně tím, za koho se vydává [26].

Pro účely této práce není implementace certifikačních autorit zahrnuta, protože je technicky velmi složitá a přesahuje rozsah tohoto projektu. Certifikační autority jsou ale na internetu nezbytné pro zajištění skutečné bezpečnosti proti útokům typu MITM.

6 KLIENT

Klient využívá třídy `Socket`, která reprezentuje klientský TCP soket. To je *„koncový bod pro komunikaci mezi dvěma počítači“* [27]. Pro datový přenos je využíván protokol TCP. Objekt soketu po úspěšném připojení zprostředkovává vstupní a výstupní datové proudy. Díky nim jsou následně vytvořeny instance transportní vrstvy a bezpečnostní vrstvy.

```
public void connect() throws Exception {
    if (!isClosed()) {
        throw new IllegalStateException("Klient je připojen k serveru.
Použijte #connect(InetSocketAddress) k přepojení na jinou adresu, nebo
#close k uzavření spojení.");
    }
    socket = new Socket();
    socket.setSoTimeout(60000); // uzavřít po minutě blokování čtení
    if (socketConfigurator != null)
        socketConfigurator.accept(socket);
    socket.connect(address);
    transportLayer = new TransportLayer(socket.getInputStream(),
socket.getOutputStream());
    securityLayer = new ClientSecurityLayer();

    handshake();

    Packet serverReady = transportLayer.readPacket();
    if (serverReady == null || serverReady.getLength() != 1 ||
serverReady.getData().at(0)[0] != 1) {
        close();
        throw new SecureTransportException("Nepodařilo se navázat
zabezpečného spojení. Server vrátil špatnou odpověď.");
    }
}
```

Po vytvoření těchto vrstev dojde k handshaku. Po výměně klíčů je aktivován middleware bezpečnostní vrstvy – komunikace tak podléhá od toho bodu symetrickému šifrování. Následně je zkontrolováno šifrování pomocí ověřovacího paketu, u kterého jsou předem známá očekávaná data. Pokud se shodují vše proběhlo v pořádku a handshake je dokončen.

```
private void handshake() throws SecureTransportException {
    try {
        var handshake = securityLayer.createHandshakePacket();
        transportLayer.writePacket(handshake);
        var serverHandshake = transportLayer.readN(256);
        securityLayer.acceptServerHandshake(serverHandshake);
        transportLayer.useMiddleware(securityLayer);
    } catch (Exception e) {
        LOGGER.log(System.Logger.Level.ERROR, "Nepodařilo se dokončit
proces handshake, uzavírám soket...", e);
        try {
            close();
        } catch (IOException e1) {
            LOGGER.log(System.Logger.Level.ERROR, "Nepodařilo se uzavřít
transportní vrstvu: " + e1.getMessage(), e1);
            throw new RuntimeException(e1);
        }
        throw new SecureTransportException("Nepodařilo se dokončit proces
handshake", e);
    }
}
```

Po připojení jsou k dispozici funkce pro posílání žádostí a zpřístupněna možnost ukončení spojení. Všechny operace klienta jsou synchronní. Do vyřešení požadavku je tak zablokováno hlavní vlákno. Pro uzavření spojení slouží metoda `close`.

7 SERVER

Úkolem serveru je odpovídat na žádosti klienta. Jeho funkcionalitu zajišťuje třída `ServerSocket`, ta implementuje soket na straně serveru. Po nastartování je vytvořeno asynchronní vlákno, na kterém jsou přijímána připojení od klientů. Po připojení klienta je, v případě toho, že je nějaké volné, z thread poolu vyčleněno volné vlákno, na kterém se řeší veškeré komunikace týkající se přímo jednoho klienta. Po odpojení klienta je vlákno uzavřeno.

```
public void start() throws IOException {
    // Create threadPool
    threadPool = Executors.newFixedThreadPool(threadPoolSize);
    serverSocket = new ServerSocket();
    serverSocket.bind(address);
    acceptThread = new Thread(() -> {
        try {
            while (running) {
                accept();
            }
        } catch (Exception e) {
            if (!running) {
                return;
            }
            System.Logger logger =
System.getLogger(JIPServerImpl.class.getName());
            logger.log(System.Logger.Level.WARNING, "Nepovedlo se přijmout
klienta", e);
        }umožňuje výměnu protokolů jedné vrstvy bez dopadu na ostatní.
Příkladem mů
    });
    running = true;
    acceptThread.start();
}
```

```
private void accept() {
    try {
        var socket = serverSocket.accept();
        socket.setSoTimeout(30000); // po 30 sekundách bude klient odpojen
        threadPool.submit(() -> {
            try {
                new ClientHandler(socket, router).handle();
            } catch (IOException e) {
                if (!running) {
                    return;
                }
                LOGGER.log(System.Logger.Level.WARNING, "Nepodařilo se
vytvořit ClientHandler", e);
            }
        });
    } catch (IOException e) {
        if (!running) {
            return;
        }
        LOGGER.log(System.Logger.Level.WARNING, "Nepodařilo se přijmout
připojení klienta", e);
    }
}
```

Každé připojení klienta je na serveru reprezentováno třídou `ClientHandler`. Ten obsahuje implementaci handshaku na straně serveru a má za úkol odpovídat na žádosti, ve kterých se nachází dotazovaná cesta k dokumentu na serveru. Cesty řeší třída `Router`, která má za úkol zjistit, zda bylo k dotazované cestě přiřazeno konkrétní řešení třídy `RequestHandler`. A následně ho využít k vytvoření odpovědi posílané klientovi.

Cesty reprezentuje abstraktní třída `Route`, která má dvě děti `StaticRoute` a `DynamicRoute`. `StaticRoute` je statická cesta, která neobsahuje parametry. `DynamicRoute` je cesta dynamická, což znamená, že obsahuje parametry, které jsou v textové podobě vyjádřeny v hranatých závorkách.

```
/statická/cesta
/cesta
/soubor.txt

/obrázky/[název]
/profil/[uživatel]
/článek/[název]
```

Každá cesta musí být registrována pomocí metody `route` třídy `Router`, její instanci lze získat pomocí funkce `getRouter` instance serveru. Pokud není vyžádaná cesta nalezena, je automaticky klientovy poslána odpověď s kódem odpovědi `NOT_FOUND` (nenalezeno).

8 Využití

Implementace zpřístupňuje dvě hlavní třídy, které reprezentují klienta a server. Jedná se o `Client` a `JIPServer`.

8.1 Server

Pro vytvoření serveru slouží statické funkce `create` třídy `JIPServer`. Je možné buď jen uvést adresu, na kterou se server naváže, nebo ještě specifikovat maximální počet využitých vláken najednou. Jedno vlákno je jeden připojený klient. Základní nastavení je 10 vláken.

```
var server = JIPServer.create(new InetSocketAddress(8080)); // Vytvoří server na lokálním portu 8080

var server = JIPServer.create(new InetSocketAddress(8080), 1); // Vytvoří server na lokálním portu 8080 a povolí komunikaci jen s jedním klientem najednou
```

Dalším krokem po vytvoření serveru je přidání cest. Slouží k tomu instance třídy `Router`, kterou lze získat funkcí `getRouter`.

```
var router = server.getRouter();
```

Nejjednodušším způsobem vytvoření cesty je využití metody `route`. Jejími parametry jsou cesta a `RequestHandler`. Ten se dá vyjádřit pomocí lambda funkce. V tomto příkladě je vytvořena statická cesta, která posílá klientovi „Ahoj!“ a dynamická cesta, která vyžaduje parametr jméno a pomocí něj vytvoří osobní pozdrav ve formátu „Ahoj, [jméno]!“

```
// Bežná praxe při programování je využívat anglická slova, avšak pro účely tohoto příkladu jsou použita česká.
router.route("/ahoj", (žádost, odpověď, _) → {
    odpověď.sendString("Ahoj!");
});

router.route("/ahoj/[jmeno]", (žádost, odpověď, parametry) → {
 odpověď.sendString("Ahoj, %s!".formatted(parametry.get("jmeno")));
});
```

Objekty žádost a odpověď umožňují manipulovat s odesílanými daty a získávat přijímaná. Klíčová funkce objektu odpovědi je odeslání souboru. K tomu slouží jeho metoda `sendFile`.

```
router.route("/cenová_analýza_vajec", (žádost, odpověď, _) → {
 try {
 odpověď.sendFile(new File("ceny_vajec.xlsx"));
    } catch (FileNotFoundException e) {
        // Pokud nebude soubor nalezen, bude spuštěn tento blok kódu
        odpověď.sendString("Soubor nebyl na serveru nalezen");
        odpověď.setStatus(StatusCode.NOT_FOUND);
    }
});
```

Cesty lze přidávat i po nastartování serveru.

Server se spouští metodou `start` a vypíná metodou `stop`. Po nastartování běží nezávisle na hlavním vlákně.

```
server.start();
server.stop();
```

8.2 Klient

Pro vytvoření klienta slouží statická funkce `create` třídy `Client`. Ta přijímá adresu serveru v podobě instance třídy `InetSocketAdress`.

```
var client = Client.create(new InetSocketAddress("example.com", 8080));
```

Pro připojení klienta k serveru slouží metoda `connect`. Její variace, která přijímá novou adresu serveru, je užitečná primárně v případě přepojování na jiný server.

```
client.connect();
```

```
client.connect(new InetSocketAddress("127.0.0.1",8080));
```

Pro zaslání žádosti slouží funkce `fetch`, která přijímá buď datový `connect` typ `String`, tedy textové vyjádření cesty k dokumentu, nebo objekt `Request`, který reprezentuje žádost. Funkce vrací instanci `ResponsePacket`.

```
var res = client.fetch("/cenová_analýza_vajec");
if (res.getStatusCode() == StatusCode.NOT_FOUND) {
    LOGGER.log(System.Logger.Level.WARNING, "Cenová analýza vajec nebyla na
serveru nalezena");
} else {
    // Uložení do souboru
    try (var fos = new FileOutputStream("ceny_vajec.xlsx")) {
        fos.write(res.getBody());
        LOGGER.log(System.Logger.Level.INFO, "Cenová analýza vajec byla
uložena do souboru ceny_vajec.xlsx");
    }
}

// Komplikovanější žádost s hlavičkami a tělem
Request request = new Request(JIPVersion.JIP1_0, "/kup/vejce");
request.setHeader("Jméno", "Jiří");
request.setHeader("Ověření", token);
request.setBody("počet=%d; forma=plato".formatted(počet));
var res = client.fetch(request);
// Nějaké zpracování odpovědi...
```

Pro odpojení ze serveru slouží metoda `close`. Ta ukončí spojení se serverem a uzavře datové proudy.

```
client.close();
```

Po uzavření je stále možné obnovit připojení či se připojit na jiný server pomocí stejného klienta, opět pomocí metody `connect`.

9 TESTOVÁNÍ

„Testování je jedním z nejdůležitějších kroků vývoje softwaru.“ [24] Průběžně jsem tak optimalizoval kód a hledal možnosti zlepšení. Tato kapitola se zabývá testováním přenosu. Tyto testy jsem prováděl na loopbacku. Server a klient tak spolu komunikovali na mém zařízení, z rozdílných procesů. Specifika mého zařízení se nachází v příloze 1. Prvním krokem po připojení klienta k serveru je handshake.

9.1 Test rychlosti připojení a handshaku

Nejdříve je třeba vytvořit a odstartovat server.

```
var server = JIPServer.create(new InetSocketAddress(8080));
server.start();
```

Následně se pomocí `for` smyčky třicetkrát otestuje doba připojení.

```
for (int i = 0; i < 30; i++) {
    var client = Client.create(new InetSocketAddress(8080));
    var current = System.nanoTime();
    client.connect();
    var elapsed = System.nanoTime() - current;
    LOGGER.log(System.Logger.Level.INFO, "Čas připojení {0} ms", elapsed /
1_000_000);
    client.disconnect();
}
```

Měření	Čas [ms]
1	257
2	189
3	121
4	118
5	38
6	78
7	156
8	107
9	51
10	70
11	57
12	63
13	87
14	44
15	83
16	66
17	48
18	71
19	53
20	65
21	100
22	66
23	22
24	58
25	61
26	88
27	44
28	60
29	80
30	34
Průměr	81
Průměrná odchylka[2]	33

Tab. 1: měření doby připojení klienta k serveru

2 Průměrná odchylka je vypočítána jako $\bar{d} = \frac{1}{n}\sum_{i=1}^{n}\left|x_i - \bar{x}\right|$ [29].

Průměrná doba připojení a handshaku dosáhla 81 ms s průměrnou odchylkou 33 ms. Domnívám se, že odchylky od průměru byly dány faktory, které není možné ovlivnit. Mezi ně může patřit variabilní vytížení stroje. Jelikož navázání spojení předchází každé komunikaci mezi klientem a serverem, je jeho doba klíčová pro celkovou výkonnost protokolu.

9.2 Třída RequestTimer

Pro další testování je využita třída `RequestTimer`, která byla vytvořena jako pomocník stopování času. Měří jednotlivou dobu procesu odeslání žádosti a příjmu odpovědi. Níže naleznete její zdrojový kód.

```
public class RequestTimer {
    private static final System.Logger LOGGER =
System.getLogger("RequestTimer");
    public static ResponsePacket watch(Task task, String title) {
        long current = System.nanoTime(); // aktuální čas
        ResponsePacket r = null;
        try {
            r = task.run(); // poslání žádosti
        } catch (Exception e) {
            LOGGER.log(System.Logger.Level.INFO, "[ŽÁDOST SELHALA] " +
title, e);
            return null;
        }
        long elapsed = System.nanoTime() - current; // čas, který uběhl
        LOGGER.log(System.Logger.Level.INFO, "[MĚŘENÍ ÚSPĚŠNÉ] {0} $$ {1}
ms", title, elapsed/1_000_000);
        return r;
    }

    public interface Task {
        ResponsePacket run() throws Exception;
    }
}
```

9.3 Test malého objemu dat

V tomto testu se posílá malý objem dat ze statické cesty na serveru. V kódu níže se tato cesta vytvoří `/ahoj`.

```
// Přidá cestu /ahoj, ta je statická, nevyužívá tedy parametry (_)
server.getRouter().route("/ahoj", (request, response, _) -> {
    response.sendString("Ahoj!");
});
```

Následně je pomocí třídy RequestTimer počítán čas. Bylo provedeno 30 měření.

Měření	Čas [ms]
1	45
2	83
3	83
4	82
5	81
6	81
7	81
8	81
9	82
10	81
11	83
12	82
13	81
14	82
15	82
16	82
17	81
18	82
19	82
20	82
21	83
22	82
23	82
24	83
25	83
26	82
27	82
28	83
29	82
30	82
Průměr	81
Průměrná odchylka	2

Tab. 2: test malého objemu dat

Výsledky tohoto testu ukazují poměrně stabilní dobu odpovědi, kterou naznačuje nízká průměrná odchylka.

9.4 Test teoretického maximálního objemu dat

Maximální objem dat jednoho paketu je zhruba 2 GB (viz kapitola 4.2). Tento test zjišťuje dobu přenosu tohoto objemu. Nejdříve je potřeba vytvořit sobour o velikosti $2 \cdot 10^9$ bytů, tedy 2 GB.

```
byte[] bytes = new byte[(int) (2 * Math.pow(10, 9))];
Arrays.fill(bytes, (byte) 0xFF); // Naplní pole bajtů hodnotou 0xFF

// Napíše bajty do souboru large.txt
try (OutputStream out = new FileOutputStream("large.txt")) {
    out.write(bytes);
}
```

Server poté tento soubor připojí k odpovědi pomocí datového proudu.

```
// Přidá cestu /max, ta je statická, nevyžaduje tedy parametry (_)
server.getRouter().route("/max", (request, response, _) → {
    try { // Pošle soubor large.txt
        response.sendFile(new File("large.txt"));
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
});
```

Klient zašle žádost třicetkrát.

```
var a = RequestTimer.watch(() → client.fetch("/max"), "max");
a = null;
System.gc(); // Pokusí se vytvořit ihned místo v operační paměti, jelikož
jsou odpovědi ukládány jako proměnné.
```

Měření	Čas [ms]
1	11211
2	11727
3	13181
4	12381
5	13132
6	12791
7	12727
8	12606
9	12398
10	12829
11	12474
12	12558
13	12458
14	12064
15	12723
16	12375
17	12077
18	12242
19	12159
20	11613
21	12109
22	11149
23	12142
24	12062
25	11957
26	11991
27	11566
28	11631
29	11716
30	11710
Průměr	12 192

Tab. 3: test přenosu 2 GB

Přenos 2 GB probíhal průměrně 12 192 milisekund, což odpovídá přenosové rychlosti 1 312 megabitů za sekundu. Test potvrdil, že je možné přenést soubor o přibližné maximální velikosti paketu.

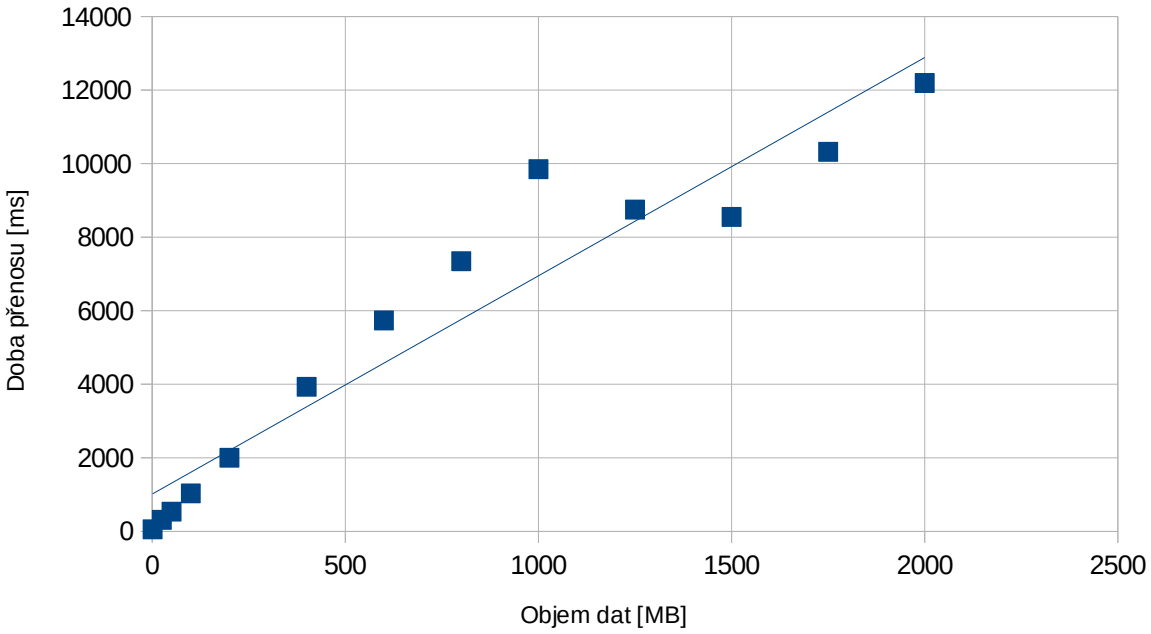
9.5 Test doby přenosu v závislosti na objemu dat

V tomto testu je provedeno 30 měření na každý zadaný datový objem. Z těchto měření je vypočítán průměr. Byly zvoleny hodnoty od 1 do 2000 MB. Data byla jako v předchozím testu načítána ze souboru.

Data [MB]	Čas [ms]	Rychlost [Mbit/s]
1	55	145
25	313	639
50	536	746
100	1033	774
200	2000	800
400	3 930	814
600	5736	837
800	7346	871
1000	9 848	812
1250	8750	1 143
1500	8551	1 403
1750	10320	1 357
2000	12192	1 312
	Průměr	897

Tab. 4: závislosti doby přenosu na objemu dat

Průměrná rychlost přenosu ze všech 390 měření je 897 megabitů za sekundu. To, že se tato rychlost zvyšuje v závislosti na počtu přenášených bajtů, je nejspíše způsobeno neovlivnitelnými faktory. Za účelem vizualizace možného statického trendu je vytvořen graf (viz obrázek 3).



Obr. 3: graf závislosti doby přenosu na objemu dat

Na grafu lze vypozorovat statistický trend. Měření mezi 1-1000 MB naznačuje lineární závislost času na objemu dat, ale po hodnotě 1000 MB nastává odklon.

10 ZÁVĚR

Cílem této práce bylo poskytnout stručný teoretický úvod do problematiky internetových protokolů, a následně navrhnout a vytvořit vlastní. Základní poznatky o internetu, jeho protokolech a šifrování představila v teoretické části. Praktická část přiblížila proces tvorby vlastního protokolu, který byl proložen ukázkami kódu. Bylo představeno jakým způsobem lze využívat vzniklou knihovnu, jejíž funkcionalita byla ověřena na závěr v kapitole testování.

Všechny stanovené cíle byly splněny – protokol byl úspěšně navržen a implementován. Výsledky ukázaly, že navržený systém správně funguje a umožňuje bezpečný přenos dat. Tato práce tak mohla nejenom rozšířit teoretické znalosti čtenáře, ale také poskytla praktickou ukázku, jak lze protokol navrhnout a implementovat.

Zdrojový kód licencovaný pod licencí MIT a jeho dokumentace jsou dostupné na adrese webové adrese https://github.com/PanJohnny/JIPProtocol.

11 ZDROJE

[1] PRAHA: ÚSTAV PRO JAZYK ČESKÝ AV ČR, V. V. I. *Internetová jazyková příručka* [online]. 2024 2008 [vid. 2024-12-27]. Dostupné z: https://prirucka.ujc.cas.cz/

[2] *Datagram* [online]. 2021 [vid. 2025-01-25]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Datagram&oldid=20326918

[3] Segmentation Explained with TCP and UDP Header. *ComputerNetworkingNotes* [online]. [vid. 2025-01-25]. Dostupné z: https://www.computernetworkingnotes.com/ccna-study-guide/segmentation-explained-with-tcp-and-udp-header.html

[4] *Datový proud* [online]. 2021 [vid. 2025-01-25]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Datov%C3%BD_proud&oldid=20295401

[5] *Základní pojmy OOP. Třída, objekt, jeho vlastnosti. Metody, proměnné. Konstruktory.* [online]. [vid. 2025-01-26]. Dostupné z: https://is.muni.cz/el/1433/podzim2006/PB162/um/02/printable.html#foilgroup1

[6] *Loopback* [online]. 2022 [vid. 2025-01-26]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Loopback&oldid=21808128

[7] *Thread pool* [online]. 2025 [vid. 2025-01-28]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Thread_pool&oldid=1272004948

[8] INFORMATIKA. *Záznam, přenos a distribuce dat a informací v digitální podobě – INFORMATIKA* [online]. [vid. 2024-12-28]. Dostupné z: https://www.naucmese.eu/informatika/zaznam-prenos-a-distribuce-dat-a-informaci-v-digitalni-podobe/

[9] S.R.O, Eri Internet. Rozumíme internetu: hloubkový pohled na technologické aspekty. *eri-internet.cz* [online]. [vid. 2024-12-28]. Dostupné z: https://eri-internet.cz/blog/jak-funguje-internet-rozbor-technologickych-aspektu

[10] *Nejčastěji používané síťové protokoly* [online]. [vid. 2024-12-28]. Dostupné z: https://www.zonercloud.cz/magazin/nejcasteji-pouzivane-sitove-protokoly

[11] *TCP/IP* [online]. 2024 [vid. 2024-12-29]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=TCP/IP&oldid=24469452#Extern%C3%AD_odkazy

[12] What is an IP Address? *GeeksforGeeks* [online]. 00:00:58+00:00 [vid. 2024-12-28]. Dostupné z: https://www.geeksforgeeks.org/what-is-an-ip-address/

[13] *Internet Protocol* [online]. Request for Comments. RFC 791. B.m.: Internet Engineering Task Force. 1981 [vid. 2024-12-29]. Dostupné z: doi:10.17487/RFC0791

[14] EDDY, Wesley. *Transmission Control Protocol (TCP)* [online]. Request for Comments. RFC 9293. B.m.: Internet Engineering Task Force. 2022 [vid. 2024-12-29]. Dostupné z: doi:10.17487/RFC9293

[15] NIELSEN, Henrik, Roy T. FIELDING a Tim BERNERS-LEE. *Hypertext Transfer Protocol – HTTP/1.0* [online]. Request for Comments. RFC 1945. B.m.: Internet Engineering Task Force. 1996 [vid. 2024-12-29]. Dostupné z: doi:10.17487/RFC1945

[16] BELSHE, Mike, Roberto PEON a Martin THOMSON. *Hypertext Transfer Protocol Version 2 (HTTP/2)* [online]. Request for Comments. RFC 7540. B.m.: Internet Engineering Task Force. 2015 [vid. 2024-12-29]. Dostupné z: doi:10.17487/RFC7540

[17] *World Wide Web* [online]. 2024 [vid. 2024-12-29]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=World_Wide_Web&oldid=24431967

[18] RESCORLA, Eric. *HTTP Over TLS* [online]. Request for Comments. RFC 2818. B.m.: Internet Engineering Task Force. 2000 [vid. 2024-12-29]. Dostupné z: doi:10.17487/RFC2818

[19] *User Datagram Protocol* [online]. Request for Comments. RFC 768. B.m.: Internet Engineering Task Force. 1980 [vid. 2024-12-29]. Dostupné z: doi:10.17487/RFC0768

[20] TCP or UDP – Which protocol does VoIP use? | VIP VoIP. *https://vipvoip.co.uk/* [online]. [vid. 2024-12-29]. Dostupné z: https://vipvoip.co.uk/tcp-vs-udp/

[21] *Algoritmy kryptologie – Wikisofia* [online]. [vid. 2024-12-29]. Dostupné z: https://wikisofia.cz/wiki/Algoritmy_kryptologie

[22] K čemu je šifrování a šifrovaná komunikace na internetu? *Dvojklik* [online]. 27. září 2021 [vid. 2024-12-29]. Dostupné z: https://www.dvojklik.cz/k-cemu-je-sifrovani-a-sifrovana-komunikace-na-internetu/

[23] Advanced Encryption Standard (AES). *GeeksforGeeks* [online]. 14:16:06+00:00 [vid. 2024-12-29]. Dostupné z: https://www.geeksforgeeks.org/advanced-encryption-standard-aes/

[24] TUCKER, Amanda. Understanding RSA Asymmetric Encryption: How It Works. *SecureW2* [online]. 30. září 2024 [vid. 2024-12-29]. Dostupné z: https://www.securew2.com/blog/what-is-rsa-asymmetric-encryption

[25] *What Is a Man-in-the-Middle (MITM) Attack? | IBM* [online]. 11. červen 2024 [vid. 2024-12-29]. Dostupné z: https://www.ibm.com/think/topics/man-in-the-middle

[26] *Certifikační autorita* [online]. 2024 [vid. 2025-01-25]. Dostupné z: https://cs.wikipedia.org/w/index.php?title=Certifika%C4%8Dn%C3%AD_autorita&oldid=24327386

[27] *Java Development Kit Version 23 API Specification* [online]. [vid. 2025-01-28]. Dostupné z: https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/net/Socket.html

[28] RADEK. Typy testování software. *Radek Kitner* [online]. 20. září 2017 [vid. 2025-01-26]. Dostupné z: https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/

[29] *Průměrná odchylka – spocti.cz* [online]. [vid. 2025-01-27]. Dostupné z: https://www.spocti.cz/prumerna-odchylka/

[30] MUDRÁK, David. *Čeština: Schéma zapouzdření aplikačních dat na jednotlivých vrstvách rodiny protokolů TCP/IP - upravená verze* [online]. 11. říjen 2008 [vid. 2024-12-29]. Dostupné z: https://commons.wikimedia.org/wiki/File:TCP-IP_zapouzdreni_-_edited.svg

12 SEZNAM OBRÁZKŮ A TABULEK

13 PŘÍLOHA 1: TESTOVACÍ ZAŘÍZENÍ

```
OS: Linux Mint 22.1 x86_64

Host: 21M5002JCK ThinkPad E16 Gen 2

Kernel: 6.8.0-51-generic

CPU: AMD Ryzen 7 7735HS with Radeon

RAM: 14739MiB (+ 2,1 GB SWAP)
```