VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

Aplikovaná informatika

VÍCEVLÁKNOVÝ WEBOVÝ SERVER

Bakalářská práce

Autor práce: Filip Vojtko

Vedoucí práce: Mgr. Antonín Přibyl

Jihlava 2025

Vysoká škola polytechnická Jihlava

Tolstého 16, 586 01 Jihlava

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	**Filip Vojtko**
Studijní program:	Aplikovaná informatika
Obor:	Aplikovaná informatika
Garant studijního programu:	Ing. Lenka Kuklišová Pavelková, Ph.D.
Název práce:	**Vícevláknový webový server**
Vedoucí práce:	Mgr. Antonín Přibyl
Cíl práce:	Cílem je implementovat vícevláknový webový server v jazyce C++, který bude podporovat protokoly HTTP 1.0 a HTTP 1.1. Celý webový server bude vyvíjen pro použití na operačním systému Linux a bude koncipován jako systémová služba (démon). Server bude možné nakonfigurovat pomocí konfiguračního skriptu. Výkon serveru bude dále porovnán s ostatními webovými servery jako jsou Apache a Nginx.

Abstrakt

Práce se zabývá implementací vlastního vícevláknového webového serveru v programovacím jazyce C++ jako systémové služby (démon) v prostředí operačního systému Linux, s možností její konfigurace pomocí konfiguračních souborů. Webový server zahrnuje vlastní správu spojení a vláken, a vlastní implementaci protokolů HTTP/1.0 a HTTP/1.1 s možností šifrování (HTTPS). V teoretické části jsou popsány klíčové technologie, algoritmy a protokoly použité v implementaci webového serveru. Následně je popsána samotná implementace a její detaily. Popisuje kompilaci a balíčkovací systém projektu, rozsah a samotnou implementaci webového serveru, a použité knihovny. Na závěr je webový server porovnán s webovými servery Apache a Nginx.

Klíčová slova

webový server, HTTP, HTTPS, C++, Linux, démon, paralelní programování, síťové programování, šifrování, GNU Autotools

Abstract

The thesis deals with implementation of own custom multithreaded web server in C++ programming language as a system service (daemon) in environment of Linux operating system, with the possibility of its configuration using configuration files. The web server includes own connection and thread management, and own implementation of protocols HTTP/1.0 and HTTP/1.1 with encryption support (HTTPS). The theortical part describes key technologies, algorithms and protocols used for implementation of the web server. Next are described details of compilation and packaging system of the project, scope and implementation of web server itself, and used libraries. Finally, the web serveru is compared with Apache and Nginx web servers.

Keywords

web server, HTTP, HTTPS, C++, Linux, daemon, parallel programming, network programming, encryption, GNU Autotools

Prohlašuji, že předložená bakalářská práce je původní a zpracoval/a jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem v práci neporušil/a autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů, v platném znění, dále též „**AZ**“).

Byl/a jsem seznámen/a s tím, že na mou bakalářskou práci se plně vztahuje **AZ**, zejména § 60 (školní dílo).

Podle § 47b zákona o vysokých školách souhlasím se zveřejněním své práce podle Směrnice pro vedení, vypracování a zveřejňování závěrečných prací na VŠPJ, a to bez ohledu na výsledek obhajoby.

Beru na vědomí, že VŠPJ má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom/a toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem VŠPJ, která má právo ode mě požadovat přiměřený příspěvek na úhradu nákladů, vynaložených vysokou školou na vytvoření díla (až do jejich skutečné výše), z výdělku dosaženého v souvislosti s užitím díla či poskytnutím licence.

V Jihlavě dne 16. dubna 2025

…….

Podpis studenta/ky

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce Mgr. Antonínu Přibylovi za podporu a možnost vypracování bakalářské práce.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

AES	Advanced Encryption Standard
API	Application Programming Interface
CA	Certificate Authority
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CRUD	Create, Read, Update and Delete
DES	Data Encryption Standard
ECB	Encryption Code Book
ECDSA	Elliptic Curve Digital Signature Algorithm
GNU	GNU's Not Unix
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
IV	Initialization Vector
MIME	Multipurpose Internet Mail Extension
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
OFB	Output Feedback
PHP	Hypertext Preprocessor
PID	Process Identificator
REST	Representational State Transfer
RSA	Rivest-Shamir-Adleman
SMP	Symmetric Multiprocessing
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TID	Thread Identificator
TLS	Transport Layer Security

TOML	Tom’s Obvious Minimal Language
UDP	User Datagram Protocol
URI	Uniform Resource Identificator
URL	Uniform Resource Locator

Úvod

HTTP (Hypertext Transfer Protocol) či webové servery jsou v dnešní době jednou z nejrozšířenějších a nejvíce využívanou serverovou technologií. Jedná se o software, který umožňuje odesílání různého typu dat prostřednictvím počítačové sítě, kterou je například právě internet. Data, které z největší části posílají jsou HTML (webové) stránky, ale dokážou odesílat i velkou řadu jiných typů dat.

Webové servery jsou založeny na architektuře klient-server. Klient v architektuře představuje iniciátora spojení se serverem, tedy server nikdy nenavazuje spojení s klientem. Klient zasílá na server požadavky, který na základě tohoto patřičně odpovídá. Vzhledem k tomu že těchto požadavků může v jednu chvíli přicházet několik a je potřeba je zpracovat, je tedy nutné, aby webový server umožňoval paralelní zpracování požadavků s ohledem na dostupné hardwarové prostředky. Protože data přenášena mezi klientem a serverem nemusí být pouze webové stránky, ale může se jednat také o citlivá data, je nevyhnutelné, aby tato komunikace probíhala také šifrovaně.

Motivací pro výběr tématu byla vytvoření vlastního webového serveru se základní funkcionalitou a uplatnění a prohloubení svých znalostí v oblasti implementace protokolů a programování na systémech založených na Linuxu.

Cílem práce je tedy implementovat webový server v programovacím jazyce C++, který bude zahrnovat vlastní implementaci komunikačních protokolů HTTP/1.0 a HTTP/1.1, umožňovat paralelní zpracování požadavků za pomocí paralelního programování, a šifrovanou komunikaci mezi klientem a serverem (HTTPS – Hypertext Transfer Protocol Secure). Cílem je implementovat základní, resp. nejdůležitější části protokolu HTTP/1.0 a HTTP/1.1, není tedy cílem implementovat jejich celou specifikaci. Webový server bude koncipován jako systémová služba (démon) pro operační systémy založené na Linuxu. Služba bude možná konfigurovat pomocí konfiguračních souborů a upravovat tak její parametry. Výkon webového serveru bude následně porovnán s výkonem webových serveru Apache a Nginx. Webový server bude realizován jako statický, tedy bez možnosti dynamického generování webových stránek (např. pomocí PHP skriptů).

1 HTTP(S)

HTTP je protokol implementovaný na aplikační vrstvě (4. vrstva) modelu TCP/IP. Protokol byl původně vyvinut pro umožnění webové komunikace, tedy pro komunikaci mezi webovým prohlížečem a klientem. Jeho původním účelem byl tedy primárně stahování HTML dokumentů. Dnes ale díky své robustnosti, umožňuje protokol stahování širokého výčtu různých souborů různého formátu, a je využíván i pro jiné účely (např. pro REST API). HTTP funguje na architektuře *klient-server*. Principem této architektury je ten, že klient (např. webový prohlížeč) navazuje spojení se serverem a nikdy ne naopak (Mozilla, 2024a). Klient je často označován jako *user-agent*, a může reprezentovat jakýkoliv nástroj, který právě využívá HTTP pro komunikaci.

HTTP vyžaduje, aby použitý transportní protokol byl spolehlivý (Mozilla, 2024a). Spolehlivost znamená, že protokol umí potvrzovat přijatá data a podporuje kontrolu a znovu odeslání ztracených či poškozených dat. Z tohoto důvodu HTTP využívá pro přenos dat protokol TCP (Transmission Control Protocol). Výjimkou je ale HTTP/3.0, který již pro své přenos využívá protokolu UDP (User Datagram Protocol), který nesplňuje podmínky spolehlivosti, ale za to umožňuje efektivnější využití šířky pásma pro přenos dat. Protokol sám o sobě je bezstavový (*stateless*), tedy na serveru se neukládají žádné informace ohledně požadavků v rámci spojení (*session*) udržovaným mezi klientem a serverem. Pro ukládání dat v rámci spojení se používají *cookies*, které je server posílá klientovi v dopovědi, kde následně u klienta zůstávají uloženy např. ve souborech webového prohlížeče.

Přesto že je HTTP robustní, je navrhnut tak, aby byl velmi jednoduchý pro použití, jednoduše rozšiřitelný a čitelný (human-readable) (Mozilla, 2024a). Čitelnosti však není možné dosáhnout při použití protokolu HTTP/2.0, protože narozdíl od přechozích verzí protokolu nezasílá data v textové formě, ale ve formě binární.

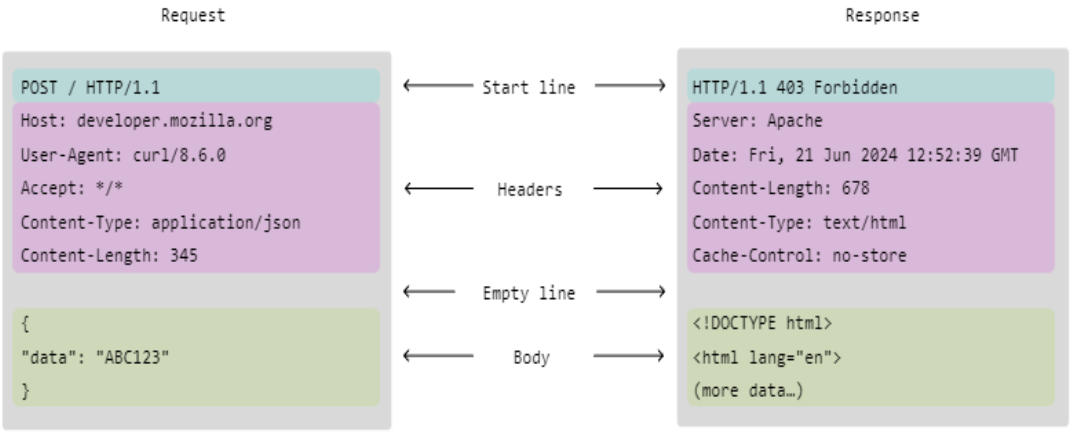
Rozdíl mezi HTTP a HTTPS je pouze ten, že HTTPS pouze využívá šifrování přenášených dat použitím šifrovacího protokolu SSL (Secure Sockets Layer) či TLS (Transport Layer Security). Protokoly verze HTTP/1.x nevyužívají šifrování implicitně, tedy HTTPS slouží jako nadstavba nad protokolem. Protokoly HTTP/2.0 a HTTP/3.0 již využívají šifrování implicitně.

1.1 Formát a struktura zprávy

HTTP zprávy jsou způsob, jakým si obě strany spojení, či relace, vyměňují data (např. server s klientem). Proto aby účastníci relace dokázali zprávám správně rozumět a zpracovávat je, tak je nutné, aby splňovaly a dodržovaly jistý formát a strukturu. Formát a struktura těchto zpráv je definován standardem jednotlivých verzí protokolu. HTTP rozlišuje dva typy zpráv, jimiž jsou požadavek (*request*) a odpověď (*response*).

1.1.1 HTTP/1.0 a HTTP/1.1

Formát a struktura zprávy je pro obě verze protokolu stejná. Obě verze protokolu mají stejný formát zpráv, a to textový. Struktura zprávy požadavku a odpovědi obou verzí protokolu je také stejná. Jejich struktura je zobrazena na obrázku níže (viz. obr. 1).



Obr. 1: Struktura požadavku a odpovědi protokolu HTTP/1.x
Zdroj: Mozilla (2024d)

Start line a hlavička je ve zprávě vždy povinná, ale tělo již povinné není. To, kdy je uvedeno i tělo zprávy, určuje především použitá metoda požadavku klienta. *Start line* v požadavku, také označován jako *request line*, má strukturu v následujícím pořadí: metoda požadavku, URI (Uniform Resource Identificator) specifikující požadovaný zdroj (*target resource*), a verzi HTTP. *Start line* v odpovědi, také označován jako *status line*, má strukturu v následujícím pořadí: verze HTTP, stavový kód a hláška stavového kódu. Hlavička zprávy obsahuje metadata, která jsou reprezentována atributy (tzv. *header fields*). Atributy se mezi verzemi protokolu liší, každá verze přináší nové atributy a některé přestává podporovat, resp. je ve zprávě ignoruje. Atributy se také liší tím, zda jsou určeny pro použití v žádosti, v odpovědi, či v obojím případě. Tělo zprávy obsahuje samotná přenášená data. Každý řádek ve zprávě je zakončen sekvencí \r\n, kterou je vždy ještě zakončena hlavička zprávy. Na tělo zprávy může být také použitá komprese (např. *gzip*), nikdy ale nemůže být použita pro hlavičku zprávy (Mozilla, 2024d).

1.2 Metody požadavku

Metody požadavku slouží k určení operace prováděné s daným cílovým zdrojem. Každá metoda má svůj vlastní účel a je používána pro jiný typ situace. Použitá metoda v požadavku také určuje, které atributy by hlavička měla obsahovat.

1.2.1 GET

Metoda *GET* slouží pro získání či stáhnutí specifikovaného cílového zdroje. Zpráva požadavku neobsahuje tělo, tedy obsahuje pouze hlavičku. Tělo je naopak až součástí odpovědi.

1.2.2 HEAD

Metoda *HEAD* je obdobou metody *GET.* Rozdíl spočívá v tom, že zde součásti odpovědi nejsou data specifikovaného cílového zdroje, ale pouze hlavička.

1.2.3 POST

Metoda *POST* slouží pro vytvoření nového, dosud neexistujícího zdroje na straně příjemce požadavku. Součástí požadavku je proto i tělo s daty. Naopak v odpovědi se nachází pouze hlavička se stavovým kódem, který popisuje výsledek operace. Metoda je využívána zejména pro odesílání dat formulářů.

1.2.4 PUT

Metoda *PUT* je velmi podobná metodě *POST*. Základním rozdílem je ten, že metoda *PUT* lze využít i pro aktualizaci cílového zdroje. Rozdíl ve vytváření nového cílového zdroje je ten, že zde stejný požadavek vždy stejný výsledek. U metody *POST* je pro stejný požadavek vždy jiný výsledek, což se označuje jako tzv. *idempotency* (Mozilla, 2024b).

1.2.5 DELETE

Metoda *DELETE* slouží pro odstranění existujícího cílového zdroje. Požadavek ani odpověď neobsahuje tělo zprávy, ale pouze hlavičku.

1.2.6 CONNECT

Metoda *CONNECT* slouží zejména pro případy, kdy klient je umístěn v počítačové síti za proxy serverem a přeje si navázat šifrované spojení (HTTPS). Klient pomocí této metody naváže s proxy spojení, kde v požadavku specifikuje URL (Uniform Resource Locator), či cílovou adresu, a port, na který se chce připojit. Pokud proxy server umožní spojení, chová se následně pro odesílané požadavky a přijímané odpovědi pouze jako tunel, tedy požadavky klienta přeposílá serveru, a odpovědi serveru přeposílá klientovi. Metoda je součástí protokolu až od verze HTTP/1.1.

1.2.7 PATCH

Metoda *PATCH* slouží pro aktualizování cílového zdroje. Narozdíl od metody *PUT* umožňuje aktualizovat pouze část cílového zdroje, metoda *PUT* aktualizuje celý cílový zdroj. Tímto se dá optimalizovat náročnost pro aktualizaci zdroje, kde není nutné zasílat opět celý zdroj pro jeho aktualizaci. Metoda je součástí protokolu až od verze HTTP/1.1. Metoda je ale nejčastěji využívána jako jedna z *CRUD* (Create, Read, Update and Delete) operací, např. pro aktualizaci záznamu v databázi.

1.2.8 OPTIONS

Metoda *OPTIONS* slouží pro získání informací a nastavení k danému zdroji. Odpověď neobsahuje tělo, tedy samotná data cílového zdroje, ale informace jako např. povolené metody, které lze použít při manipulaci se zdrojem (atribut *Allow*).

1.3 Stavové kódy

Stavový kód je vždy součástí odpovědi a slouží pro specifikaci výsledku operace požadavku. Je reprezentován v odpovědi jako celé číslo, které následuje zněním stavového kódu. Stavový kód v odpovědi závisí na použité metodě požadavku, s čímž souvisí také obsažené atributu v hlavičce požadavku. Pro atributy standard definuje, jak má vypadat odpověď pro konkrétní situace, resp. jaký stavový kód má odpověď obsahovat. Stavové kódy se rozdělují do několika skupin, dle jejich účelu.

1.3.1 Informational

Stavové kódy tohoto typu jsou označovány jako *1xx* a využívají pro stavový kód číslo z rozsahu od 100 do 199. Slouží jako stavový kód pro provizorní odpověď. Skupina těchto stavových kódů není součástí verze HTTP/1.0. Příklady jsou stavové kódy *100 Continue* a *101 Switching Protocols*.

1.3.2 Successful

Označovány jako *2xx*, kde pro stavový kód využívají číslo v rozsahu od 200 do 299. V odpovědi specifikuje, že server požadavku rozuměl a že jej zpracoval. Příklady jsou stavové kódy *200 OK*, *201 Accepted*, *204 No Content* či *206 Partial Content*.

1.3.3 Redirection

Označovány jako *3xx*, kde pro stavový kód využívají číslo v rozsahu od 300 do 399. V odpovědi specifikuje, že je klientem vyžadována další akce navíc, pro dosažení splnění požadavku. Příklady jsou stavové kódy *301 Moved Permanently* či *304 Not Modified*.

1.3.4 Client error

Označovány jako *4xx*, kde pro stavový kód využívají číslo v rozsahu od 400 do 499. V odpovědi specifikuje chybu na straně klienta, resp. chybu požadavku. Příklady jsou stavové kódy *400 Bad Request*, *403 Forbidden* či *404 Not Found*.

1.3.5 Server error

Označovány jako *5xx*, kde pro stavový kód využívají číslo v rozsahu od 500 do 599. V odpovědi specifikuje, že se vyskytla chyba nebo není schopen zpracovat požadavek. Příklady jsou stavové kódy *500 Internal Server Error*, *501 Not Implemented* či *503 Service Unavailable*.

1.4 Navazování spojení

Před samotným přenosem dat mezi klientem a serverem je potřeba nejdříve navázat spojení. Vzhledem k tomu, že HTTP je založený na architektuře *client-server*, tak spojení iniciuje vždy klient. HTTP/1.x využívá pro veškerou komunikaci protokolu TCP, který je implementován na transportní vrstvě (3. vrstva) modelu TCP/IP. Klient navazuje se serverem spojení na portu *80*, který je standartním portem používaný pro komunikaci prostřednictvím HTTP. Dalšími porty používané pro HTTP jsou *8000* a *8080* (Mozilla, 2024c). Pokud klient navazuje šifrované spojení (tj. HTTPS), tak spojení se serverem navazuje na portu *443*. Navazování spojení jako takového zařizuje již protokol TCP, a to využitím mechanismu zvaným *three-way handshake*. Pro navázání šifrovaného spojení pro HTTPS následuje ještě proces *SSL/TLS handshake*.

1.5 Správa spojení

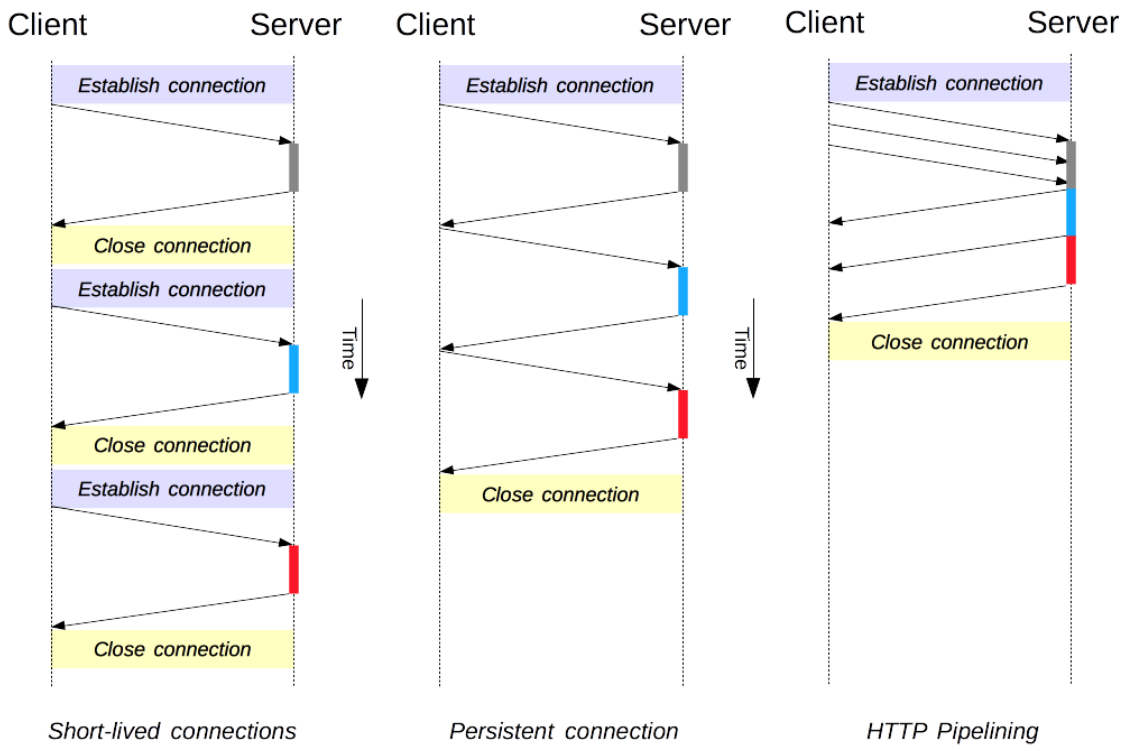
Po navázání komunikace se serverem již existuje platné spojení, které lze využít pro přenos dat. Spojení je potřebné také jistým způsobem spravovat. Každá verze protokolu pracuje se spojením trochu odlišně.

1.6 HTTP/1.0

HTTP/1.0 pracuje se spojením tak, že po každé sekvenci požadavku a odpovědi je spojení ukončeno (tzv. *short-lived connection*). Z dnešního hlediska je tento velmi neefektivní, protože se musí pro každý požadavek vytvořit nové spojení, které následně musí hned uzavřít, čímž vzniká jistá režie. Další režie vzniká tím, že při vytvoření spojení probíhá tzv. *TCP slow start* (Mozilla, 2024d). Ten vytváří režii pro přenos dat tak, že na začátku spojení je možné odeslat pouze menší množství dat, před přijetím potvrzení o jejich přijetí od druhé strany spojení.

1.7 HTTP/1.1

HTTP/1.1 odstraňuje předešlý problém tím, že v rámci jednoho spojení je možné odesílat požadavky a přijímat odpovědi, dokud nebude spojení ze strany klienta či serveru ukončeno (tzv. *persistent connection*). Celý průběh výměny dat funguje tak, že klient po odeslání každého požadavku musí počkat na přijetí celé odpovědi, než může odeslat další požadavek, což vytváří problém zvaný jako *head-of-line blocking*, který limituje propustnost (Mozilla, 2024d). Sekvenční přístup se pokouší nahradit tzv. *pipelining*, který umožňuje odeslat více požadavků v rámci jednoho spojení bez nutnosti čekat na jejich odpověď. Problém je zde ale v tom, že server musí poskytnou odpovědi ve stejném pořadí jako přijal odpovědi. Problém je tedy přesun pouze na druhou stranu spojení. *Pipelining* nemá ale velmi dobrou podporu, a proto je také v současnosti implicitně vypnutý jak na webových serverech, tak ve webových prohlížečích (Mozilla, 2024d). Webové prohlížeče ale řeší *head-of-line blocking* tím, že se serverem navážou více spojení.



Obr. 2: Správa HTTP spojení

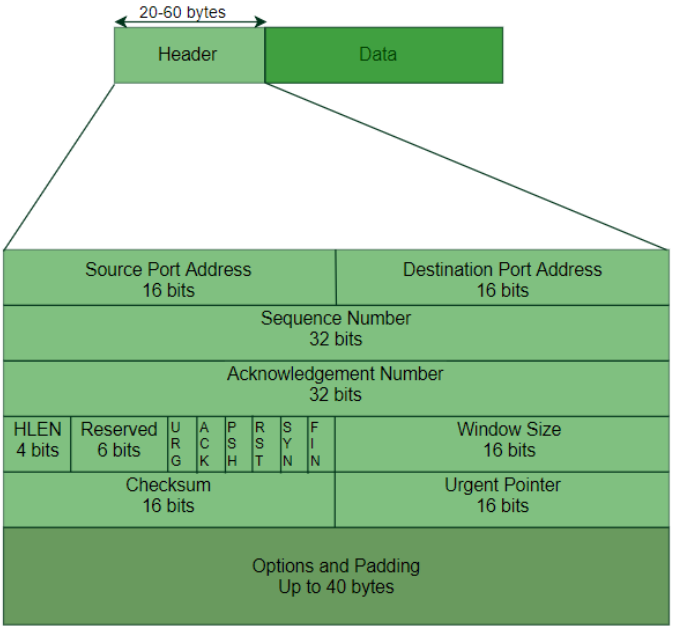
Zdroj: Mozilla (2024b)

2 TCP

TCP je protokol implementovaný na transportní vrstvě (3. vrstva) TCP/IP modelu. Jedná se o spojově orientovaný protokol, tedy před jakýmkoliv přenášením dat je nejdříve nutné navázat platné spojení mezi koncovými uzly spojení. Základní jednotkou, se kterou protokol pracuje, je *segment*. TCP zajišťuje spolehlivý přenos dat, což znamená to, že pokud jsou nějaká data ztracená či poškozená, zajišťuje opětovné odeslání těchto dat. Zajišťuje navíc to, že data jsou předány aplikacím ve správném pořadí (Grigorik, 2013) a každý přijatý segment je potvrzován.

2.1 Struktura segmentu

Každý segment má pevnou velikost označovanou jako *MSS* (Maximum Segment Size). *MSS* je odvozeno z *MTU* (Maximum Transmission Unit), která udává maximální množství dat, které datová linka dokáže přenést najednou. Struktura segmentu je znázorněna níže (viz. obr. 3).



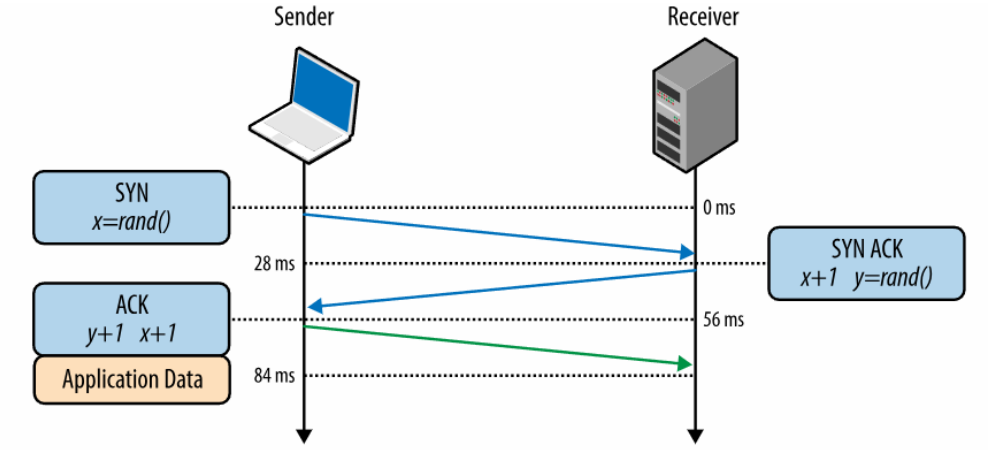
Obr. 3: Struktura TCP segmentu
Zdroj: GeeksforGeeks (2021)

2.2 Segmentace

Segmentace je proces rozdělení dat na segmenty o velikosti *MSS*. Umožňuje aplikacím bezstarostně předat data transportní vrstvě o libovolné velikosti. Potom co data dorazí příjemci se TCP opět postará o to, že jednotlivé segmenty předá aplikaci ve správném pořadí. Správné pořadí segmentů se zajišťuje pomocí *sequence* a *acknowledge number*. Obě strany spojení si udržují svoje sekvenční číslo a je zároveň součástí každého segmentu. Určuje pořadí prvního bytu každého segmentu. Acknowledge number pak slouží pro potvrzení přijatého segmentu takovým způsobem, že k sekvenčnímu číslu v segmentu přičte velikost datového bloku segmentu.

2.3 Navazování a ukončování spojení

Navazování komunikace zajišťuje tzv. *three-way handshake*, který se skládá ze 3 kroků. Prvním krokem je, že klient iniciuje spojení odesláním segmentu, ve kterém je v hlavičce nastaven bit *SYN*. Server po přijetí segmentu odešle segment, kde nastaví bity hlavičky *SYN* a *ACK*, čímž potvrzuje přijetí segmentu. Po obdržení tohoto segmentu následuje poslední krok, ve kterém klient odešle segment, který má nastavený *ACK* bit v hlavičce.



Obr. 4: Three-way handshake

Zdroj: Grigorik (2013)

Pro možnost navázání spojení musí ale server naslouchat na tzv. *socketu*. Socket je nástroj využívaný pro síťovou komunikaci a představuje koncový bod spojení. Socket má v operačním systému alokovanou dočasnou paměť (*buffer*) pro dočasné uložení přijatých dat, než budou přečteny aplikací. Socket je vždy vytvořen na obou stranách spojení během jeho navazování. Socket je kombinací IP adresy a portu, proto každý vytvořený socket otevírá nový port.

Ukončování spojení je uskutečněn procesem zvaným *four-way handshake* a může jej iniciovat klient i server. Skládá se z celkem 4 kroků. Prvním krokem je, že iniciátor spojení odešle segment, ve kterém je nastaven bit *FIN*. Druhá strana spojení odešle segment s nastaveným *ACK* bitem. Tímto se socket iniciátora spojení zavírá a přechází do stavu *time wait*. V tomto stavu čeká na ukončení spojení z druhé strany. Druhá strana spojení odešle segment s nastaveným bitem *ACK*, který mu iniciátor ukončení spojení odpovídá segmentem s nastaveným bitem *ACK*. Po přijetí potvrzení a uplynutí timeoutu stavu *time wait* je spojení kompletně ukončeno.

3 Symetrická kryptografie

Jedná se o šifrovací systém, který využívá pro šifrování i dešifrování dat stejného sdíleného privátního klíče (Klíma, 2005, s. 4). Z toho plyne, že tento klíč nesmí vlastnit nikdo jiný než účastníci spojení, v takovém případě by komunikace nebyla bezpečná.

3.1 Proudové šifry

Proudové šifry se typ symetrického šifrování, kdy se šifruje každý bit či byte dat. Pro šifrování dat se využívá privátní sdílený klíč a IV (Initialization Vector), které jsou vstupem pro generátor proudu klíčů (tzv. *keystream generator*). Klíč slouží pro samotné utajení dat a IV zodpovídá za nastavení generátoru do nějaké jiné výchozí pozice. Generátor proudu klíčů následně generuje proud klíče (tzv. *running key*), který je následně použit pro šifrování dat. Šifrování dat je realizováno pomocí logické operace *XOR* (Exclusive Or), což je i důvodem proč má proud klíče stejnou délku jako šifrovaná data. Využívá se zde inverzní vlastnosti této operace, která je definována následovně: *(A XOR B) XOR B = A*. Parametr *A* zde reprezentuje data pro zašifrování a parametr *B* proud klíče. Operace XOR je výpočetně jednoduchá a rychlá, ale není tolik bezpečná. Kvůli tomu se pro každou zprávu generuje nový IV a generuje se opět nový proud klíče, což zaručí to, že i za použití stejného klíče bude zpráva vždy šifrována jinak. IV se vždy předávají mezi stranami spojení vždy v otevřené formě (Klíma, 2005, s. 13). Proto každý IV vygenerovaný v rámci jednoho spojení by měl být unikátní. Z důvodu rychlosti šifrování dat jsou proudové šifry používané pro nepřetržitý proud dat, kde pokud se nějaká data ztratí, dešifrovací algoritmus na ně nemusí čekat, aby mohl data dešifrovat.

3.2 Blokové šifry

Blokové šifry na rozdíl od šifer proudových šifrují data po blocích o velikosti *N.* Tedy data pro zašifrování se rozdělí na bloky pevné délky, které vstupují do šifrovací funkce spolu se sdíleným privátním klíčem. Blokové šifry narozdíl od výše popsaných šifer nepoužívají pouze techniku substituci, ale také transpozici. Nenahrazuje tedy jednotlivé znaky (byty či bity) otevřeného textu jinými znaky, ale také mění jejich pořadí. Blokové šifry se rozdělují podle toho, v jakých operačních módech pracují. Těmito módy jsou *ECB* (Electronic Code Block), *CBC* (Cipher Block Chaining), *CFB* (Cipher Feed Back) a *OFB* (Output Feed Back) (Tutorialspoint, 2024a).

ECB je základní operační mód, který funguje na principu toho, že každý blok otevřeného textu je šifrován samostatně za použití klíče a šifrovací metody. Jeho výhodou je, že pokud při přijetí dat je v nějakém šifrovaném bloku chyba, tak chyba se nešíří dál do ostatních bloků a ovlivní pouze daný blok. Nevýhodou naopak je, že pokud je blok otevřených dat stejný, tak vznikne také stejný šifrovaný blok. Proto není doporučeno používat bloky o velikosti 40 bitů a méně (Tutorialspoint, 2024b).

Dalším módem je *CBC*, který již funguje tak, že pro šifrování bloku otevřeného textu již vyžívá předešlý šifrovaný blok. Pro aktuální blok dat otevřeného textu se provede operace XOR s předešlým, již šifrovaným blokem dat. Výstup této operace je následně zašifrován. Pro zašifrování prvního bloku dat se využívá IV. Tento IV se následně přidá k šifrovaným datům, ale v nešifrované formě, tak aby data bylo také možné dešifrovat. Výhodou tohoto módu je větší

bezpečnost šifrování než předchozí mód. Nevýhodami naopak jsou větší výpočetní složitost kvůli operacím navíc. Navíc chyba v šifrovaném bloku přenáší chybu na sousední blok. Šifrování se dá zapsat jako $C_i = E_k(P_i \ XOR \ C_{i-1})$ a dešifrování jako $P_i = D_k(C_i) \ XOR \ C_{i-1}$, kde $i > 0$ a C_0 je IV (Tutorialspoint, 2024c). Platí, že C_i je zašifrovaný blok, P_i je blok otevřeného textu a E_k a D_k jsou šifrovací a dešifrovací funkce, které transformují data na základě klíče k.

Mód *CFB* je podobný jako mód jako *CBC*, s tím rozdílem, že pro zašifrování otevřeného textu zde vstupuje pouze předchozí zašifrovaný blok dat otevřeného textu. Proces šifrování lze zapsat jako $C_i = P_i \ XOR \ E_k(C_{i-1})$ a dešifrování jako $P_i = C_i \ XOR \ E_k(C_{i-1})$, kde $i > 0$ a C_0 je IV (Tutorialspoint, 2024d).

Posledním módem je *OFB*, který je také velmi podobný jako *CFB* či *CBC*. Zde ale pro zašifrování bloku otevřeného textu vstupuje do šifrovací funkce pouze šifrovaný IV. Šifrování lze zapsat jako $C_i = P_i \ XOR \ V_i$ a dešifrování jako $P_i = C_i \ XOR \ V_i$, kde $V_i = E_k(V_{i-1})$ a jedná se o IV (Tutorialspoint, 2024e). Nevýhodou této metody je, že mohou vznikat opakovaným šifrováním stejné šifrované IV. Toto může vést ke vzniku jistých vzorů v šifrovaných datech (Tutorialspoint, 2024e).

3.2.1 AES

Bloková šifra *AES* (Advanced Encryption Standard) šifruje bloky o velikosti 128 bitů. Jako klíč podporuje tři různé velikosti: 128, 192 a 256 bitů. Velikost klíče určuje také počet rund. Pro 128bitový klíč je rund 10, pro 196bitový rund 12, a pro 256bitový rund 14 (Klíma, 2005, s. 33). Z klíče je během šifrování vytvořeno celkem *4 + Počet rund * 4* podklíčů o velikosti 32 bitů, které jsou z původního klíče vytvářeny definovaným způsobem (Klíma, 2005, s. 33). Každá runda využívá celkem 4 tyto podklíče (128 bitů) a je tvořena čtyřmi kroky jdoucími za sebou: *ByteSub*, *ShiftRow*, *MixColumn*, *AddRoundKey*. Jediný rozdíl je pro poslední rundu, která z těchto kroků vynechává krok *MixColumn*. Před samotným šifrováním bloku dat je ale blok rozdělen na matici 4x4 bytů. První operací je *ByteSub*, která pomocí S-boxů realizuje substituci každého z bytu matice za jiný. Operace je nelineární, což zajišťuje bezpečnost šifry. Druhou operací je *ShiftRow*, která posouvá jednotlivé řádky matice o daný počet bytů doleva. První řádek není nijak posunut, v druhém řádku se každý byte posune o 1 byte, ve třetím řádku o 2 byty, a ve třetím o 3 byty. Třetím krokem je *MixColumn*, kde pro každý sloupec matice je vypočítán nový sloupec, kde tento výpočet je dán následujícím vztahem (viz. obr. 5).

$$
\begin{aligned}
b_0 &= 0x02*a_0 \oplus 0x03*a_1 \oplus 0x01*a_2 \oplus 0x01*a_3 \ , \\
b_1 &= 0x01*a_0 \oplus 0x02*a_1 \oplus 0x03*a_2 \oplus 0x01*a_3 \ , \\
b_2 &= 0x01*a_0 \oplus 0x01*a_1 \oplus 0x02*a_2 \oplus 0x03*a_3 \ , \\
b_3 &= 0x03*a_0 \oplus 0x01*a_1 \oplus 0x01*a_2 \oplus 0x02*a_3 \ ,
\end{aligned}
$$

Obr. 5: Výpočet hodnot sloupce operace MixColumn

Zdroj: Klíma (2005, s. 34)

Zde b_0 až b_4 odpovídají bytům sloupce výstupní matice. Dále a_0 až a_4 jsou byty sloupce dat pro zašifrování. Posledním krokem je *AddRoundKey*, kde pro každý byte matice je provedena operace XOR s bytem na stejné pozici podklíče.

4 Asymetrická kryptografie

Asymetrická kryptografie se využívá v kombinaci se symetrickou kryptografií. Symetrická kryptografie slouží pro šifrování přenášených dat během spojení, za využití jednoho stejného sdíleného privátního klíče. Asymetrická kryptografie je ale používána pro bezpečnou distribuci tohoto sdíleného privátního klíče mezi účastníky spojení. Narozdíl od symetrické kryptografie se zde využívají dva klíče: *veřejný* a *privátní* (Neckář, 2016). Veřejný klíč je používán pro šifrování dat a může být znám komukoliv. Privátní klíč je naopak využíván pro dešifrování a nesmí jej znát nikdo jiný, než kdo daný klíč vytvořil, jinak by komunikace nebyla považována za bezpečnou. Pro tyto dva klíče musí platit, že vše zašifrováno veřejným klíčem lze dešifrovat pouze klíčem privátním. Zároveň musí platit, že vše zašifrováno privátním klíčem lze dešifrovat pouze klíčem veřejným (Neckář, 2016), což umožňuje využít asymetrického šifrování a pro *digitální podpis*. Digitální podpis se využívá pro ověření identity (autentizace). Funguje na principu toho, že pro nešifrovaná data spočítáme otisk zprávy (hash), který následně zašifrujeme privátním klíčem a přiložíme k samotným datům zprávy. Příjemce dešifruje otisk zprávy veřejným klíčem a spočítá si otisk zprávy samotných dat. Pokud se otisky shodují, je identita odesílatele ověřena (Neckář, 2016).

4.1 RSA

RSA (Rivest-Shamir-Adleman) je asymetrická šifra, která je založena na vlastnosti prvočísel. Jedná se o vlastnost, kde výsledek vynásobení dvou velmi velkých prvočísel nelze rozložit zpět na daná prvočísla v nějakém rozumném čase. Princip šifry je následující. Strana *A* vygeneruje dvě náhodná a velmi velká prvočísla *p* a *q*. Tato čísla dále vynásobí. Výsledek jejich násobení je označován jako *n*. Dále se vypočítá hodnota Eulerovy funkce, dána vztahem $\varphi(n) = (p-1)*(q-1)$. Dále je zvoleno číslo *e*, pro které platí, že $e < \varphi(n)$ a zároveň je nesoudělné s $\varphi(n)$. Nakonec se spočítá číslo *d*, které je multiplikativní inverzí čísla *e*. Číslo *d* musí být takové, aby platilo $e*d \equiv 1 \pmod{\varphi(n)}$. Strana *A* poskytne straně *B* veřejný klíč (n, e) a privátní klíč (n, d) si uchová. Šifrování je realizováno jako $c = z^e \bmod n$. Dešifrování následně jako $z = c^d \bmod n$.

4.2 Diffie-Hellman

Jedná se o algoritmus pro bezpečnou výměnu sdíleného privátního klíče mezi stranami spojení. Algoritmus je založen na problému *diskrétního logaritmu*. Problém spočívá v tom, že pokud máme čísla základ a exponent, je jednoduché spočítat výsledek. Pokud ale máme výsledek a základ, je velmi obtížné se dostat k exponentu, obzvlášť pokud jsou základ a exponent velká čísla. Algoritmus funguje na takovém principu, že strana *A* a *B* se domluví na společném základu *q* a modulu *p*. Modulo *p* je velmi velké prvočíslo a *q* je obecně celé číslo. Dále strana *A* vygeneruje číslo *a* a stana *B* číslo *b*. Čísla *a* a *b* tvoří privátní klíč a nepředávají se. Strana *A* provede výpočet $x = g^a \bmod p$ a strana *B* provede $y = g^b \bmod p$. Výsledky výpočtu si strany pošlou a přichází výpočet samotného sdíleného privátního klíče. Strana *A* provede $k = y^a \bmod p$ a strana *B* provede $k = x^b \bmod p$. Klíč *k* je následně použit pro šifrování dat.

5 SSL/TLS

SSL a TLS jsou šifrovací protokoly, který pro šifrování, zjištění integrity dat a autentizaci využívá několik algoritmů. SSL protokol má celkem tři verze: 1.0, 2.0 a 3.0. Všechny tyto verze protokolu nejsou již dnes považovány za bezpečné a jejich podpora na zařízeních by měla být vypnutá. Nástupcem SSL je právě TLS, který má také celkem čtyři verze: 1.0, 1.1, 1.2 a 1.3. Verze 1.0 a 1.1 jsou dnes již považovány také za zastaralé, a ne tolik bezpečné (SSL, 2024). Verze 1.2 a 1.3 jsou ale považovány za bezpečné. Na obrázkách níže jsou zobrazeny algoritmy využívané protokoly SSL a TLS (viz. obr. 6 a 7).

SSL Protocols	OpenSSL Identifier	Key Exchange	Authentication	Encryption	MAC
SSLv3, TLSv1	DHE-DSS-AES128-SHA	Ephemeral Diffie-Hellman	DSA	AES(128)	SHA1
SSLv3, TLSv1	DHE-DSS-AES256-SHA	Ephemeral Diffie-Hellman	DSA	AES(256)	SHA1
SSLv3, TLSv1	EDH-DSS-DES-CBC3-SHA	Ephemeral Diffie-Hellman	DSA	3DES(168)	SHA1
SSLv3, TLSv1	IDEA-CBC-SHA	RSA	RSA	IDEA(128)	SHA1
SSLv3, TLSv1	AES128-SHA	RSA	RSA	AES(128)	SHA1
SSLv3, TLSv1	AES256-SHA	RSA	RSA	AES(256)	SHA1
SSLv3, TLSv1	DES-CBC3-SHA	RSA	RSA	3DES(168)	SHA1
SSLv3, TLSv1	DHE-RSA-AES128-SHA	Ephemeral Diffie-Hellman	RSA	AES (128)	SHA1
SSLv3, TLSv1	DHE-RSA-AES256-SHA	Ephemeral Diffie-Hellman	RSA	AES(256)	SHA1
SSLv3, TLSv1	EDH-RSA-DES-CBC3-SHA	Ephemeral Diffie-Hellman	RSA	3DES(168)	SHA1
SSLv3, TLSv1	RC4-MD5	RSA	RSA	RC4(128)	MD5
SSLv3, TLSv1	RC4-SHA	RSA	RSA	RC4(128)	SHA1

Obr. 6: Algoritmy používané v SSL 3.0 a TLS 1.0

Zdroj: Hstechdocs (2013)

Cipher Suite Name	Protocol (SSL_METHOD)	Key Exchange	Authentication	Encryption	Message Digest
AES256-GCM-SHA384	TLSv1.2	RSA	RSA	AESGCM(256)	AEAD
AES256-SHA256	TLSv1.2	RSA	RSA	AES(256)	SHA256
AES256-SHA	TLSv1.2 TLSv1.1	RSA	RSA	AES(256)	SHA1
AES128-GCM-SHA256	TLSv1.2	RSA	RSA	AESGCM(128)	AEAD
AES128-SHA256	TLSv1.2	RSA	RSA	AES(128)	SHA256
AES128-SHA	TLSv1.2 TLSv1.1	RSA	RSA	AES(128)	SHA1
DES-CBC3-SHA	TLSv1.2 TLSv1.1	RSA	RSA	3DES(168)	SHA1
DHE-RSA-AES128-SHA256	TLSv1.2	DH	RSA	AES(128)	SHA256
NULL-SHA256	TLSv1.2	RSA	RSA	None	SHA256

Obr. 7: Algoritmy používané v TLS 1.1 a TLS 1.2
Zdroj: Broadcom (2024)

SSL/TLS jsou založeny na *CA* (Certificate Authority) certifikátech, které umožňují ověření identity a předání potřebných informací pro možnost šifrování komunikace. S těmito certifikáty jsou úzce spojeny certifikační autority. Jedná jsou důvěryhodné instituce, které slouží pro vydávání samotných CA certifikátů v platnost. Takový vydaný certifikát svazuje majitele certifikátu s veřejným klíčem. Certifikáty obsahují data jako platnost certifikátu, majitel certifikátu a veřejný klíč. Obsahuje také digitální podpis samotné certifikační autority, která slouží pro ověření identity subjektu (např. webové stránky).

Certifikát je ale nutný nějak mezi stranami spojení předat. Předání certifikátu je jeden z kroků je procesu *SSL/TLS handshake*, který se skládá z několika kroků. Prvním krokem je *ClientHello*, ve kterém klient pošle serveru verzi SSL/TLS a algoritmy pro šifrování (tzv. *cipher suites*), které klient podporuje. Následuje krok *ServerHello*, ve kterém server odpovídá klientovi, kde si vybírá nejvyšší možnou verzi SSL/TLS a algoritmy, které z poskytnutého seznamu také podporuje. Server dále pošle klientovi svůj CA certifikát. Klient přijatý certifikát ověří, resp. zkontroluje datum splatnosti, zda nebyl certifikát revokován, a ověří digitální podpis certifikační autority pomocí *root* certifikátů, které má ve webovém prohlížeči či operačním systému dostupné. Klient dále vygeneruje klíč, který bude následně využíván pro symetrické šifrování. Klíč zašifruje přijatým klíčem v CA certifikátu a odešle jej serveru. Server si přijatý klíč pomocí privátního klíče dešifruje a tím je *SSL/TLS handshake* dokončen a může začít šifrovaná komunikace mezi klientem a serverem (SSL, 2023).

6 Paralelní programování

Paralelní programování je koncept, který má za účel zrychlit vykonávání operací. Základní myšlenka je, že se jednotlivé úlohy nevykonávají sériově, tedy postupně za sebou, ale jsou vykonávány souběžně (Oracle, 2008). Přístup paralelního programování dokáže v určitých situacích velmi zrychlit chod a odezvu aplikace, ale přináší také jisté problémy. Základní problém, který paralelní programování přináší, je synchronizace a práce se sdílenými daty (Faigl, 2016, s. 21). Problém spočívá v tom, že pokud více aplikací, resp. více vláken v kontextu jedné aplikace, přistupuje ke stejným datům zároveň, je nutné tento stav ošetřit, tak aby nedocházelo k tvz. *data races*. Ty vedou k nepředvídatelnému a nedefinovanému chování (Teofilo, 2023). Vzniká právě tehdy když jeden proces či vlákno daný sdílený zdroj modifikuje, a ostatní z něj pouze čtou, nebo ho také modifikují. K ošetření takovýchto situací slouží právě níže popsané synchronizační nástroje. Skutečného paralelismu je ale možné dosáhnout pouze s hardwarovou podporou, tedy pokud procesor disponuje více než jednou výpočetní jednotkou, resp. jádry (Oracle, 2008). V reálných systémech, kde běží stovky až tisíce procesů, avšak skutečného paralelismu není možné dosáhnout. Obecně v systémech, kde je možný běh více úloh, než má procesor jader. V takovém případě lze dosáhnout pouze souběžnosti (Oracle, 2008). Souběžnost je vlastnost, na které jsou postavené plánovače úloh v moderních operačních systémech. Fungují tak, že každé jádro má svoji frontu úloh, které čekají na vykonání. Plánovač následně vykonává jednotlivé úlohy tak, že jim přiřazuje stejný časový interval (Oracle, 2008), po který mohou být úlohy vykonávány na daném jádře procesoru. Po vyprchání časového intervalu je vykonávání úlohy přerušeno, je zařazena opět na konec fronty úloh, a následně spuštěna další úloha v pořadí (příklad *round-robin* plánovače (Oracle, 2008)). Tento přístup se nazývá *SMP* (Symmetric Multiprocessing). Každý proces může ale spouštět další vlákna, která také vykonávají jistou úlohu. Proto se v linuxových systémech zavádí pojmy *PID* (Process Identificator) a *TID* (Thread Identificator). V linuxových systémech je víceméně vlákno to samé jako proces, s tím rozdílem že ale vlákno sdílí stejný virtuální prostor (paměť) s procesem, který jej spustil (Oracle, 2008), na rozdíl právě od vytvoření samotného podprocesu (např. systémové volání *fork*).

6.1 Synchronizační nástroje

Synchronizační nástroje slouží k ošetření výše zmíněných problémů, které vznikají během souběžného přistupování ke sdíleným datům. Nástroje bývají implementovány na nejnižší úrovni přímo v operačním systému. Těchto služeb operačního systému pro řízení přístupu ke sdíleným datům následně využívají knihovny programovacích jazyků, které tyto nástroje umožňují jednoduše použít.

6.1.1 Exkluzivní zámek

Jedná se o základní synchronizační nástroj, který je znám také pod názvem *mutex* (Mutual Exclusion). Nástroj funguje na principu exkluzivního přístupu ke sdílenému zdroji. Znamená to tedy, že pouze jeden proces může přistupovat k danému sdílenému zdroji (Yoru, 2023). Daný proces, který zámek zamkne, jej musí také odemknout. Pokud by nedošlo k odemknutí procesem, který zámek vlastní, nastala by situace zvaná *deadlock*, kdy by se žádný

jiný proces nedokázal dostat ke sdílenému zdroji. Pokud jeden proces vlastní zámek a jiný proces si jej vyžádá, tak implicitním chováním zámku je že je blokující, tedy druhý proces bude uspán, dokud zámek proces neuvolní. V programovacím jazyce C++ je exkluzivní zámek dostupný od verze C++11 jako *std::mutex*. V programovacím jazyce C je definován standardem POSIX jako *pthread_mutex_t*.

6.1.2 Sdílený zámek

Sdílený zámek nástroj je obdobou předchozího exkluzivního zámku, ale s tím rozdílem, že se zde rozlišují dva typy přístupu ke sdílenému zdroji. Těmito typy přístupů jsou přístup pouze pro čtení a přístup pro modifikaci sdíleného zdroje. Pokud více procesů potřebuje zároveň souběžně číst ze sdíleného zdroje, tak zámek bude zamknut ve sdíleném stavu. V tomto stavu nelze sdílený zdroj nijak modifikovat, tedy pokud má alespoň jeden proces sdílený zámek, jiný proces si nedokáže vzít zámek exkluzivně, ale pouze sdíleně. Naopak pokud si proces vezme exkluzivní zámek, tak je možné daný sdílený zdroj modifikovat a žádný jiný proces si nedokáže vzít sdílený ani exkluzivní zámek (Yoru, 2023). V programovacím jazyce C++ je sdílený zámek dostupný od verze C++17 jako *std::shared_mutex*. V programovacím jazyce C je definován standardem POSIX jako *pthread_rwlock_t*.

6.1.3 Atomická proměnná

Vzhledem k tomu, že získání zámku vytváří jistou režii, existuje tak jiný, rychlejší mechanismus pro přístup ke sdíleným datům, který se nazývá *atomická proměnná*. Jedná se o mechanismus, který využívá hardwarové podpory pro exkluzivní přístup k dané části paměti (Toefilo, 2023). Využívá k tomu atomických instrukcí, které jsou ale závislé na dané platformě, a tak některé netriviální datové typy kvůli zarovnání v paměti nemusí být tzv. *lock-free*, ale interně pro exkluzivní přístup využívají dodatečného zámku (Toefilo, 2023). V programovacím jazyce C++ je sdílený zámek dostupný od verze C++11 jako *std::atomic*.

6.1.4 Semafor

Semafor funguje na principu inkrementování a dekrementování atomické celočíselné proměnné. Během inicializace semaforu se hodnota atomické proměnné nastaví na počáteční hodnotu, která určuje kolik vláken může přistupovat k danému sdílenému zdroji. Pokud vlákno potřebuje přistupovat ke sdílenému zdroji, dekrementuje semafor o hodnotu *1*. Proces je možné opakovat, dokud hodnota atomické proměnné semaforu nebude rovna nule, v takovém případě vlákno čeká, resp. je uspáno, dokud nebude hodnota alespoň jedna. V opačném případě, pokud vlákno již nepotřebuje přistupovat ke sdílenému zdroji, inkrementuje atomickou proměnnou semaforu o hodnotu *1* (Oracle, 2010). Pro inkrementaci je obecně využívána funkce s názvem *post*, a pro dekrementaci funkce *wait*. V programovacím jazyce C je semafor definován standardem POSIX jako *sem_t*.

6.2 Thread pool

Jedná se o nástroj, který již kombinuje výše zmíněné synchronizační nástroje. Nejzákladnějším typem je tzv. *fixed thread pool*. Ten při své inicializaci vytvoří specifikovaný počet vláken (Oracle,

2024), které na začátku uspí, aby zbytečně nevyužívaly procesorový čas. Všechna vlákna se vytvářejí a spouštějí hned při inicializaci. Důvodem je předcházení systémovým chybám, které mohou vzniknout během vytváření vlákna již za chodu aplikace, a předcházení nedostatku hardwarových či systémových prostředků či jiných prostředků (Oracle, 2024). Vlákna ze začátku nemají žádnou přiřazenou úlohu, ta jim je přiřazena na základě zařazení nějaké dané úlohy do fronty úloh. Po zařazení úlohy do fronty úloh je následně jedno spuštěné vlákno probuzeno. Vlákno odebere z fronty úloh danou úlohu a začne jí vykonávat. Po jejím vykonání vlákno pokračuje v činnosti vykonávání dalších úloh nebo je opět uspáno. Pokud je zařazená nová úloha do fronty úloh a zároveň jsou všechna vlákna aktivní a již vykonávají nějakou úlohu, tak úloha bude vykonána až jedno z vláken dokončí dříve přiřazenou úlohu (Oracle, 2024). Mechanismus tohoto typu je výborný pro škálování výkonu, kde vlákno vykonává úlohu pouze pokud je nějaká ve frontě úloh dostupná, v opačném případě je vlákno uspáno.

7 Systémová služba

Systémová služba, v linuxových systémech označována také jako *daemon,* je proces, který běží na pozadí a vykonává jistou činnost (Kerrisk, 2025a). Systémové služby nejsou interaktivní, to znamená že s nimi nelze interagovat přímo skrze grafické ani textové rozhraní, pouze prostřednictvím např. síťové komunikace. Běh systémových služeb v moderních linuxových systémech zajišťuje služba *systemd*. Systemd je proces, který je spouštěn jako první proces během bootování systému, proto je vždy jeho PID 1. Systemd zajišťuje start systému, tedy spuštění všech potřebných systémových služeb a zajišťuje běh, spouštění a zastavování všech systémových služeb (Kerrisk, 2025b). Jedná se o moderní náhradu dřívějšího *init* procesu. Init proces dokáže spouštět systémové služby pouze sériově, kde každý skript určený pro spuštění dané systémové služby je *shell* skript víceméně bez žádných pravidel. Skripty pro spuštění jednotlivých systémových služeb procesem jsou uloženy v adresáři */etc/init.d* a pro jejich ovládání slouží příkaz *service*. Systemd dokáže spouštět systémové služby paralelně a pomocí konfiguračního souboru, který má jasný formát, lze definovat i závislosti pro službu. Tedy po spuštění jakých služeb se má daná služba spustit během bootovacího procesu. Systemd konfigurační soubory jsou uloženy v adresáři */etc/systemd/system*, jednotlivé skripty mají příponu *.service*. Ovládát tyto služby lze příkazem *systemctl*. Pro zpětnou kompatibilitu systémových služeb určených pro spuštění pod službou *init* avšak jejich spuštění je podporováno i pod službou *systemd.*

Ze výše zmíněných důvodu existují dva typy služeb: *SysV daemon* a *New-Style daemon* (Kerrisk, 2025a). *SysV daemon* je služba vytvořená pro běh pod procesem *init* a *New-Style daemon* pod procesem *systemd*. Pro vytvoření a základního nastavení *New-Style daemon* lze využít následující kroky:

1) Nastavení obslužné funkce pro signál *SIGTERM*, který slouží pro vypnutí služby. Před vypnutím služby je nutné informovat *systemd*, že se služba chystá ukončit svoji činnost, a to voláním funkce *sd_notify*, kde se uvede v příznak *STOPPING=1* (Kerrisk, 2025a). Dodatečně může být signalizován průběh vypínání služby pomocí příznaku *STATUS=…* (např.: *STATUS=Stopped* či *STATUS=Stopping failed*).
2) Nastavení obslužné funkce pro signál *SIGHUP*, který slouží aktualizaci služby (znovunačtení konfiguračních souborů). Před aktualizací služby je nutné pomocí funkce *sd_notify* oznámit *systemd* aktualizaci služby *s* nastaveným příznakem *RELOADING=1* a její aktualizaci s nastaveným příznakem *READY=1* (Kerrisk, 2025a).
3) Po spuštění služby je nutné informovat *systemd* o tom, že se byla služba úspěšně zapnuta. To se provede voláním funkce *sd_notify* s příznakem *READY=1* (Kerrisk, 2025a).

Se samotným *systemd* se váže i *journald*. Jedná se o službu, která je standartně používána pro zachytávání a ukládání výstupů služeb spuštěných pod *systemd*. Jedná se se tedy o logovací systém, který ukládá všechny výstupy strukturovaně a centralizovaně, tedy všechny výstupy lze najít na jednom místě. Výstupy ukládá v adresáři */var/log/journal*. Pro ovládání *journald* slouží příkaz *journalctl*, kterým lze i získat výstupy jednotlivých služeb (Sematext, 2025). Jounald nahradil dřívější *syslog*. Ten je ale stále podporován moderními linuxovými systémy a distribucemi, proto je možné systémové služby nastavit přesměrování

výstupů do *syslog*, které ukládá taktéž centralizovaně v adresáři */var/log/syslog* či */var/log/messages*.

8 GNU Autotools

GNU Autotools je sada nástrojů vyvinutých pro použití v unixových systémech. Byly vyvinuty pro automatizaci překladu a sjednocení překládacího systému balíčků (GNU, 2024). Pro překlad využívá standartního skriptu v unixových systémech pro překlad – *Makefile*. Makefile definuje pravidla, pomocí kterých lze definovat závislosti projektu, které potřebuje pro překlad, jaké soubory potřebuje pro překlad a jaký je výstup překladu. Umožňuje také definovat i jiná pravidla pro provádění jiných operací. Základní myšlenka GNU Autotools je ta, aby se po každé, když je potřeba upravit některé vlastnosti pro překlad balíčku, nemusel ručně upravovat Makefile skript (GNU, 2024). Zároveň také cílí na přenositelnost mezi různými systémy, protože lze definovat soubor se závislostmi daného balíčku (Kenlon, 2019). Utilitami pro vytvoření kompletního překládacího systému balíčku (*GNU Build System)* jsou *autoscan, aclocal, autoheader, automake* a *autoconf*.

8.1 Autoscan

Utilita *autoscan* slouží pro automatizování definice závislostí balíčků. Funguje tak, že prohledá zdrojové kódy v adresáři, odkud je utilita spuštěna, a vytvoří souboru *configure.ac*, který obsahuje závislosti balíčku. Výsledný soubor je možné si upravit, např. změnit minimální požadovanou verzi balíčků, na kterých je daný balíček závislý.

8.2 Aclocal

Utilita *aclocal* vygeneruje soubor *aclocal.m4* s makry, které bude dále potřebné ve vytváření překládacího systému balíčku. Makra generuje na základě souboru *configure.ac*, a samotná makra generuje ze systémových souborů uložených v adresáři */usr/share/aclocal*.

8.3 Autoheader

Utilita *autoheader* slouží pro vygenerování souboru, díky kterému lze následně vygenerovat souboru pro možnost přenositelnosti samotného zdrojového kódu. Vygenerovaný soubor touto utilitou je *config.h.in*, který je vygenerován na základě souboru *configure.ac*.

8.4 Automake

Utilita *automake* zpracovává soubor *Makefile.am* (Kenlon, 2019), který je jediným ručně psaným souborem. Částečně ještě soubor *configure.ac*, který se dá ale vygenerovat utilitou *autoscan* a následně upravit. Soubor *Makefile.am* obsahuje definici toho jaké soubory se mají překládat, jaké jsou výstupní aplikace či knihovny, jaké knihovny se mají k výsledným aplikacím linkovat a podobně. Výstupem této utility je soubor *Makefile.in*.

8.5 Autoconf

Utilita *autoconf* slouží pro vytvoření samotného konfiguračního skriptu *configure* pro daný balíček. Konfigurační skript generuje ze souborů *configure.ac* a *aclocal.m4* (Kenlon, 2019)

a slouží již pro samotnou kontrolu závislostí balíčku na aktuálním systému, které jsou potřeba pro překlad balíčku (Kenlon, 2019). Pomocí skriptu *configure* lze nastavovat různé vlastnosti pro překlad, např. jakým překladačem se má balíček překládat a zda se překládá pro aktuální, či jinou architekturu (*cross-compilation*). Spuštěním skriptu *configure* lze získat již finální soubory *Makefile* a *config.h*. Před samotným spuštěním skriptu se ale ještě musí zavolat příkaz *automake --add-missing*. Soubor *Makefile* obsahuje pravidla pro překlad balíčku s nastavenými vlastnostmi pro překlad pomocí skriptu *configure*. Překlad se provádí příkazem *make* (resp. *make all*, ale toto pravidlo je spuštěno implicitně). Pro instalaci souborů, které jsou výstupem překladu balíčku, do standartních adresářů se použije příkaz *make install* (Kenlon, 2019), naopak pro jejich odinstalaci příkaz *make uninstall*. Pro smazání souborů, které vznikly překladem balíčku lze využít příkaz *make clean*. Soubor *config.h* lze použít v samotném zdrojovém kódu, pro podporu přenositelnosti. Definuje makra s informací o tom, jaké závislosti jsou dostupné na aktuálním systému, kde je balíček překládán, vůči těm, které jsou specifikovány v *configure.ac*. Dle toho se dá podmíněným překladem ve zdrojovém kódu přizpůsobit a např. využít jinou alternativu.

9 Praktická část

Praktická část popisuje realizaci a implementaci samotného webového serveru a důležité poznámky. Obsahuje také srovnání výkonu a paměťových nároků s webovými servery Nginx a Apache.

9.1 Rozsah implementace

Samotný rozsah implementace webového serveru je určen především rozsahem implementace HTTP/1.0 a HTTP/1.1. Server umožňuje přiřazování jednotlivých spojení vláknům *1:1*. Umožňuje šifrovanou komunikaci pomocí šifrovacích algoritmů, které jsou podporovány TLS verze 1.2 a 1.3. Podporuje také kompresi odesílaných dat. Server podporuje *server-driver content negotiation* pouze pro parametr *Accept-Encoding*, který dovoluje se serveru rozhodnout, zda bude využita komprese pro tělo odpovědi. Typ zdroje pro odeslání se jinak vybírá na základě URI (*URI-based*), která je poskytnutá v požadavku od klienta. Na základě URI si server dokáže odvodit typ zdroje, pokud hlavička požadavku neobsahuje atribut *Content-Type*.

9.1.1 HTTP/1.0

Implementace HTTP verze 1.0 podporuje metody *GET, HEAD, POST* a *PUT*. Všechny metody kromě *POST* jsou plně podporovány. Metoda *POST* zpracovává pouze formulářová data, která jsou pro přenos souboru, protože je server statický.

Zpracovávané atributy hlavičky požadavku jsou: *Accept-Encoding*, *If-Modified-Since*, *Content-Length*, *Content-Type, Content-Encoding* a *Content-Disposition*. *Accept-Encoding* podporuje typy komprese *gzip* (resp. *x-gzip*) a *deflate*. Ostatní atributy jsou plně podporovány. Nezpracovávané atributy jsou v požadavku ignorovány.

Podporované atributy v odpovědi jsou: *Server, Content-Type, Content-Length, Date, Vary, Allow, Expires, Last-Modified* a *Location*.

9.1.2 HTTP/1.1

Implementace HTTP verze 1.1 podporuje vše co implementace verze 1.0, kde navíc podporuje metodu *POST*.

Mezi zpracovávané atributy požadavku navíc patří Range, If-*Range, If-Match*, *If-None-Match, If-Unmodified-Since* a *Expect*. Atributy jsou plně podporovány. Nezpracovávané atributy jsou v požadavku ignorovány.

Dalšími podporovanými atributy jsou navíc *Connection, Accept-Ranges, Cache-Control, ETag, Content-Location, Content-Range* a *Transfer-Encoding*. Atribut *ETag* podporuje pouze *strong validators*, tedy *weak validators* nejsou podporovány (např. *ETag: W/“abcd123“).* Atribut *Cache-Control* podporuje položky jako *public, private, no-cache, no-store, must-understand, must-revalidate* a *proxy-revalidate*. Bližší informace k těmto položkám se nachází v kapitole 9.3.3.

9.2 Balíčkovací systém

Projekt obsahuje vlastní jednoduchý balíčkovací systém, aby nemusel projekt spoléhat na nainstalované balíčky na aktuální stanici. Všechny balíčky jsou uloženy jako v archívu *.tar.gz* a umístěny v adresáři *packages*. Balíčky, které obsahuje jsou *httpparser*, *openssl-3.0.16, toml11* a *zlib-1.3.1*.

Balíček *httpparser* slouží pro deserializaci HTTP požadavku. Balíček *openssl-3.0.16* je pro využití šifrovacích algoritmů pro podporu HTTPS. Balíček *toml11* je balíček používání pro zpracování konfiguračních souborů webového serveru, které využívají právě formát TOML (Tom's Obvious Minimal Language). Balíček *zlib-1.3.1* slouží pro podporu komprese těla HTTP požadavků a odpovědí.

Adresář s balíčky obsahuje ještě adresář *patches*, ve kterém jsou uloženy patche pro balíčky. Aktuálně obsahuje adresář pouze jeden patch, a to pro balíček *httppparser* (soubor *httpparser.patch*).

9.3 Zprovoznění projektu

Projekt je nejdříve nutný zprovoznit. Tento proces se skládá z několika kroků, které je nutné provést.

9.3.1 Kompilace

Nejdříve nutné zkompilovat ze samotných zdrojových kódů. Pro překlad projektu byly zvoleny *GNU Autotools*. Projekt obsahuje skript *configure*, který slouží pro vygenerování souboru *Makefile* pro následný překlad projektu. Pro překlad stačí spustit konfigurační skript příkazem *CC=gcc CFLAGS="" CXX=g++ CXXFLAGS="" ./configure*. Skript vygeneruje soubor *Makefile*, který slouží pro překlad projektu. To se provede spuštěním příkazu *make*. Příkaz spustí sekvenci několika vlastně definovaných pravidel, které *GNU Autotools* negenerují.

První krok je vytvoření adresářů, kam budou balíčky rozbaleny z archívu a vytvoření adresáře *staging_dir* v každém z těchto adresářů. Tento adresář bude sloužit jako výstupní adresář pro překlad balíčků. Tento krok je realizován pravidlem *init_packages*.

```
init_packages:
	@mkdir -p $(PACKAGES_DIR)
	@echo "Creating package directories..."
	@for pckg in $(ALL_PACKAGES_NAMES); do \
		mkdir -p "$(PACKAGES_DIR)/$$pckg"; \
		mkdir -p "$(PACKAGES_DIR)/$$pckg/$(STAGING_DIR)"; \
	done
```

Jako druhý krok se provede rozbalení všech balíčku z archívu. To se provede vykonáním pravidla *extract_packages*.

```
extract_packages:
@echo "Extracting package archives..."
@for pckg in $(ALL_PACKAGES_NAMES); do \
tar -xzf "$(PACKAGES_ARCH_DIR)/$$pckg$(PACKAGES_ARCH_SUFFIX)" -C
"$(PACKAGES_DIR)/$$pckg"; \
done
```

Třetí krok je aplikování patchů na jednotlivé balíčky. To je realizováno pravidlem *apply_patches*.

```
apply_patches:
	@echo "Applying patches..."
	@for patch_name in $(ALL_PATCHES_NAMES); do \
		cd "$(PACKAGES_DIR)/$$patch_name/$$patch_name" && cat
"$(PROJECT_ROOT_DIR2)/$(PATCHES_DIR)/$$patch_name.patch" | patch -p1; \
	done
```

Dalším krokem je již překlad samotných balíčků. Z balíčků se překládá a vytváří knihovna pouze z balíčku *openssl-3.0.16* a *zlib-1.3.1*. Jejich překlad je zajištěn pravidlem *openssl* a *zlib*.

```
zlib:
	@echo "Building zlib..."
	@cd $(ZLIB_PACKAGE_BASE)/zlib-$(ZLIB_VER) && \
	CC=$(CC) CFLAGS="$(CFLAGS)" \
	./configure --prefix=../$(STAGING_DIR) && \
	$(MAKE) && \
	$(MAKE) install

openssl:
	@echo "Building OpenSSL..."
	@cd $(OPENSSL_PACKAGE_BASE)/openssl-$(OPENSSL_VER) && \
	CC=$(CC) CFLAGS="$(CFLAGS)" \
	./Configure --prefix=$(shell pwd)/$(OPENSSL_PACKAGE_BASE)/$(STAGING_DIR)
\
	no-ssl2 no-comp no-zlib no-shared && \
	$(MAKE) && \
	$(MAKE) install
```

Posledním krokem je překlad samotného projektu, který ale nevyužívá pravidla pro překlad vygenerované pomocí *automake*, ale je využito vlastního pravidla *build*, který umístí objektové a spustitelný soubor, jako výstup překladu, do adresáře *build*.

```
build: $(BUILD_DIR)/WebServerd$(EXEEXT)
$(BUILD_DIR)/%.o: $(PROJECT_SRC_DIR)/%.cpp
	@mkdir -p $(BUILD_DIR)
	$(CXX) $(CXXFLAGS) $(AM_CXXFLAGS) $(AM_CPPFLAGS) -c $< -o $@

$(BUILD_DIR)/WebServerd$(EXEEXT): $(OBJS)
	$(CXX) $(LDFLAGS) $(AM_LDFLAGS) -o $@ $(OBJS) $(LDADD)
```

9.3.2 Instalace

Instalace je realizována pomocí skriptu *install.sh* umístěný v adresáři *resources/scripts*. Instalační skript zajišťuje následující:

- Nakopírování spustitelného souboru *WebServerd* do adresáře */sbin*
- Nakopírování adresáře *resources/defaults* do adresáře */var/WebServerd*
- Nakopírování souborů z adresáře *resources* do adresáře */etc/WebServerd*
- Vytvoření uživatele *wsd*, pod kterým bude služba spuštěna
- Nastavení patřičných práv jednotlivých adresářů, aby k nim měla služba přistup

Instalace se spustí příkazem *sudo make install*. Pravidlo *install* přetěžuje pravidlo, které generuje *automake*.

```
install:
	@echo "Installing..."
	cd $(SCRIPTS_DIR) && ./$(INSTALL_SCRIPT)
```

Je zde definováno take pravidlo *uninstall*, které provede odinstalaci projektu spuštěním skriptu *uninstall.sh* v adresáři *resources/scripts*. Toto lze provést příkazem *sudo make uninstall*. Pravidlo opět přetěžuje pravidlo generováno pomocí *automake*.

9.3.3 Spuštění

Spuštění služby se provede příkazem *systemctl start WebServerd*. Vypnutí služby lze provést příkazem *sudo systemctl stop WebServerd*. Pro aktualizaci služby *sudo systemctl reload WebServerd.*

9.4 Konfigurační soubory

Webový server má celkem tři konfigurační soubory: *WebServerd.service*, *WebServerd.conf* a *resources.conf*.

9.4.1 WebServerd.service

Konfigurační skript *WebServerd.service* slouží pro konfiguraci systémové služby pro *systemd*.

```
1   [Unit]
2   Description=HTTP Web Server
3   After=network.target
4
5   [Service]
6   User=wsd
7   ExecStart=/sbin/WebServerd
8   ExecReload=/bin/kill -HUP $MAINPID
9   StandardOutput=journal
10  StandardError=journal
11  #StandardOutput=syslog
12  #StandardError=syslog
13  #SyslogIdentifier=WebServerd
14  Restart=no
15  TimeoutStartSec=10
16  TimeoutStopSec=20
17  Type=notify
18  NotifyAccess=all
19
20
21  [Install]
22  WantedBy=multi-user.target
```

Obr. 8: Konfigurační soubor WebServerd.service

Zdroj: Vlastní zpracování (2025)

Soubor obsahuje definici toho, po jaké službě se má během bootování systému spustit (parametr *After*). Dále obsahuje definici toho, jak má *systemd* zacházet se službou. Parametr *User* definuje, pod jakým uživatelem je služba spuštěna, z čehož se odvíjí přístupová práva služby.

Dále definuje parametry *ExecStart* a *ExecReload*, které určují, jaký spustitelný soubor se má spustit pro spuštění služby a to, že služba podporuje příkaz *systemctl reload* pro aktualizaci služby. Parametry *StandardOutput* a *StandardError* určují kam bude přesměrován standartní výstup. Zde se výstup přesměrovává do *journald*, kde tento výstup lze získat příkazem *journalctl -u WebServerd.* Parametry *TimeoutStastSec* a *TimeoutStopSec* specifikují časový intervalu v sekundách, který definuje prahovou hodnotu toho za jak dlouho se má služba zapnout, resp. vypnout, než bude zapnutí, resp. vypnutí, považováno za neúspěšné. Parametry *Type* a *NotifyAccess* pak definují to, že služba využívá funkce *sd_notify* pro oznámování svého stavu službě *systemd*.

9.4.2 WebServerd.conf

Konfigurační skript *WebServer.conf* slouží pro nastavení parametrů samotného webového serveru. Skript využívá formátu TOML a obsahuje následující parametry:

- Parametr **server_name** specifikuje název serveru.
- Parametr **ip_address** specifikuje IP adresu, na které webový server bude spuštěn a přijímat nová spojení.
- Parametr **port** specifikuje port pro nešifrovanou komunikaci.
- Parametr **port_https** specifikuje port pro šifrovanou komunikaci
- Parametr **https_enabled** umožňuje zapnout či vypnout HTTPS, tedy naslouchání na portu specifikovaném v *port_https*. V případě že je povoleno, tak je spuštěno nové vlákno, který *port_https* obsluhuje.
- Parametr **ssl_certificate_rsa** specifikuje cestu k certifikátu používanému pro RSA, pokud je povoleno HTTPS.
- Parametr **private_key_rsa** specifikuje cestu k privátnímu klíči používanému pro RSA, pokud je povoleno HTTPS.
- Parametr **ssl_certificate_ecdsa** specifikuje cestu k certifikátu používanému pro ECDSA (Elliptic Curve Digital Signature Algorithm), pokud je povoleno HTTPS.
- Parametr **private_key_ecdsa** specifikuje cestu k privátnímu klíči používanému pro ECDSA, pokud je povoleno HTTPS.
- Parametr **cipher_suites** specifikuje seznam šifrovacích algoritmů povolených a podporovaných webovým serverem. Webový server dokáže podporovat šifrovací algoritmy podporované TLS verzí 1.2 a 1.3.
- Parametr **client_threads** specifikuje počet vláken spuštěných pro klienty
- Parametr **file_chunk_size** specifikuje velikost určenou v bytech, po jak velkých částech se budou mapovat části souboru do operační paměti. Velikost by měla být ideálně násobkem stránky, jinak bude velikost zarovnána na nejbližší větší násobek.
- Parametr **max_header_size** specifikuje maximální velikost hlavičky HTTP požadavku v bytech.
- Parametr **client_body_buffer_size** specifikuje v bytech velikost úložiště operační paměti, do které bude uloženo tělo HTTP požadavku. Pokud bude tato velikost přesažena, tělo bude uloženo v dočasném souboru v adresáři */var/WebServerd/.tmpfiles*.

- Parametr **client_max_body_size** specifikuje maximální velikost těla HTTP požadavku v bytech.
  Podrobnější informace a popis jednotlivých parametrů se nachází v samotném konfiguračním souboru.
- Parametr **prefer_content_encoding** specifikuje, zda pokud hlavička HTTP požadavku obsahuje *Accept-Encoding*, tak bude automaticky použita komprese pro tělo každé HTTP odpovědi. Podporované algoritmy pro kompresi jsou *gzip* a *deflate*.

9.4.3 resources.conf

Konfigurační soubor *resources.conf* slouží pro nastavení zdrojů, ke kterým má webový server přístup. Jedná se tedy o všechny zdroje (soubory) umístěné v adresáři */var/WebServerd*. Konfigurační soubor využívá opět formátu TOML.

Nastavení parametrů pro jednotlivé zdroje se nemusí provádět pro všechny soubory, ke kterým má server přístup, ale pouze pro ty, pro které existuje potřeba je explicitně definovat. Jednotlivé zdroje se definují tak, že se pro ně zvolí unikátní název jako identifikátor daného zdroje jako *[unikatni_nazev_zdroje]*. Následně jsou definovány jednotlivé parametry samotného zdroje. Pro to, aby webový server dané parametry definovaného zdroje zpracoval, je nutné tento název přidat do seznamu *all_resources_name*. Níže je uvedený příklad toho, jak to je správně zapsané:

```
all_resources_name = [
'index_html'
]

[index_html]
resource_path = 'index.html'
methods_allowed = ['GET', 'HEAD', 'OPTIONS', 'PUT', 'POST', 'DELETE']
accept_ranges = 'bytes'
expires = 3600
cache_type = 'public'
```

Pro každý zdroj je možné definovat tyto parametry:

- Parametr **resource_path** specifikuje cestu k souboru v adresáři */var/WebServerd* a cesta je specifikována vždy bez lomítka na začátku.
- Parametr **methods_allowed** specifikuje jaké HTTP metody lze použít k přístupu k danému zdroji. Povolené hodnoty jsou pouze *GET, HEAD, OPTIONS, PUT, POST* a *DELETE*.
- Parametr **accept_ranges** specifikuje, zda se ke zdroji může přistupovat po částech (hodnota *bytes*), nebo pouze jen k celému zdroji (hodnota *none*).
- Parametr **expires** specifikuje čas v sekundách, kolik může být HTTP odpověď ze serveru cachována (nastavení parametru hlavičky *Expires* v HTTP odpovědi). Pokud je hodnota nastavena na *0*, tak je cáchování odpovědi vypnuto. Pokud je nastavena na *-1*, je parametr hlavičky http odpovědi nastaven na hodnotu *Thu, 01 Jan 1970 00:00:01 GMT*.
- Parametr **cache-type** specifikuje, jak má být daný zdroj, který je součástí HTTP odpovědi, cachovaný. Hodnota *no-store* lze použít, pokud je hodnota parametru *expires* nastavena

na *0* a určuje, že odpověd nemá být cachována. Hodnota *no-cache* je nastavena, pokud hodnota parametru *expires* je nastavena na *-1* a určuje, že klient má požádat pokaždé o aktuální verzi zdroje, vždy když jej chce použít. Hodnota *public* je nastavena, pokud je hodnota parametru *expires* větší než *0* a určuje, že odpověď může být cachována ve sdílené cache (např. proxy server) a zároveň i v privátní cache. Hodnota *private* je nastavena, pokud je hodnota parametru *expires* větší než *0*, a určuje že odpověď má být cachována pouze v privátní cache (např. ve webovém prohlížeči).

Pro zdroje, které nejsou explicitně definovány v tomto konfiguračním souboru, je hodnota jejich parametrů nastavena následovně: *resource_path* je nastavena na URI požadavku, *methods_allowed* je nastaven na všechny metody podporované webovým serverem, *accept_ranges* je nastaven na hodnotu *bytes*, *expires* na hodnotu *3600* (1 hodina), a *cache_type* na hodnotu *public.* Server si neudržuje v operační paměti parametry všech zdrojů v adresáři */var/WebServed*, ale pouze pro ty, které jsou definované v tomto konfiguračním souboru. Pro ty zdroje, co zde nejsou definovány, se udržují parametry v operační paměti maximálně pro *100* zdrojů, 101. zdroj se uloží namísto zdroje, který je nejstarší.

Podrobnější informace a popis parametrů se nachází v samotném konfiguračním souboru.

9.5 Zdrojový kód

V této části jsou popsány jednotlivé komponenty zdrojového kódu projektu, jakož jsou třídy, jmenné prostory a ostatní soubory. Obsahují popis toho, co jednotlivé komponenty zajišťují a jaký mají účel. Zároveň obsahují kód nejdůležitějších částí s popisem toho, jak fungují a co vykonávají. Zdrojový kód byl psán tak, aby jej bylo možné přeložit i pro starší verze C++, jakož jsou verze C++11 a C++14, pro podporu přenositelnosti mezi platformami, které nemají podporu překladače pro novější verze C++. Z tohoto důvodu nejsou některé moderní koncepty standardní knihovny C++ využívány, jakož je např. *std::shared_mutex*.

9.5.1 Soubor Logger.hpp

Soubor *Logger.hpp* obsahuje makra, které obalují pouze funkci *fprintf* pro standardní výstup.

```
#define LOG_INFO(args...) { fprintf(stdout, args); fprintf(stdout, "\n"); }
#define LOG_ERR(args...) { fprintf(stderr, args); fprintf(stderr, "\n"); }
#ifdef DBG
#define LOG_DBG(args...) { fprintf(stderr, args); fprintf(stderr, "\n"); }
#else
#define LOG_DBG(args...) { }
#endif
```

9.5.2 Soubor Globals.hpp

Soubor *Globals.hpp* obsahuje definici konstant, které jsou v rámci celého webovém serveru využívány. Obsahuje definici cest k adresáři s konfiguračními soubory, k adresáři se zdroji serveru atd.

9.5.3 Soubor Codec.hpp

Soubor *Codec.hpp* implementuje jmenný prostor (namespace) *Codec*. Implementuje funkce *compress_data* a *decompress_data*, pomocí kterých lze kompresovat a dekompresovat data. Funkce podporují dva typy komprese či dekomprese: *deflate* a *gzip*. Oba typy komprese využívají stejné funkce *inflate* (pro kompresi) a *deflate* (pre dekompresi), které jsou funkcemi knihovny *zlib*. Funkce využívají kombinace algoritmů *LZ77* a *Huffman encoding* (Bařina, 2019, s. 35). Jediný rozdíl mezi typem komprese *deflate* a *gzip* je ten, že velikost *sliding window* používané pro kompresi lze pro *deflate* nastavit pouze do velikosti 32 kb, *gzip* ale podporuje i větší velikost okna (OpenMV, 2025).

```
bool Codec::compress_data(std::string& compressed_data, const void* data,
const size_t data_size, const bool gzip)
{
    z_stream zs = {0};

    int windowBits = 15;
    if (gzip) {
        windowBits += 16;
    }

    if (deflateInit2(&zs, Z_DEFAULT_COMPRESSION, Z_DEFLATED,
                     windowBits, 8, Z_DEFAULT_STRATEGY) != Z_OK)
    {
        return false;
    }

    zs.next_in = reinterpret_cast<Bytef*>(const_cast<void*>(data));
    zs.avail_in = static_cast<uInt>(data_size);

    int ret;
    char buffer[BUFFER_SIZE];


do
    {
        zs.next_out = reinterpret_cast<Bytef*>(buffer);
        zs.avail_out = sizeof(buffer);

        ret = deflate(&zs, Z_FINISH);
        if (compressed_data.size() < zs.total_out)
        {
            compressed_data.append(buffer, zs.total_out -
compressed_data.size());
        }
    } while (ret == Z_OK);

    deflateEnd(&zs);
    if (ret != Z_STREAM_END) {
        return false;
    }

    return true;
}
```

9.5.4 Třída SslConfig.hpp

Třída *SslConfig.hpp* zajišťuje načtení SSL certifikátů a privátních klíčů používaných pro šifrování komunikace. Webový server podporuje pro autentizaci dva typy klíčů a certifikátů dva typy: *RSA* a *ECDSA*. Samotné načtení zajišťuje metoda *set*. Nejdůležitější metodou je privátní metoda

configureContext, která zajišťuje nastavení šifrovacích algoritmů (*cipher suites*), nastavení minimální verze protokolu TLS, a nastavení jednotlivých certifikátů a privátních klíčů.

```
bool SslConfig::configureContext()
{
    // Cipher suites
    std::string cipher_suites;
    for (const std::string& cs : Config::params().cipher_suites) {
        cipher_suites += cs;
        cipher_suites += ":";
    }
    cipher_suites.erase(cipher_suites.size()-1);

    // RSA
    if (SSL_CTX_use_certificate_chain_file(ctx_,
Config::params().ssl_certificate_rsa.c_str()) <= 0) {
        LOG_ERR("Failed to load RSA certificate");
        return false;
    }
    if (SSL_CTX_use_PrivateKey_file(ctx_,
Config::params().private_key_rsa.c_str(), SSL_FILETYPE_PEM) <= 0) {
        LOG_ERR("Failed to load RSA private key");
        return false;
    }

    // ECDSA
    if (SSL_CTX_use_certificate_chain_file(ctx_,
Config::params().ssl_certificate_ecdsa.c_str()) <= 0) {
        LOG_ERR("Failed to load ECDSA certificate");
        return false;
    }
    if (SSL_CTX_use_PrivateKey_file(ctx_,
Config::params().private_key_ecdsa.c_str(), SSL_FILETYPE_PEM) <= 0) {
        LOG_ERR("Failed to load ECDSA private key");
        return false;
    }

    SSL_CTX_set_ciphersuites(ctx_, cipher_suites.c_str());
    SSL_CTX_set_min_proto_version(ctx_, TLS1_2_VERSION);

    return true;
}
```

9.5.5 Soubor HttpGlobal.hpp a HttpGlobal.cpp

Soubor *HttpGlobal.hpp* obsahuje definice konstant používaných v třídách implementující funkcionalitu pro podporu HTTP protokolu. Obsahuje definice podporovaných formátů zdrojů (souborů), podporované HTTP metody, definici statusových kódů apod. Obsahuje také podpůrné funkce, např. funkce *httpContentType* pro převod přípony zdroje na *MIME* (Multipurpose Internet Mail Extension) typ a naopak (funkce *httpContentTypeSuffix*). Obsahuje také funkci *httpParseRequest*, která zajišťuje deserializaci HTTP požadavku za pomocí volání funkce balíčku *httpparser*.

9.5.6 Soubor HttpGlobal2.hpp

Soubor *HttpGlobal2.hpp* obsahuje pouze definici toho, jaký datový typ se používá pro *ETag* (Entity Tag). Definice je oddělena od souboru *HttpGlobal.hpp* z důvodu toho, že je využívána pouze v několika souborech, které nepotřebují všechny tyto definice, proto je není potřeba všechny zahrnovat do příslušných souborů.

9.5.7 Třída WebServerError

Třída *WebServerError* je třídou, která implementuje vlastní výjimku. Je realizována jako třída, která využívá jako rodičovskou třídu *std::exception*, což je třída standardní C++ knihovny.

9.5.8 Třída HttpPacketBase

Třída *HttpPacketBase* je pouze abstraktní třídou, která slouží zejména pro třídu *Http*, která je také abstraktní, a vytváří nezávislé rozhraní na verzi HTTP protokolu pro ostatní třídy, které z ní dědí. Z tohoto důvodu existuje i tato třída.

9.5.9 Třída HttpPacket

Třída *HttpPacket* přímo dědí ze třídy *HttpPacketBase*. Třída je využívána HTTP verzí 1.0, tak 1.1., a implementuje vytváření HTTP odpovědi tím, že převádí strukturovaná data do textového formátu. Zajišťuje tedy jejich serializaci. Na kódu níže je příklad a vytvoření atributu *Accept-Ranges*.

```
void HttpPacket::Header::acceptRanges(const char* ranges)
{
	if (http_version_ != HttpVersion::HTTP_1_1) {
		return;
	}
	data_ << "Accept-Ranges: " << ranges << HEADERS_ENDLINE;
}
```

Třída obsahuje dvě vnořené třídy: *Header* a *Body*. Vnořená třída *Header* slouží pro vytvoření hlavičky odpovědi. Třída *Body* zase pro přidání těla do odpovědi. Pokud se posílá soubor, neukládá data souboru, ale cestu k tomuto souboru, která je následně použita pro jeho odeslání po částech. Toto zajišťuje metoda *addFile*.

```
bool HttpPacket::Body::addFile(const std::string& rel_path, const uint64_t
file_size)
{
	std::string fpath = std::string(RESOURCES_DIR) + "/" + rel_path;
	data_.is_file_ = true;
	data_.data_ = std::move(fpath);
	data_.content_length_ = file_size;

	return true;
}
```

9.5.10 Třída HttpPacketBuilderBase

Třída *HttpPacketBuilderBase* vychází ze stejné myšlenky jako třída *HttpPacketBase*, tedy poskytnout abstraktní třídu pro třídu *Http*, která není závislá na verzi HTTP.

9.5.11 Třída HttpPacketBuilder

Třída *HttpPacketBuilder* dědí přímo ze třídy *HttpPacketBuilderBase*. Jedná se o obalovací třídu nad třídou *HttpPacket*, která usnadňuje vytváření HTTP odpovědi. Obsahuje metodu *createCommonHeaders*, která do odpovědi přidá společné atributy zasílané v hlavičce téměř

každé odpovědi. Přidá do odpovědi i tělo, pokud to je požadováno (resp. je do metody předáno v parametru).

```
void HttpPacketBuilder::createCommonHeaders(const Config::RParams* rparam,
const HttpStatusCode status_code, const bool status_code_page, const bool
keep_alive)
{
    HttpPacket::Header& pheader = packet_.header();

    if (pheader.httpVer() == HttpVersion::UNSUPPORTED) {
        throw WebServerError("Failed to create HTTP response (HTTP version not
supported)");
    }

    // Pridavam jen pokud odesilam nejaky resource (pripadne i metoda HEAD)
    if (rparam) {
        packet_.body().addFile(rparam->resource_path, rparam->resource_size);
    }

    pheader.statusLine(pheader.httpVer(), status_code);
    pheader.server(Config::params().web_server_name);

    // Date
    time_t cas = time(nullptr);
    struct tm* gmt = gmtime(&cas);
    char datum[100];
    strftime(datum, sizeof(datum), HTTP_DATE_FORMAT, gmt);
    pheader.date(datum);

    // Pridavam jen pokud odesilam nejaky resource (pripadne i metoda HEAD)
    if (rparam)
    {
        // Content-Type
        std::string file_suffix;
        if (!getFileSuffix(rparam->resource_path, file_suffix)) {
            throw WebServerError("Failed to get file suffix");
        }
        const HttpContentTypeS* content_type = httpContentType(file_suffix);
        if (content_type == nullptr) {
            throw WebServerError("Invalid content type");
        }
        pheader.contentType(content_type->content_type_label);

        // Content-Length
        pheader.contentLength(packet_.body().data().content_length_);

        if (pheader.httpVer() == HttpVersion::HTTP_1_1)
        {
            if (keep_alive) { pheader.connection("keep-alive"); }
            else { pheader.connection("close"); }
        }

        // Pokud je to error page tak dalsi atributy generovat nebudu
        if (status_code_page)
        {
            if (end_headers_) { pheader.end(); }
            return;
        }

        // Vary
        pheader.vary();

        // Last-Modified
        time_t mod_cas = rparam->last_modified;
        gmt = gmtime(&mod_cas);
        strftime(datum, sizeof(datum), HTTP_DATE_FORMAT, gmt);
        pheader.lastModified(datum);
```

```
        // Expires
        if (rparam->expires == -1) {
            pheader.expires("Thu, 01 Jan 1970 00:00:01 GMT");
        }
        else if (rparam->expires > 0)
        {
            time_t exp = cas + rparam->expires;
            gmt = gmtime(&exp);
            strftime(datum, sizeof(datum), HTTP_DATE_FORMAT, gmt);
            pheader.expires(datum);
        }

        // Cache-Control
        pheader.cacheControl(rparam->cache_type, rparam->expires);

        // Header fields jen pro HTTP/1.1
        if (pheader.httpVer() == HttpVersion::HTTP_1_1)
        {
            // ETag
            pheader.etag(rparam->etag);

            // Accept-Ranges
            pheader.acceptRanges(rparam->accept_ranges.c_str());
        }
    }

    if (end_headers_) { pheader.end(); }
}
```

Třída obsahuje také metody, které začínají názvem *build* (např. *buildNotFound* či *buildNoContent*), které slouží pro sestavení odpovědi se statusovým kódem jiným, než je kód *200 OK*. Níže je zdrojový kód některých těchto metod.

```
void HttpPacketBuilder::buildNotFound()
{
    Config::RParams* rparam;
    try
    {
        rparam = getStatPageRParams(NOT_FOUND_WEB_PAGE);
        createCommonHeaders(rparam, HttpStatusCode::NOT_FOUND, true);
        rparam->unlock();
    }
    catch (const WebServerError& exc) {
        rparam->unlock();
        throw WebServerError(exc.what());
    }
    catch (const std::exception& exc) {
        rparam->unlock();
        throw std::runtime_error(exc.what());
    }
}
void HttpPacketBuilder::buildNoContent()
{
    createCommonHeaders(HttpStatusCode::NO_CONTENT, true);
}

void HttpPacketBuilder::buildNoContent(const Config::RParams* rparam)
{
    if (!rparam) {
        throw WebServerError("Failed to build packet (null resource
parameters)");
    }

    HttpPacket::Header& pheader = packet_.header();
```

```
    setEndHeaders(false);
    createCommonHeaders(HttpStatusCode::NO_CONTENT, true);
    pheader.contentLocation(rparam->resource_path);

    // Last-Modified
    char datum[100];
    time_t mod_cas = rparam->last_modified;
    struct tm* gmt = gmtime(&mod_cas);
    strftime(datum, sizeof(datum), HTTP_DATE_FORMAT, gmt);
    pheader.lastModified(datum);

    if (pheader.httpVer() == HttpVersion::HTTP_1_1) {
        pheader.etag(rparam->etag);
    }
    pheader.end();
}
```

9.5.12 Třída Config

Třída *Config* slouží pro načítání konfiguračních souborů a uložení informací z nich načtených pro použití v rámci webového serveru. Třída je realizována pomocí návrhového vzoru *singleton*.

Načtení a zpracování konfiguračního souboru *WebServerd.conf* zajišťuje metoda *buildParams*. Ta ukládá jednotlivé parametry do členské proměnné *params_*, která je datového typu *Params*. *Params* je vnořenou třídou třídy *Config*.

```
bool Config::buildParams()
{
	try
	{
		const toml::value input = toml::parse(CONFIG_FILE_FPATH);

		getValue(params_.web_server_name, SERVER_NAME, input);
		getValue(params_.ip_address, IP_ADDRESS, input);
		getValue(params_.port, PORT, input);
		getValue(params_.port_https, PORT_HTTPS, input);
		getValue(params_.https_enabled, HTTPS_ENABLED, input);
		getValue(params_.ssl_certificate_rsa, SSL_CERTIFICATE_RSA,
input);
		getValue(params_.private_key_rsa, PRIVATE_KEY_RSA, input);
		getValue(params_.ssl_certificate_ecdsa, SSL_CERTIFICATE_ECDSA,
input);
		getValue(params_.private_key_ecdsa, PRIVATE_KEY_ECDSA, input);
		getValue(params_.cipher_suites, CIPHER_SUITES, input);

		getValue(params_.client_threads, CLIENT_THREADS, input);
		getValue(params_.file_chunk_size, FILE_CHUNK_SIZE, input);
		getValue(params_.max_header_size, MAX_HEADER_SIZE, input);
		getValue(params_.client_body_buffer_size,
CLIENT_BODY_BUFFER_SIZE, input);
		getValue(params_.client_max_body_size, CLIENT_MAX_BODY_SIZE,
input);
		getValue(params_.prefer_content_encoding,
PREFER_CONTENT_ENCODING, input);
	}

	catch (const toml::syntax_error& e) {
		LOG_ERR("Invalid syntax of configuration file");
		return false;
	}

	catch (const WebServerError& e) {
		LOG_ERR(e.what());
		return false;
	}
```

```
	catch (const std::exception& e) {
		LOG_ERR("Error while configuration file parsing (error: %s)",
e.what());
		return false;
	}

	return true;
}
```

Načtení a zpracování konfiguračního souboru *resources.conf* je realizováno metodou *buildRParams*. Funguje obdobně s tím rozdílem, že se načítají jednotlivé parametry v cyklu pro každý definovaný zdroj, a že se data načítají do datové kolekce *rparams_*.

Jednotlivé parametry načtené z obou konfiguračních souborů se ukládají do slovníku (*std::unordred_map*), kde jako klíč je použitý parametr *resource_path*, kterým se dá jednoduše dostat k uloženému zdroji za použití URI v HTTP požadavku. Hodnota asociovaná ke každému klíči je reprezentována instancí datového typu *RParams*. *RParams* je vnořenou třídou třídy *Config*, a popisuje parametry jednotlivých zdrojů.

Třída *Config* také implementuje rozhraní pro přístup k parametrům jednotlivých zdrojů. To zajišťují metody *rparams* a *orparams*. Metoda *rparams*, slouží pro získání parametrů jednotlivých zdrojů definovaných v *resources.conf*. Naopak *orparams* je metodou, která zajišťuje přístup a dynamické přidávání a odebírání parametrů zdrojů, které nejsou definovány v *resources.conf*. Parametry jsou ukládány v *other_rparams_*, která umožňuje uložit až *100* parametrů těchto zdrojů. Pro uložení 101. parametrů zdroje se uloží namísto parametrů zdroje, který je nejstarší. To je zajištěno tím, že je při každém volání metody *orparam*s aktualizován atribut *last_resource_access* jednotlivých zdrojů. Aktualizaci atributu *last_resource_access* zajišťuje metoda *updateLastAccess*.

```
Config::RParams& Config::rparams(const std::string& resource, const bool
resource_lock_shared)
{
	Config::RParams& rparam = obj_.rparams_.at((!resource.empty() &&
resource.at(0) == '/') ? resource.substr(1) : resource);
	rparam.lock(resource_lock_shared);
	return rparam;
}

Config::RParams* Config::orparams(const std::string& resource, const bool
resource_lock_shared)
{
	try
	{
		std::lock_guard<std::mutex> lock(obj_.orparams_mutex_);

		static const std::vector<std::string> default_allowed_methods =
			{ "GET", "HEAD", "POST", "PUT", "DELETE", "OPTIONS" };

		// Nejdrive zkusit najit resource
		const std::string rsrc_path = ((!resource.empty() &&
resource.at(0) == '/') ? resource.substr(1) : resource);
		const auto rsrc_it = obj_.other_rparams_.find(rsrc_path);
		if (rsrc_it != obj_.other_rparams_.end())
		{
			rsrc_it->second.lock(resource_lock_shared);
			return &rsrc_it->second;
		}
```

```
            Config::RParams rparam;
            rparam.resource_path = rsrc_path;
            rparam.methods_allowed = default_allowed_methods;
            rparam.accept_ranges = "bytes";
            rparam.expires = 3600;  // 1 hodina v sekundach
            rparam.cache_type = "public";
            // Zatim nastavuji defaultni hodnoty, protoze resource nemusi byt
jeste vytvoreny
            rparam.resource_size = 0;
            rparam.last_modified = -1;
            rparam.etag = "";
            rparam.last_resource_access = time(NULL);

            if (obj_.other_rparams_.size() == MAX_OTHER_RPARAMS)
            {
                  const auto rparam_it =
std::min_element(obj_.other_rparams_.cbegin(), obj_.other_rparams_.cend(),
                  [](const std::pair<const std::string, Config::RParams>& r1,
                     const std::pair<const std::string, Config::RParams>& r2)
                  {
                        return r1.second.last_resource_access <
r2.second.last_resource_access;
                  });

                  if (rsrc_path != rparam_it->first) {
                        obj_.other_rparams_.erase(rparam_it);
                  }
            }

            obj_.other_rparams_[rsrc_path] = std::move(rparam);
            Config::RParams* rp = &obj_.other_rparams_[rsrc_path];
            rp->lock(resource_lock_shared);
            return rp;
      }

      catch (const std::exception& exc)
      {
            LOG_ERR("Failed to handle new resource");
      }

      return nullptr;
}
```

Jak je z kódu výše vidět, přístup k parametrům je zajištěn v uzamčeném kontextu. To zajišťuje metoda *lock* vnořené třídy *RParams*. Jednotlivé zdroje využívají celkem dva zámky. Prvním zámkem je zámek získaný pomocí systémového volání *flock*, který uzamkne přístup k souboru jako takovému. Druhým zámkem je *access_lock*, který souží pro synchronizaci přístupu k parametrům zdroje, uložených v *rparams_* a *orparams_*. Oba zámky využívají dvou módu přístupu: sdílený a exkluzivní. Sdílený mód je využit tehdy, když daný HTTP požadavek pro přístup ke zdroji má HTTP metodu *GET, HEAD* či *OPTIONS*. Exkluzivní mód pak pro metodu *PUT, POST* či *DELETE* (obecně *state-changing* metody). Zamykání probíhá vždy při pokusu o získání parametrů daného zdroje. Uvolnění zámků poté zajišťuje metoda *unlock*, která ale musí být volána explicitně.

```
void Config::RParams::lock(const bool resource_lock_shared)
{
      if (resource_lock_shared)
      {
            if (pthread_rwlock_rdlock(&access_lock) != 0) {
                  goto err;
            }
```

```
		if (resource_fd != -1)
		{
			if (flock(resource_fd, LOCK_SH) == -1) {
				goto err;
			}
		}
	}
	else
	{
		if (pthread_rwlock_wrlock(&access_lock) != 0) {
			goto err;
		}

		if (resource_fd != -1)
		{
			if (flock(resource_fd, LOCK_EX) == -1) {
				goto err;
			}
		}
	}

	access_counter++;
	return;

err:
	throw WebServerError(
		std::string("Failed to lock resource (err: ") + strerror(errno) +
")");
}
```

Vnořená třída *RParams* také implementuje metodu *update*, která je na daným zdrojem volána po zpracování a vykonání každého HTTP požadavku. Zajišťuje aktualizaci parametrů zdroje a volá metodu *generateETag*, pro vytvoření *ETagu*, který se posílá v téměř každé HTTP odpovědi (od verze HTTP/1.1). *ETag* slouží jako identifikátor aktuální verze zdroje a umožňuje dělat podmíněné HTTP požadavky a tím šetřit šířku pásma (Mozilla, 2024e). *ETag* je možné reprezentovat jako časovou značku posledního času modifikace zdroje (souboru) s dostatečným rozlišením (Mozilla, 2024e). Tento způsob reprezentace byl zvolen pro webový server, kde rozlišení časové značky je v milisekundách.

```
std::string Config::RParams::generateETag(const int64_t sec, const int64_t
nsec)
{
	const int64_t e = sec*1000 + nsec/1000000;
	return (std::string("\"") + std::to_string(e) + "\"");
}
```

9.5.13 Třída ThreadPool

Třída *ThreadPool* slouží pro spravování vláken, které jsou využívána pro připojené klienty. Třída využívá pro synchronizaci vláken semaforu a atomické členské proměnné *run_*, kterou se synchronizuje celá instance třídy.

Pro vytvoření a spuštění vláken slouží metoda *start*. Metoda inicializuje semafor na hodnotu *0*, čímž se ze startu zajistí, že žádné vytvořené vlákno nebude vykonávat žádnou činnost.

```
bool ThreadPool::start()
{
	try
	{
		if (!run_)
		{
			if (sem_init(&semaphore_, 0, 0) == -1)
			{
				LOG_ERR("Failed to init thread pool");
				return false;
			}

			run_ = true;
			for (auto& thread : threads_) {
				thread = std::thread(&ThreadPool::worker, this);
			}

			waitForRun();
			return true;
		}
	}

	catch (const std::exception& e)
	{
		LOG_ERR("Failed to spawn threads");
		stop();
	}

	return false;
}
```

Pro samotné zastavení vytvořených vláken slouží metoda *stop*, která nejdříve nastaví hodnotu *run_* na *false*, čímž se signalizuje vláknům po přečtení hodnoty této proměnné, že mají zastavit svou činnost. Následně se pomocí *sem_post* probudí všechna uspaná vlákna, které si právě přečtou hodnotu proměnné *run_*, na základě které ukončí svou činnost.

```
bool ThreadPool::stop(const bool force)
{
	try
	{
		bool ret = true;
		std::lock_guard<std::mutex> lock(mutex_);

		if (run_)
		{
			if (!force) {
				waitForTasks();
			}

			run_ = false;

			for (size_t i = 0; i < threads_.size(); ++i)
			{
				if (sem_post(&semaphore_) == -1)
				{
					ret = false;
				}
			}

			for (auto& thread : threads_)
			{
				if (thread.joinable())
				{
					thread.join();
				}
```

```
                    }

                    sem_destroy(&semaphore_);
                    return ret;
             }
      }

      catch (const std::exception& e) {
             LOG_ERR("Failed to stop thread pool (error: %s)", e.what());
      }

      return false;
}
```

Zařazení úlohy do fronty úloh zajišťuje metoda *queueTask*. Jako parametr přijímá danou úlohu, kterou následně zařadí do fronty úloh. Před samotným zařazením je nutné ale nejdříve získat zámek, protože k frontě úloh zároveň přistupují i spuštěná vlákna, která jednotlivé úlohy z fronty postupně vykonávají.

```
bool ThreadPool::queueTask(const Task& task)
{
      try
      {
             std::lock_guard<std::mutex> lock(mutex_);
             if (run_)
             {
                    task_queue_.push_back(task);
                    if (sem_post(&semaphore_) == -1)
                    {
                           task_queue_.pop_back();
                           return false;
                    }

                    return true;
             }
      }

      catch (const std::exception& e) {
             LOG_ERR("Failed to queue task (error: %s)", e.what());
      }

      return false;
}
```

Samotná metoda, kterou každé spuštěné vlákno vykonává, je metoda *worker*. Metoda je spuštěnými vlákny vykonávána, dokud je má výše zmíněná proměnná *run_* hodnotu *true*. Proto, aby vlákno neplýtvalo procesorový čas, je uvedeno do spánku pomocí *sem_wait*, který čeká, dokud není probuzeno pomocí *sem_post*. Po probuzení vlákna zkontroluje, zda má vykonávat nějakou úlohu, nebo zda ukončit svoji činnost. V případě že má vykonat po probuzení nějakou úlohu, tak si jí v zamčeném kontextu vybere z fronty, zámek odemkne a vykoná danou úlohu.

```
void ThreadPool::worker()
{
      Task task;
      running_threads_ += 1;

      while (run_)
      {
             if (sem_wait(&semaphore_) == -1) {
                    continue;
```

```
			}

			// Kontrola po probuzeni vlakna
			if (!run_) {
				break;
			}

			mutex_.lock();
			if (!task_queue_.empty())
			{
				task = std::move(task_queue_.front());
				task_queue_.pop_front();
				mutex_.unlock();

				if (task)
				{
					busy_threads_ += 1;
					task();
					busy_threads_ -= 1;
				}
			}

			else {
				mutex_.unlock();
			}
		}

		running_threads_ -= 1;
}
```

9.5.14 Třída TcpServer

Třída *TcpServer* implementuje rozhraní, které zajišťuje síťovou komunikaci a práci se sockety. Jedná se o vrstvu přímo nad vrstvou, kterou představuje třída *ThreadPool*, což této třídě umožňuje využívat její služby. Třída obsahuje také vnořenou třídu *Connection*, kterou je reprezentováno spojení s klientem.

Hlavní metodou třídy je metoda *start*. Ta zajišťuje spuštění vláken a vytvoření dvou socketů pro přijímání nových spojení. Jeden socket pro nešifrované spojení, a druhý pro spojení šifrované.

```
bool TcpServer::start()
{
	if (!run_)
	{
		if (!thread_pool_.start())
		{
			LOG_ERR("Failed to spawn threads");
			return false;
		}

		// Vytvoreni socketu pro pripojovani
		if (!initSocket(socket_, server_)) {
			return false;
		}
		if (Config::params().https_enabled)
		{
			if (!initSocket(socket_ssl_, server_ssl_)) {
				return false;
			}
		}

		run_ = true;
		deactivated_ = false;
```

```
			return true;
		}

		return false;
}
```

Pro zastavení serveru slouží metoda *stop*. Před zastavením vláken je nejdříve nutné server deaktivovat (metoda *deactivate*), která zavře sockety vytvořené pro přijímání nových spojení. Ještě důležitější ale je zavřít sockety samotných klientů sekvencí systémových volání *shutdown* a *close*. Systémové volání *shutdown* zajistí uzavření spojení ze strany servery, což vede k probuzení těch vláken, které jsou uspány blokujícím systémovým voláním (např. *recv* pro příjem dat ze socketu). Jejich probuzení umožní kontrolu toho, zda je spojení pořád otevřeno a následně ukončit činnost, kterou vykonává dané vlákno. Aktivní (neuspaná) vlákna jsou ukončena stejným způsobem, tedy kontroly, zda je spojení pořád otevřeno.

```
bool TcpServer::stop()
{
		try
		{
			bool ret = true;
			if (run_)
			{
				if (!deactivated_) {
					ret = deactivate();
				}

				run_ = false;

				for (std::shared_ptr<TcpServer::Connection>& conn :
connections_)
				{
					if (shutdown(conn->socket_, SHUT_RDWR) == -1) {
						ret = false;
					}
					if (close(conn->socket_) == -1) {
						ret = false;
					}
				}
				if (!thread_pool_.stop(true))
				{
					LOG_ERR("Failed to stop TCP server (failed to stop
threads)");
					ret = false;
				}

				return ret;
			}
		}

		catch (const std::exception& e) {
			LOG_ERR("Failed to stop TCP server (error: %s)", e.what());
		}

		return false;
}
```

Pro kontrolu toho, zda je spojení pořád aktivní, slouží metoda *isConnected*. Ta funguje na principu neblokujícího čtení ze socketu používaného pro komunikaci s klientem. Pokud systémové volání *recv* vrátí hodnotu *0*, je spojení uzavřeno. Tato hodnota je ale vrácena pouze v případě, že v bufferu socketu nejsou již žádná data ke čtení. Pokud vrátí *-1* a zároveň

je hodnota *errno* na stavena na příznak, že by *recv* blokoval, tedy čekal na data na příjem, tak je jasné že spojení je otevřeno. V ostatních případech je spojení otevřeno.

```
bool TcpServer::isConnected(const std::shared_ptr<TcpServer::Connection>&
connection) const
{
    if (run_)
    {
              errno = 0;
              uint8_t buffer;
              const int ret = recv(connection->socket_, &buffer,
sizeof(buffer), MSG_PEEK | MSG_DONTWAIT);

              if (ret == 0) {
                     return false;
              }
              else if (ret == -1)
              {
                     if (errno == EAGAIN || errno == EWOULDBLOCK) {
                            return true;
                     }
                     else {
                            return false;
                     }
              }
              else {
                     return true;
              }
       }

    return false;
}
```

Příjem nových spojení zajišťují metoda *acceptConnection* a *acceptConnectionSsl* (šifrované spojení). Každá metody přijímá spojení na jiném portu. Tyto funkce ale nezajistí přiřazení vlákno klientovi. Pro tento účel slouží funkce *handleConnection*. Metoda si uloží dané spojení pro možnost ho následně ukončit v metodě *stop*, a přiřadí mu vlákno. Následné ukončení spojení realizuje metoda *endConnection*, která zavře socket daného spojení.

Pro odesílání dat slouží metoda *sendText*. Metoda odesílá data v smyčce z toho důvodu, že se nemusí odeslat všechna data najednou. Metoda rozlišuje mezi odesíláním dat v šifrované (funkce *SSL_write_ex*) a nešifrované formě (funkce *send*).

```
int TcpServer::sendAll(const TcpServer::Connection& connection, const char*
data, const size_t size)
{
    size_t total = 0;
    size_t bytesleft = size;
    int n;
       int ret;

    while (total < size)
       {
              errno = 0;
              if (connection.ssl_)
              {
                     size_t nn = 0;
                     ret = SSL_write_ex(connection.ssl_, data + total,
bytesleft, &nn);
                     n = static_cast<int>(nn);
              }
              else
              {
```

```
            ret = send(connection.socket_, data + total, bytesleft,
MSG_NOSIGNAL);
                  n = ret;
            }

            // napr.: ECONNRESET (ciste close() nad aktivnim spojenim), EPIPE
(kdyz peer zavola shutdown(sock, SHUT_RD))
            if (ret == -1) {
                  return -1;
            }
            else if (ret == 0) {
                  return 0;
            }

        total += n;
        bytesleft -= n;
    }

      return 1;
}
int TcpServer::sendText(const TcpServer::Connection& connection, const char*
data, const size_t size)
{
      if (run_ && data != nullptr && size > 0)
      {
            int ret = sendAll(connection, data, size);
            return ret;
      }

      return -1;
}
```

Pro příjem slouží metoda *receiveText*, která má dvě verze. První verze zajišťuje příjem přesně specifikovaného počtu bytů. Druhá verze přijímá data, dokud se v ní nenachází specifikovaná sekvence bytů. Obě verze metod rozlišují mezi příjmem dat v šifrované (funkce *SSL_read_*ex) a nešifrované formě (funkce *recv*). Příjem dat je opět ve smyčce, protože jednotlivé funkce nemusí vrátit požadovaný počet bytů k přečtení. Probíhá také kontrola, zda nebylo spojení přerušeno, aby cyklus nebyl „nekonečný“. Níže je kód první verze metody pro příjem předem specifikovaného počtu bytů.

```
int TcpServer::receiveText(const TcpServer::Connection& connection,
std::string& data, const uint64_t bytes_to_recv, const bool peek_data)
{
      const int recv_flag = ((peek_data) ? MSG_PEEK : MSG_WAITALL);
      uint64_t total = 0;
      int n;
      int ret;

      // Pockat az budou nejaka data dostupna
      ret = waitForData(connection);
      if (ret != 1) { return ret; }


      // Nacteni samotnych dat
      while (total != bytes_to_recv)
      {
            errno = 0;
            if (connection.ssl_)
            {
                  size_t nn = 0;
                  if (peek_data) {
                        ret = SSL_peek_ex(connection.ssl_,
const_cast<char*>(data.data()) + total, bytes_to_recv - total, &nn);
```

```
                    }
                    else {
                          ret = SSL_read_ex(connection.ssl_,
const_cast<char*>(data.data()) + total, bytes_to_recv - total, &nn);
                    }
                    n = static_cast<int>(nn);
             }
             else
             {
                    ret = recv(connection.socket_,
const_cast<char*>(data.data()) + total, bytes_to_recv - total, recv_flag);
                    n = ret;
             }

             // Kontrola chyby
             if (ret == -1) {
                    return -1;
             }
             // Kontrola zda nebylo preruseno spojeni
             else if (ret == 0) {
                    return 0;
             }

             total += n;
       }

       return 1;
}
```

9.5.15 Třída HTTP

Třída *HTTP* je abstraktní třídou, která slouží jako základní a rodičovskou třídou pro třídy, které z ní dědí, resp. třídy, které implementují konkrétní verzi HTTP protokolu (třídy *Http1_0* a *Http1_1*). Definuje základní rozhraní pomocí veřejných *pure virtual* metod *handleConnection*, *sendResponse* a *receiveRequest*. Obsahuje ale i implementaci metod, které jsou využívány všemi třídami, které z ní dědí. Jedná se o metody, které umožňují pracovat se zdroji, jakož je kontrola přístupu, jejich vytváření, mazání apod. Takovou metodou je např. *deleteResource*.

```
bool Http::deleteResource(const std::string& file_name)
{
    const std::string file_path = std::string(RESOURCES_DIR) + file_name;
    if (remove(file_path.c_str()) == -1) {
        LOG_ERR("Failed to delete resource (resource: %s)",
file_path.c_str());
        return false;
    }

    return true;
}
```

Dalšími metodami, které třída *Http* implementuje, jsou metody pro podporu mechanismu, který je využíván tehdy, kdy klient posílá na server velké soubory, resp. větší než hodnota parametru *client_body_buffer_size* konfiguračního souboru *resources.conf*. Mechanismus funguje na takovém principu, že ukládá data do dočasného souboru vytvořeném v adresáři */var/WebServerd/.tmpfiles*. V tomto adresáři se pro každé spojení vytvoří adresář, který má název jako číslo socketu každého spojení. V něm se až vytváří soubory pro dočasné ukládání přijatých dat, který je přijetí všech dat přesunut do destinace. Jednotlivé soubory jsou pojmenovány dle *stream ID*, ale vzhledem k tomu že implementované verze HTTP/1.0 a 1.1

nepodporují multiplexing, je tato vlastnost implementována pro možnost rozšíření serveru o implementaci HTTP/2.0, který již multiplexing podporuje. Kvůli tomu má vytvořený dočasný soubor vždy název *0*, což je v HTTP/2.0 stream ID rezervováno pro celé spojení.

Níže uvedený kód zajišťuje vytvoření adresáře pro dočasné soubory (metoda *createTempDirectory*) a vytvoření názvu pro dočasný soubor (metoda *generateTempFilePath*).

```
bool Http::createTempDirectory()
{
    char buffer[PATH_MAX];
      snprintf(buffer, sizeof(buffer), TEMPORARY_DIR_NAME_FORMAT,
tcp_connection_->getSocket());
      temp_files_dir_ = buffer;

    // Zkontrolovat zda uz existuje adresar a pripadne vytvorit adresar pro
docasne soubory daneho spojeni
    errno = 0;
    DIR* dir = opendir(temp_files_dir_.c_str());
    if (dir) {
        closedir(dir);
    }
    else if (!dir && errno == ENOENT)
    {
        if (mkdir(temp_files_dir_.c_str(), 0700) == -1) {
            LOG_ERR("Failed to create temporary files directory");
            return false;
        }
    }
    else {
        LOG_ERR("Failed to check if temporary files directory exists (error:
%s)", strerror(errno));
        return false;
    }

    return true;
}

void Http::generateTempFilePath(const Http::StreamId stream_id)
{
      char buffer[PATH_MAX];
      snprintf(buffer, sizeof(buffer), TEMPORARY_FILE_NAME_FORMAT,
tcp_connection_->getSocket(), stream_id);

    Http::TempFile tmp_file;
    tmp_file.stream_id_ = stream_id;
    tmp_file.data_in_temp_file_ = false;
    tmp_file.file_path_ = std::string(buffer);

    std::pair<Http::StreamId, Http::TempFile> item(stream_id, tmp_file);
      temp_files_.insert(item);
}
```

9.5.16 Třída Http1_0

Třída *Http1_0* již zahrnuje implementaci HTTP verze 1.0 a dědí přímo ze třídy *Http*. Implementuje všechny *pure virtual* metody třídy *Http*, tedy implementace už je specifická pro konkrétní verzi protokolu.

Hlavní metodou je metoda *handleConnection*, která se stará o dané spojení. Zajišťuje volání dílčích metod pro příjem požadavku, jeho kontrolu a deserializaci, provedení požadavku, následné odeslání odpovědi a ukončení spojení. Metoda nejdříve provede přípravu pro možnost

vytváření dočasných (temporary) souborů. Následně voláním metody *receiveRequest* čeká, dokud nedorazí nějaký požadavek. Metoda *receiveRequest* se pak také postará o veškeré jeho zpracování. Po přijetí a zpracování požadavku následuje kontrola, vytvoření odpovědi a provedení akcí, které určují jednotlivé zpracovávané atributy hlavičky požadavku (volání metod *requestGetMethod* či *requestHeadMethod*). Provedení akcí se v tomto bodě provádí pouze pro metody *GET*, *HEAD* a *DELETE*, protože pro metody *PUT* a *POST* se provádí akce již metodami, které jsou volané metodou *receiveRequest*. To je z toho důvodu, že pro tyto dvě metody jsou akce pro vykonání pouze příjem a následné uložení odesílaných dat obsažených v těle HTTP požadavku. Následuje odeslání odpovědi, které je označeno návěštím *send_response*. Ta rozlišuje, zda se odesílá odpověď s jiným statusovým kódem, než je *200 OK*. To může znamenat, že během zpracovávání požadavku byla objevena či nastala nějaká chyby. To neplatí pouze pro statusové kódy *201 Created*, *304 Not Modified* apod., které jsou zasílány právě pokud byl vytvořen nový zdroj, nebo se jedná o odpověď na podmíněný požadavek. Nakonec už pouze ukončí spojení voláním metody *endConnection*.

```
void Http1_0::handleConnection()
{
    try
    {
        if (isConnected())
        {
            int ret;

            // Vytvorit adresar pro temporary files
            if (!this->createTempDirectory())
            {
                packet_builder_sp_.buildInternalServerError();
                status_page_ = true;
                goto send_response;
            }

            // Vygenerovat nazev pro temp file
            this->generateTempFilePath(0);
            temp_file_ = &temp_files_.at(0);

            // Prijmout request
            ret = receiveRequest();
            if (ret == 0) {
                goto end_conn;
            }
            else if (ret == -1) {
                goto send_response;
            }

            // Kontrola atributu hlavicky pozadavku
            switch (request_method_)
            {
            case HttpMethod::GET:
                this->requestGetMethod();
                break;
            case HttpMethod::HEAD:
                this->requestHeadMethod();
                break;
            case HttpMethod::DELETE:
                this->requestDeleteMethod();
                break;
            }

    send_response:
            // Poslat odpoved
            if (status_page_)
```

```
            {
                this->sendResponse(packet_builder_sp_.packet());
                if (rparam_)
                {
                    bool remove_orparam = true;
                    HttpStatusCode hsc =
                        packet_builder_sp_.packet().header().statusCode();
                    switch (hsc)
                    {
                        case HttpStatusCode::OK:
                        case HttpStatusCode::CREATED:
                        case HttpStatusCode::NO_CONTENT:
                        case HttpStatusCode::PARTIAL_CONTENT:
                        case HttpStatusCode::NOT_MODIFIED:
                            remove_orparam = false;
                            break;
                    }
                    if (remove_orparam)
                    {
                        Config::orparamsRemove(rparam_);
                        rparam_ = nullptr;
                    }
                }
            }

            else
            {
                if (!packet_builder_.packet().header().hasEnd()) {
                    packet_builder_.packet().header().end();
                }
                this->sendResponse(packet_builder_.packet());
                // Prevence vuci ponechani resourcu v nekonzistentnim stavu
                if (rparam_ && !rparam_->isSet()) {
                    const_cast<Config::RParams*>(rparam_)->update();
                }
            }

            goto end_conn;
        }
    }

    catch (const std::exception& exc)
    {
        LOG_ERR("Error: %s", exc.what());

        if (rparam_) {
            const_cast<Config::RParams*>(rparam_)->unlock();
            rparam_ = nullptr;
        }
        packet_builder_sp_.buildInternalServerError();
        this->sendResponse(packet_builder_sp_.packet());
    }

end_conn:
    try
    {
        // Ukoncit spojeni
        if (rparam_)
        {
            const_cast<Config::RParams*>(rparam_)->unlock();
            rparam_ = nullptr;
        }
        this->endConnection();
    }
    catch (const std::exception& e) {
        LOG_ERR("Failed to end client connection and release its resources");
    }
}
```

Metoda *receiveRequest*, jak již bylo zmíněno výše, provádí příjem požadavku a jeho validaci a zpracování. Metoda se skládá z několika po sobě jdoucích kroků, které je nutné realizovat. Nejdříve je realizován příjem požadavku, a to pomocí volání metody *receiveText* instance třídy *TcpServer*. Následuje deserializace požadavku voláním funkce *httpParseRequest*, která je pouze obaluje volání funkce balíčku *httpparser*, který deserializaci zajišťuje. Následně je volána funkce *httpMethod*, která provádí kontrolu, zda požadavek využívá HTTP metody, která je zároveň podporována serverem a danou verzí protokolu. Metoda *validateResource* dále kontroluje, zda daný zdroj na serveru vůbec existuje a zda má server patřičné oprávnění k němu přistupovat. Nakonec se provede kontrola parametrů daného zdroje, které jsou nastaveny v konfiguračním souboru *resources.conf*, zda požadavek nevyužívá např. pro přístup HTTP metodu, které není povolena. Dále už následuje pouze příjem, zpracování a uložení těla požadavku, což se provádí pouze pro metody *PUT* a *POST*.

```
int Http1_0::receiveRequest()
{
    std::string buffer;
    int ret;

    // Nacteni hlavicky HTTP requestu
    ret = this->tcp_server_->receiveText(this->tcp_connection_, buffer,
        Config::params().max_header_size, HEADERS_END, false);
    ret = receiveRetCheck(ret, -1);
    if (ret != 1) {
        return ret;
    }

    // Prekopirovani hlavicky
    request_data_ = std::move(buffer);

    // Rozparsovani a kontrola HTTP requestu
    httpparser::Request request;
    if (!httpParseRequest(request_data_, request))
    {
        LOG_ERR("Failed to parse HTTP request");
        packet_builder_sp_.buildBadRequest();
        status_page_ = true;
        return -1;
    }
    moveHttpRequest(request_, request);
    request_uri_ = request_.uri;

    if (request_uri_.empty()) {
        packet_builder_sp_.buildBadRequest();
        status_page_ = true;
        return -1;
    }
    if (!prepareRequestUri()) {
        return -1;
    }

    // Nacteni HTTP metody requestu
    request_method_ = httpMethod(request_.method, this->http_version_);
    if (request_method_ == HttpMethod::NOT_ALLOWED)
    {
        packet_builder_sp_.buildNotImplemented();
        status_page_ = true;
        return -1;
    }

    ret = validateResource();
```

```
    if (ret == -1)
    {
        packet_builder_sp_.buildNotFound();
        status_page_ = true;
        return -1;
    }
    else if (ret == -2)
    {
        packet_builder_sp_.buildForbidden();
        status_page_ = true;
        return -1;
    }

    if (!getResourceParam())
    {
        packet_builder_sp_.buildInternalServerError();
        status_page_ = true;
        return -1;
    }

    if (!checkResourceConstraints()) {
        return -1;
    }

    // Zda budu prijimat i telo HTTP requestu (metody POST, PUT)
    return receiveRequestBody();
}
```

Příjem těla požadavku realizuje metoda *receiveRequestBody*. Ta rozlišuje mezi metodami *PUT* a *POST* pro příjem těla požadavku, protože každá z těchto metod posílá tělo požadavku jinak. Metoda *PUT* odesílá tělo požadavku buď rovnou celá, nebo po částech, pokud má soubor větší velikost. V obou případech ale platí, že jedná pouze o holá data souboru. Metoda *POST* odesílá data oddělena řetězcem *boundary*, specifikovaný v atributu *Content-Type*, protože data mohou být posílána i s jinými formulářovými daty. Webový server ale zpracovává pouze data, která jsou daty odesílaného souboru přes daný formulář. Příjem těla pro metodu *PUT* zajišťuje metoda *receiveRequestBodyPutMethod* a pro metodu *POST* zase metoda *receiveRequestBodyPostMethod*. Obě metody fungují podobně, načítají tolik dat, kolik je specifikováno atributem *Content-Length* samotného požadavku. Data jsou načítána a ukládána po částech, kterou zde určuje proměnná *chunk_size*. Pokud jsou data ještě kompresována, jsou před samotným uložením ještě dekompresována pomocí funkce *Codec::decompress*. Pokud velikost přijímaných dat přesahuje *client_body_buffer_size*, tak jsou data ukládána do dočasného souboru. V opačném případě do bufferu, který následně bude zapsán do cílového souboru. To zajišťuje metoda *requestPutMethodFunc*, která po přijetí všech dat.

```
int Http1_0::receiveRequestBodyPutMethod()
{
    uint64_t content_length;
    if (headersContentLength(&content_length) == -1) {
        return -1;
    }

    if (content_length > Config::params().client_max_body_size)
    {
        packet_builder_sp_.buildContentTooLarge();
        status_page_ = true;
        return -1;
    }
```

```
    // Kontrola toho zda neni telo HTTP requestu vetsi nez nastavena velikost
bufferu pro prijem tela
    int temporary_file_fd;
    if (content_length > Config::params().client_body_buffer_size)
    {
        // Pokud je, tak ukladam do temporary file
        if (!this->createTemporaryFile(temporary_file_fd,
*const_cast<Http::TempFile*>(temp_file_)))
        {
            packet_builder_sp_.buildInternalServerError();
            status_page_ = true;
            return -1;
        }
    }

    // Nacteni tela HTTP requestu
    uint64_t total = 0;
    std::string buffer;

    while (total < content_length)
    {
        buffer.resize(std::min(content_length,
static_cast<uint64_t>(Config::params().client_body_buffer_size)));
        const uint64_t chunk_size = std::min(buffer.size(), content_length -
total);
        int ret = this->tcp_server_->receiveText(this->tcp_connection_,
buffer, chunk_size, false);
        ret = receiveRetCheck(ret, temporary_file_fd);
        if (ret != 1) {
            return ret;
        }

        // Dekompresovat data
        std::string enc_data;
        if (content_encoding_ != HttpContentEncoding::NONE)
        {
            const bool is_gzip = ((content_encoding_ ==
HttpContentEncoding::GZIP) ||
                                  (content_encoding_ ==
HttpContentEncoding::X_GZIP));

            if (!Codec::decompress_string(enc_data, buffer, is_gzip))
            {
                packet_builder_sp_.buildInternalServerError();
                status_page_ = true;
                return -1;
            }
            buffer = std::move(enc_data);
        }

        // Ulozeni casti tela HTTP requestu
        if (temp_file_->data_in_temp_file_) {
            write(temporary_file_fd, buffer.data(), chunk_size);
        }
        else {
            request_data_.append(buffer);
        }

        total += chunk_size;
    }

    if (temp_file_->data_in_temp_file_)
    {
        if (fsync(temporary_file_fd) == -1)
        {
            LOG_ERR("Failed to store request body into temp file (error: %s)",
strerror(errno));
            this->deleteTemporaryFile(temporary_file_fd, *temp_file_);
```

```
            close(temporary_file_fd);
            packet_builder_sp_.buildInternalServerError();
            status_page_ = true;
            return -1;
        }

        close(temporary_file_fd);
    }

    if (!this->requestPutMethodFunc()) {
        return -1;
    }

    return 1;
}

bool Http1_0::requestPutMethodFunc()
{
    // Ukladam resource jen pro metodu POST a PUT
    const bool file_existed =
        (access((std::string(RESOURCES_DIR) + request_uri_).c_str(), F_OK) ==
0);

    /*Pokud nebyly ulozeny data tela HTTP requestu do dosacneho souboru,
    tak jsou v request_data_ a pro dalsi zpracovani je musim odstranit*/
    if (!temp_file_->data_in_temp_file_) {
        removeRequestDataHeaders();
    }
    if (!this->storeResource(const_cast<Config::RParams*>(rparam_),
*temp_file_,
        request_uri_, request_data_.data(), request_data_.size()))
    {
        packet_builder_sp_.buildInternalServerError();
        status_page_ = true;
        return false;
    }

    const_cast<Config::RParams*>(rparam_)->update();

    if (file_existed) {
        packet_builder_sp_.buildNoContent(rparam_);
    }
    else {
        packet_builder_sp_.buildCreated(rparam_);
    }
    status_page_ = true;

    return true;
}
```

Pro odesílání odpovědi pak slouží metoda *sendResponse*. Pokud server neodesílá data žádného zdroje, tak volá metodu *sendText* instence třídy *TcpServer*. V opačném případě je zdroj odesílán po částech o velikosti, kterou definuje konfigurační parametr *file_chunk_size*. Pro odeslání dané částí souboru je volána metoda *sendFileChunk*, která zajistí mapování konkrétní části souboru do paměti a její následné odeslání. Mapování je realizováno systémovým voláním *mmap*. Metoda *sendFileChunk* také zajišťuje kompresi odesílaných dat.

```
bool Http1_0::sendResponse(HttpPacketBase& packetb)
{
    ...

    else if (packet_body->is_file_ && !packet.header().isHeadMethod())
    {
```

```
        const int file_fd = open(packet_body->data_.c_str(), O_RDONLY |
O_NOCTTY);
        if (file_fd == -1)
        {
            LOG_ERR("Failed to open file to send (file: %s)", packet_body-
>data_.c_str());
            goto err;
        }

        uint64_t sent_bytes = 0;
        uint64_t chunk_size;
        while (sent_bytes < packet_body->content_length_)
        {
            chunk_size = std::min(packet_body->content_length_ - sent_bytes,
                    static_cast<uint64_t>(Config::params().file_chunk_size));

            send_ret = sendFileChunk(sent_bytes, file_fd,
static_cast<uint32_t>(chunk_size));
            if (send_ret == -1)
            {
                LOG_ERR("Failed to send file: %s", packet_body-
>data_.c_str());
                close(file_fd);
                goto err;
            }
            else if (send_ret == 0) {
                return true;
            }

            sent_bytes += chunk_size;
        }

        close(file_fd);
    }

    ...
}

int Http1_0::sendFileChunk(const uint64_t offset, const int file_fd, const
uint32_t chunk_size)
{
    int ret = 1;

    long page_size = sysconf(_SC_PAGESIZE);
    if (page_size == -1) {
        return -1;
    }

    uint64_t page_aligned_offset = offset - (offset % page_size);
    uint64_t offset_diff = offset - page_aligned_offset;
    uint64_t mapping_size = chunk_size + offset_diff;

    void* mapping = mmap(NULL, mapping_size, PROT_READ, MAP_PRIVATE, file_fd,
page_aligned_offset);
    if (mapping == MAP_FAILED) {
        return -1;
    }

    void* data_to_send = static_cast<char*>(mapping) + offset_diff;
    size_t send_size = chunk_size;

    std::string dec_data;
    if (content_encoding_ != HttpContentEncoding::NONE)
    {
        if (!compressDataToSend(dec_data, data_to_send, chunk_size))
        {
            ret = -1;
            goto end_send;
```

```
        }
        data_to_send = const_cast<void*>(static_cast<const
void*>(dec_data.data()));
        send_size = dec_data.size();

        if (http_version_ != HttpVersion::HTTP_1_0)
        {
            char chunk_size_hex[50];
            snprintf(chunk_size_hex, sizeof(chunk_size_hex), "%zX\r\n",
send_size);
            ret = tcp_server_->sendText(this->tcp_connection_, chunk_size_hex,
strlen(chunk_size_hex));
            if (ret != 1) {
                goto end_send;
            }
        }

            ret = tcp_server_->sendText(this->tcp_connection_,
                static_cast<const char*>(data_to_send), send_size);
            if (ret <= 0) {
                goto end_send;
            }
            if (content_encoding_ != HttpContentEncoding::NONE &&
                http_version_ != HttpVersion::HTTP_1_0) {
             ret = tcp_server_->sendText(this->tcp_connection_, "\r\n");
            }
        }

end_send:
    munmap(mapping, mapping_size);
    return ret;
}
```

Třída obsahuje také metody pro zpracování jednotlivých atributů hlavičky HTTP/1.0 požadavku. Takovými metodami jsou např. *headersContentLength* či *headersAcceptEncoding*. Metody fungují tak, že nejdříve pomocí metody *getHeaderField* se pokusí získat daný parametr z deserializovaného požadavku. Pokud jej požadavek obsahoval, tak provede kontrolu jeho hodnoty a příslušné akce s ním související.

```
int Http1_0::headersContentLength(uint64_t* content_length)
{
    if (getHeaderField("Content-Length"))
    {
        errno = 0;
        const uint64_t content_len = strtoull(header_field_->value.c_str(),
NULL, 10);
        if (errno == EINVAL || errno == ERANGE)
        {
            packet_builder_sp_.buildBadRequest();
            status_page_ = true;
            return -1;
        }

        if (content_length) {
            *content_length = content_len;
        }

        return 1;
    }

    packet_builder_sp_.buildLengthRequired();
    status_page_ = true;
    return -1;
}
```

Volání jednotlivých metod pro zpracování deserializovaných atributů HTTP požadavku pak zajišťují metody jako *requestGetMethod*, *requestPutMethod* apod. Každá metoda je volána dle HTTP metody daného požadavku a zpracovává jiné, resp. atributy, které jsou potřeba.

```
bool Http1_0::requestGetMethod()
{
    if (rparam_ && !rparam_->isSet()) {
        const_cast<Config::RParams*>(rparam_)->update();
    }

    packet_builder_.createCommonHeaders(const_cast<Config::RParams*>(rparam_),
HttpStatusCode::OK, false);

    // Content negotioation (proactive content negotioation = server driven)
    if (headersAcceptEncoding() == -1) { return false; }
    if (headersIfModifiedSince() == -1) { return false; }
    return true;
}

bool Http1_0::requestPutMethod()
{
    if (headersContentType() == -1) { return false; }
    if (headersContentEncoding() == -1) { return false; }
    return true;
}
```

9.5.17 Třída Http1_1

Třída *Http1_1* implementuje HTTP verze 1.1. Třída dědí přímo ze třídy *Http1_0*, tedy zároveň i nepřímo dědí ze třídy *Http*. Třída využívá mnoha metod již implementovaných třídou *Http1_0*, jakož jsou především metody *receiveRequest* a *sendResponse*. Navíc ale rozšiřuje třídu *Http1_0* o možnost perzistentního připojení s klientem a několika atributů, které mohou jsou v hlavičce požadavku kontrolovány. Přidává také podporu HTTP metody *OPTIONS*.

Hlavní metoda *handleConnection* je zde přetížena, ale je téměř stejná jako tato metoda implementována ve třídě *Http1_0*. Umožňuje navíc pouze udržování spojení a resetování stavu objektu pomocí metody *reset* (tj. uvedení do konzistentního stavu pro možnost zpracování dalšího požadavku.

Metody pro zpracování atributů jednotlivých HTTP metod jsou také přetíženy a doplněny o kontrolu atributů, které třída *Http1_1* přidává. Příkladem je třeba metoda *requestGetMethod*.

```
bool Http1_1::requestGetMethod()
{
    if (rparam_ && !rparam_->isSet()) {
        const_cast<Config::RParams*>(rparam_)->update();
    }

    packet_builder_.createCommonHeaders(const_cast<Config::RParams*>(rparam_),
HttpStatusCode::OK, false, true);

    int ret = headersIfNoneMatch();
    if (ret == -1) { return false; }
    else if (ret == 0) {
        if (headersIfModifiedSince() == -1) { return false; }
    }

    if (headersIfNoneMatch() == -1) { return false; }
    if (headersIfMatch() == -1) { return false; }
```

```
    if (headersIfUnmodifiedSince() == -1) { return false; }

    ret = headersRange(true);
    if (ret == -1) { return false; }
    else if (ret == 1)
    {
        ret = headersIfRange();
        if (ret == -1) {
            return false;
        }
        else if (ret == 1)
        {
            packet_builder_.reset();
            packet_builder_.setEndHeaders(false);
            packet_builder_.setHttpVersion(this->http_version_);

            packet_builder_.createCommonHeaders(const_cast<Config::RParams*>(r
            param_), HttpStatusCode::PARTIAL_CONTENT, false, true);
        }
    }

    // Content negotioation (proactive content negotioation = server driven)
    if (headersAcceptEncoding() == -1) { return false; }

    return true;
}
```

Nejzajímavějším atributem, jehož podporu třída přidává, je atribut *Range*. Ten umožňuje přistupovat ke zdroji nejen jako k celku, ale po částech. Tedy klient si může vyžádat pouze specifikovanou část zdroje. To ovšem musí být povoleno pro daný zdroj v konfiguračním souboru *resources.conf*, pro ostatní zdroje je tento přístup povolen automaticky. Odeslání odpovědi, kde je vyžádána pouze část zdroje, zajišťuje metoda *sendResponseRanges*. Funguje na stejném principu jako běžné odesílání souboru, tedy volá metodu *sendFileChunk*, do které ale vstupuje HTTP požadavkem vyžádaná část zdroje.

9.5.18 Třída WebServer

Třída *WebServer* představuje nejvyšší vrstvu webového serveru. Třída je realizována pomocí návrhového vzoru *singleton* a využívá rozhraní tříd *TcpServer* a *Http* (resp. *Http1_0* a *Http1_1*). Implementuje rozhraní pro zapnutí a vypnutí serveru, načtení konfiguračních souborů a příjem nových spojení.

Samotné zapnutí serveru zajišťuje metoda *start*, která pomocí metody *loadConfigFiles* načte konfigurační soubory a inicializuje SSL/TLS. Poté zapne TCP server, který dalším voláním zajistí vytvoření a spuštění vláken.

```
bool WebServer::start()
{
	if (!server_.loadConfigFiles()) {
		return false;
	}

	if (!server_.tcp_server_->start()) {
		return false;
	}

	LOG_INFO("Web server started");
	return true;
}
```

```
bool WebServer::loadConfigFiles()
{
	if (!Config::loadConfig()) {
		return false;
	}

	if (!Config::loadResourcesConfig()) {
		return false;
	}

	if (!ssl_config_.set()) {
		return false;
	}

	return tcp_server_->set();
}
```

Zastavení serveru provádí metoda *stop*. Metoda zastaví TCP server a vlákno *https_thread_* , které slouží pro příjem spojení, které využívá šifrované komunikace.

```
bool WebServer::stop()
{
	if (server_.isRunning())
	{
		server_.tcp_server_->stop();
		if (Config::params().https_enabled &&
			server_.https_thread_.joinable())
		{
			server_.https_thread_.join();
		}

		return true;
	}

	return false;
}
```

Metoda *run* pak tvoří hlavní část hlavní smyčky programu. Zajišťuje spuštění vlákna *https_thread_* a spouští smyčku, ve které přijímá šifrované a nešifrované spojení.

```
void WebServer::run()
{
	try
	{
		if (Config::params().https_enabled)
		{
			server_.https_thread_ = std::thread([]()
			{
				try
				{
					WebServer& server = WebServer::server_;
					while (server.isRunning() &&
!server.isDeactivated())
					{
						std::shared_ptr<TcpServer::Connection>
connection = server.tcp_server_->acceptConnectionSsl();
						if (!connection->hasSocket()) {
							continue;
						}

						server.createSessionSsl(connection);
					}
				}
```

```
                        catch (const std::exception& e) {
                                LOG_ERR("Error: %s", e.what());
                        }
                });
        }

        try
        {
                while (server_.isRunning() && !server_.isDeactivated())
                {
                        std::shared_ptr<TcpServer::Connection> connection =
server_.tcp_server_->acceptConnection();
                        if (!connection->hasSocket()) {
                                continue;
                        }

                        server_.createSession(connection);
                }
        }
        catch (const std::exception& e) {
                LOG_ERR("Error: %s", e.what());
        }
    }

    catch (const std::exception& e) {
            LOG_ERR("Error: %s", e.what());
    }
}
```

Metod *run* volá metody *createSession*, resp. *createSessionSsl* pro šifrovanou komunikaci. Tyto metody zajistí přiřazení vlákna danému spojení, ve kterém zjistí verzi HTTP a na základě toho předají odpovědnost za obsluhu klienta přímo do HTTP vrstvy voláním metody *Http::handleConnection*.

```
void WebServer::createSessionSsl(std::shared_ptr<TcpServer::Connection>
connection)
{
    const bool result = tcp_server_->handleConnection(connection, [this,
connection]() mutable
    {
        try
        {
            if (tcp_server_->isConnected(connection))
            {
                std::unique_ptr<Http> http_client;
                SSL* ssl = SSL_new(ssl_config_.ctx());
                SSL_set_fd(ssl, connection->getSocket());

                // TLS handshake
                if (SSL_accept(ssl) != 1)
                {
                    if (ssl) { SSL_free(ssl); }
                    tcp_server_->endConnection(connection);
                    return;
                }
                connection->setSsl(ssl);

                if (!getClientHttpVersion(connection, http_client)){
                    return;
                }

                http_client->handleConnection();
            }
```

```
                    else {
                          tcp_server_->endConnection(connection);
                    }
             }

             catch (const std::exception& e) {
                    tcp_server_->endConnection(connection);
             }
      });

      if (!result)
      {
             LOG_ERR("Failed to create encrypted session");
             HttpPacketBuilder hpb(HttpVersion::HTTP_1_0);
             hpb.buildServiceUnavailable();
             sendToClient(connection, hpb, tcp_server_);
      }
}
```

9.5.19 Soubor WebServerd.cpp

Soubor *WebServerd.cpp* je vstupním bodem pro spuštění služby. Obsahuje hlavní smyčku programu a zajišťuje nastavení systémové služby jako takové.

Pro nastavení služby slouží funkce *serviceInit*, která nastaví obslužné funkce signálů *SIGTERM* a *SIGHUP*. Signál *SIGTERM* zasílá systemd službě, pokud má být ukončena. Signál *SIGHUP* zase když má být služba aktualizována. Signál *SIGPIPE* se ignoruje, protože je generován po zavolání systémového volání *send* potom, co druhá strana spojení dané spojení ukončila, a takovouto událost zde není potřeba obsluhovat žádnou obslužnou funkcí. Obslužné funkce signálů zároveň nastavují proměnné *daemon_run* and *daemon_reload* na příslušné hodnoty, které slouží pro řízení hlavní smyčky programu.

```
void sigTermHandler(int signum)
{
      daemon_run = false;
      daemon_reload = false;
      WebServer::stop();
}

void sigHupHandler(int signum)
{
      daemon_run = true;
      daemon_reload = true;
      WebServer::stop();
}

void serviceInit()
{
      daemon_run = true;
      daemon_reload = false;

      // Nastavit stdout bez bufferu, aby se informace hned logovaly
      setvbuf(stdout, NULL, _IONBF, 0);
      signal(SIGPIPE, SIG_IGN);
      signal(SIGTERM, sigTermHandler);
      signal(SIGHUP, sigHupHandler);
}
```

Samotné spouštění serveru zajišťuje funkce *serviceStart*. Ta pouze zavolá metodu *WebServer::start* a předá informaci systemd pomocí volání *sd_notify* o stavu zapínání služby.

```
bool serviceStart()
{
	sd_notify(0, "STATUS=Starting...");
	LOG_INFO("Starting...");

	if (!WebServer::start()) {
		return false;
	}

	if (sd_notify(0, "READY=1") < 0)
	{
		sd_notify(0, "STATUS=Starting failed");
		LOG_ERR("Start failed");
		WebServer::stop();
		return false;
	}

	sd_notify(0, "STATUS=Started");
	LOG_INFO("Started");
	return true;
}
```

Pro aktualizaci služby slouží funkce *servcieRestart*. Funkce funguje na stejně jako funkce *serviceStart* s tím rozdílem, že je zde stav webového serveru resetován.

```
void serviceReload()
{
	try
	{
		sd_notify(0, "STATUS=Reloading...");
		LOG_INFO("Reloading...");

		if (sd_notify(0, "RELOADING=1") < 0) {
			goto err;
		}

		WebServer::reset();
		if (!WebServer::start()) {
			goto err;
		}

		if (sd_notify(0, "READY=1") < 0) {
			goto err;
		}

		daemon_run = true;
		daemon_reload = false;

		sd_notify(0, "STATUS=Reloaded");
		LOG_INFO("Reloaded...");
	}
	catch (const std::exception& e) {
		goto err;
	}

	return;

err:
	daemon_run = false;
	sd_notify(0, "STATUS=Reloading failed");
	LOG_ERR("Reloading failed");
	WebServer::stop();
}
```

Pro vypnutí služby pak slouží funkce *serviceStop*, která funguje opět obdobně.

```
void serviceStop()
{
	sd_notify(0, "STATUS=Stopping...");
	LOG_INFO("Stopping...");

	if (sd_notify(0, "STOPPING=1") < 0)
	{
		sd_notify(0, "STATUS=Stopping failed");
		LOG_ERR("Failed to shut down");
		return;
	}

	WebServer::stop();
	sd_notify(0, "STATUS=Stopped");
	LOG_INFO("Stopped");
}
```

Hlavní smyčka programu je samotnou funkcí *main*. Ta zajišťuje volání výše popsaných funkcí a zároveň pak hlavní smyčku programu. Ta volá metodu *WebServer::run*, která zpracovává nově přicházející spojení. Hlavní smyčka je řízena proměnnými *daemon_run* a *daemon_reload*.

```
int main(int argc, char* argv[])
{
	try
	{
		serviceInit();
		if (!serviceStart()) {
			return EXIT_FAILURE;
		}

		// Main loop
		while (daemon_run)
		{
			WebServer::run();
			if (daemon_reload) {
				serviceReload();
			}
		}

		serviceStop();
	}

	catch (const std::exception& e)
	{
		LOG_ERR("Error: %s", e.what());
		return EXIT_FAILURE;
	}

	return EXIT_SUCCESS;
}
```

9.6 Testování funkčnosti

Testování jednotlivých funkcionalit a celkové funkčnosti webového serveru bylo nejčastěji prováděno v prostředí webových prohlížečů, které mimo samotné zobrazení přijatých dat také nabízí ladící nástroje jako je konzole, zobrazení holých dat, které byly přijaty, či sledování síťového provozu. Nejvíce používaným ladícím nástrojem byl nástroj pro sledování síťového provozu, který zobrazuje odeslaná a přijaté HTTP packety s veškerým jejich obsahem, tedy včetně atributů hlavičky a těla HTTP packetu. Webové prohlížeče využité pro test byly *Brave*, *Mozilla Firefox*, *Google Chrome* a *Microsoft Edge*. K otestování funkčnosti webového serveru byly využity i základní zátěžové testy, které jsou popsány v kapitole 9.7.

Pomocí prohlížečů bylo možné otestovat funkcionalitu šifrované, tak nešifrované komunikace se serverem, kterou byla ověřena správná funkčnost odesílání dat ze serveru. Bylo pomocí nich také možné vyzkoušet, zda webový server správně využívá kompresi dat pro tělo odpovědi. Tohoto bylo možné dosáhnout tím, že většina webových prohlížečů odesílá v hlavičce požadavku *Accept-Encoding*. Na základě tohoto atributu v požadavku a povoleném parametru *prefer_content_encoding* v konfiguračním souboru pak webový server pro data odpovědi využívá kompresi. To pak lze i vidět v ladícím nástroji pro sledování síťového provozu webové ho prohlížeče, že odpověď obsahuje atributy *Content-Encoding* a *Transfer-Encoding*. Tyto atributy webovému prohlížeči sdělují, že má přijatá data dekompresovat. Z ladícího nástroje je také vidět, že caching je správně podporován, protože na následné podmíněné požadavky (např. s atributem *If-Modified-Since* s hodnotou *ETagu*) po aktualizaci webové stránky je obdržena odpověď *304 Not Modified*.

Otestování některých funkcionality pro přístup pouze k části zdroje byl využit nástroj *curl*. Příkaz pro otestování byl následovný:

```
curl -k --ssl-reqd --http1.1 -r "0-20" -X GET -i https://127.0.0.1
```

Nástroj *curl* byl využit i pro otestování nahrávání a mazání zdrojů na serveru. Pro nahrání zdroje byl využit následující příkaz:

```
curl -k --http1.1 -X PUT --upload-file file.txt -i https://127.0.0.1/file.txt
```

Následně pro odstranění zdroje následující příkaz:

```
curl -k --http1.1 -X DELETE -i https://127.0.0.1/file.txt
```

9.7 Porovnání s webovými servery Apache a Nginx

Webové servery Apache a Nginx podporují širokou škálu mechanismů a konceptů, které implementace vlastního webového serveru nenabízí. Zřejmě tím největším rozdílem je ten, že Apache a Nginx jsou dynamickými servery, které umožňují provozovat dynamické weby, které dokážou pracovat i s databází. Dalším značným rozdílem je ten, že podporují nejmodernější verze HTTP, jakož jsou 2.0 a 3.0, které umožňují přenos a zpracování požadavků s menší latencí. Dalším rozdílem je, že Apache i Nginx mohou být provozovány i jako proxy servery.

Prvky, které naopak vlastní implementace webového serveru má společné s těmito servery, je že podporuje caching odeslaných HTTP odpovědí ze strany serveru. Dále také podpora šifrované komunikace (HTTPS) a komprese přenášených dat.

9.7.1 Srovnání výkonu

Pro měření výkonu (tzv. *load testing*) existuje několik nástrojů, kterými lze toto realizovat. Mezi ten nejkomplexnější patří *Apache JMeter*, protože umožňuje variabilní nastavení testů a zároveň nabízí i grafické rozhraní. Dalšími nástroji pro testování zátěže serveru jsou *Locust, Gatling* či *Siege*.

Jako nástroj pro srovnání výkonu serverů jsem si vybral nástroj *Siege*. Důvod k výběru tohoto nástroje je, že je velmi jednoduchý na použití a zároveň nabízí poměrně dostatečnou variabilitu nastavení pro měření výkonnosti. Hlavními parametry, které nástroj *Siege* měří, jsou: *transactions, availability, elapsed time, data transferred, response time* a *transaction rate*.

Parametr *transactions* udává počet transakcí, resp. odeslaných požadavků na server. Parametr *availability* vyjadřuje percentuálně podíl toho, kolik požadavků server zpracoval vůči celkovému počtu odeslaných požadavků. Parametr *elapsed time* vyjadřuje celkovou dobu pro vykonání celého testu. Parametr *data transfered* vyjadřuje celkové množství dat v MB, které byly přijaty ze serveru. Parametr *response time* vyjadřuje průměrnou dobu, kterou server potřeboval pro obsluhu jednotlivých požadavků. Nakonec parametr *transaction rate*, který udává průměrný počet zpracovaných požadavků za sekundu.

Nastavení všech webových serverů pro zátěžové testy je ponecháno implicitní, tedy webové servery Apache a Nginx pracují v *event-based* módu. Zátěžové testy probíhaly na *4* souběžná spojeních, kde každý zátěžový test v každém ze souběžných spojení odesílal jiný počet požadavků na server. Zátěžové testy byly prováděny na nešifrovaných spojeních, tedy HTTP. Jednotlivé požadavky byly typu *GET*. Zátěžové testy byly prováděny následujícím příkazem, kde přepínač *-c* určuje počet souběžných spojení, a přepínač *-r* počet požadavků, které má každé spojení vykonat:

```
siege -c <POCET_PRIPOJENI> -r <POCET_POZADAVKU>
http://127.0.0.1/html/index.html
```

1) Zátěžový test pro *100* požadavků

Server/ Parametr	Transactions	Availability	Elapsed time	Data transferred	Response time	Transaction rate
WebServerd	394	98,5	0,04	0,04	0,00	9850,00
Apache	400	100,00	0,02	0,10	0,00	20000,00
Nginx	400	100,00	0,02	0,05	0,00	20000,00

Tab. 1: Zátěžový test pro 100 požadavků

Zdroj: Vlastní zpracování

2) Zátěžový test pro *1000* požadavků

Server/ Parametr	Transactions	Availability	Elapsed time	Data transferred	Response time	Transaction rate
WebServerd	3987	99,67	0,50	0,45	0,00	7974,00
Apache	4000	100,00	0,21	1,03	0,00	19047,62
Nginx	4000	100,00	0,23	0,47	0,00	17391,30

Tab. 2: Zátěžový test pro 100 požadavků

Zdroj: Vlastní zpracování

3) Zátěžový test pro *10000* požadavků

Server/ Parametr	Transactions	Availability	Elapsed time	Data transferred	Response time	Transaction rate
WebServerd	39859	99,65	4,77	4,49	0,00	8356,18
Apache	40000	100,00	2,06	10,34	0,00	19417,48
Nginx	40000	100,00	2,34	4,69	0,00	17094,02

Tab. 3: Zátěžový test pro 10000 požadavků

Zdroj: Vlastní zpracování

4) Zátěžový test pro *100000* požadavků

Server/ Parametr	Transactions	Availability	Elapsed time	Data transferred	Response time	Transaction rate
WebServerd	399442	99,86	50,48	50,48	0,00	7912,88
Apache	400000	100,00	22,57	103,38	0,00	17722,64
Nginx	400000	100,00	25,33	46,92	0,00	15791,55

Tab. 4: Zátěžový test pro 100000 požadavků

Zdroj: Vlastní zpracování

Z tabulek 1 až 4 lze vidět, že webový server není nemá sice úplně stoprocentní dostupnost, a ne všechny požadavky jsou zpracovány, přesto jsou výsledky poměrně uspokojivé vzhledem k záměru aplikace. Zpracování požadavků je přibližně dvakrát pomalejší, než u serverů Apache a Nginx. Z tohoto důvodu je i parametr *transaction race* výrazně menší než u ostatních serverů. Následující tabulka (viz. tab. 5) obsahuje ještě zátěžový test webového serveru, kde komunikace probíhala s využitím šifrování. Naměřené hodnoty ale nelze přímo porovnávat s hodnotami výše naměřenými hodnotami, protože zde vzniká režie navíc kvůli šifrování a dešifrování dat. Test byl realizován následujícím příkazem:

```
siege -c <POCET_PRIPOJENI> -r <POCET_POZADAVKU>
https://127.0.0.1/html/index.html
```

Požadavky/ Parametr	Transactions	Availability	Elapsed time	Data transferred	Response time	Transaction rate
100	398	99,50	0.26	0,05	0,00	1530,77
1000	3989	99,72	2,53	0,55	0,00	1576,68
10000	39916	99,79	26,02	5,48	0,00	1534,05
100000	399000	99,75	270,24	54,79	0,00	1476,47

Tab. 5: Zátěžový test webového serveru s využitím šifrování

Zdroj: Vlastní zpracování

Závěr

Cílem práce byl implementovat vícevláknový webový server v programovacím jazyce C++ s vlastní implementací protokolů HTTP/1.x a podporou HTTPS, který je koncipován jako systémová služba v prostředí linuxových operačních systémů, s možností její konfigurace pomocí konfiguračního souboru. Na začátku práce byla provedena rešerše dostupných webových serverů a jakou funkcionalitu nabízí. Vzhledem k jejich robustnosti je vlastní implementace webového serveru minimální, protože zajišťuje podporu pouze vybraných částí protokolu HTTP verze 1.0 a 1.1 a jedná je pouze statický.

V práci se podařilo navíc zprovoznit šifrovaní komunikace (HTTPS) za využití knihovny *openssl*. Co se naopak nepovedlo byl pokus i o tzv. *event-based* implementaci, která by umožnila správu více spojení na jednom vláknu, což by přispělo k lepší škálovatelnosti serveru. Práce je obohacena také vlastním jednoduchým balíčkovacím systémem, což tvoří projekt nezávislý na balíčkách systému. Obohacena je také univerzálním překládacím systémem GNU Autotools, který umožňuje přenositelnost balíčku mezi různými architekturami a platformami.

Výsledky testování manuálního testování ukázaly, že webový server funguje správně. Z výsledků zátěžových testů lze vyvodit, že server při větším počtu požadavků je nedokáže všechny zpracovat, ale i tak jsou výsledky přesvědčivé a dostačující.

Dalším vývojem a zlepšení webového serveru je zprovoznění možnosti obsluhovat více klientů na jednom vlákně. Dále implementace podpory pro HTTP/2.0 pro lepší kompatibilitu s moderními nástroji, podpora více zpracovávaných atributů a podpora dynamických webů pomocí FastCGI, kterým je možné spouštět PHP skripty.

Seznam použité literatury

BAŘINA, David. Kompresní techniky. Online, STUDIJNÍ PODPORA. Božetěchova 1/2, 612 00 Brno-Královo Pole: Vysoké učení technické v Brně, Fakulta informatiky, 2019. Dostupné z: https://www.fit.vut.cz/person/ibarina/public/pub/MUL/2018/compression-cs.pdf. [cit. 2025-04-10].

BROADCOM INC. Cipher Suites Supported by TLS/SSL. Online. Cipher Suites Supported by TLS/SSL. 2024. Dostupné z: https://techdocs.broadcom.com/us/en/ca-mainframe-software/traditional-management/ca-xcom-data-transport-for-hp-nonstop/11-1/administrating/generating-tls-ssl-certificates/cipher-suites-supported-by-tls-ssl.html. [cit. 2024-12-22].

Cryptography - Block Cipher Modes of Operation. Online. Cryptography - Block Cipher Modes of Operation. 2024a. Dostupné z: https://www.tutorialspoint.com/cryptography/block_cipher_modes_of_operation.htm. [cit. 2024-12-22].

Cryptography - Cipher Block Chaining (CBC) Mode. Online. Cryptography - Cipher Block Chaining (CBC) Mode. 2024c. Dostupné z: https://www.tutorialspoint.com/cryptography/cipher_block_chaining_mode.htm. [cit. 2024-12-22].

Cryptography - Cipher Feedback (CFB) Mode. Online. Cryptography - Cipher Feedback (CFB) Mode. 2024d. Dostupné z: https://www.tutorialspoint.com/cryptography/cipher_feedback_mode.htm. [cit. 2024-12-22].

Cryptography - Electronic Code Book (ECB) Mode. Online. Cryptography - Electronic Code Book (ECB) Mode. 2024b. Dostupné z: https://www.tutorialspoint.com/cryptography/electronic_code_book_mode.htm. [cit. 2024-12-22].

Cryptography - Output Feedback (OFB) Mode. Online. Cryptography - Output Feedback (OFB) Mode. 2024e. Dostupné z: https://www.tutorialspoint.com/cryptography/output_feedback_mode.htm. [cit. 2024-12-22].

FAIGL, Jan. Vícevláknové aplikace. Online, STUDIJNÍ PODPORA. Praha: České vysoké učení technické v Praze, 2016. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/a0b36pr2/lectures/lecture05-slides.pdf. [cit. 2024-11-24].

GEEKSFORGEEKS. Services and Segment structure in TCP. Online. Services and Segment structure in TCP. 2021. Dostupné z: https://www.geeksforgeeks.org/services-and-segment-structure-in-tcp/. [cit. 2024-12-22].

GRIGORIK, Ilya. Building Blocks of TCP. Online. Building Blocks of TCP. 2013. Dostupné z: https://hpbn.co/building-blocks-of-tcp/. [cit. 2024-12-22].

Introducing the GNU Build System. Online. Introducing the GNU Build System. 2024, 12.7. 2024. Dostupné z: https://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html. [cit. 2025-05-05].

KENLON, Seth. Introduction to GNU Autotools. Online. Introduction to GNU Autotools. 2019, 25.7. 2019. Dostupné z: https://opensource.com/article/19/7/introduction-gnu-autotools. [cit. 2025-05-05].

KERRISK, Michael. Daemon(7) — Linux manual page. Online. Daemon(7) — Linux manual page. 2025a, 5.4. 2025. Dostupné z: https://man7.org/linux/man-pages/man7/daemon.7.html. [cit. 2025-04-05].

KERRISK, Michael. Systemd(1) — Linux manual page. Online. Systemd(1) — Linux manual page. 2025b, 2.2. 2025. Dostupné z: https://man7.org/linux/man-pages/man1/systemd.1.html. [cit. 2025-04-05].

KLÍMA, Vlastimil. Základy moderní kryptologie – Symetrická kryptografie II. Online, STUDIJNÍ PODPORA. Ke Karlovu 2027/3 Praha 2, Nové Město 121 16 Praha 2: Univerzita Karlova, Matematicko-fyzikální fakulta, 2005. Dostupné z: https://www.karlin.mff.cuni.cz/~tuma/nciphers/Symetricka_kryptografie_II.pdf. [cit. 2024-12-22].

MOZILLA CORPORATION. A typical HTTP session. Online. A typical HTTP session. 2024c. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Session. [cit. 2024-12-22].

MOZILLA CORPORATION. An overview of HTTP. Online. An overview of HTTP. 2024a. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. [cit. 2024-12-22].

MOZILLA CORPORATION. Connection management in HTTP/1.x. Online. Connection management in HTTP/1.x. 2024b. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x. [cit. 2024-12-22].

MOZILLA CORPORATION. ETag. Online. ETag. 2024e. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/ETag. [cit. 2024-12-22].

MOZILLA CORPORATION. HTTP messages. Online. HTTP messages. 2024d. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. [cit. 2024-12-22].

NECKÁŘ, Jan. Algoritmus RSA. Online. Algoritmus RSA. 2016. Dostupné z: https://www.algoritmy.net/article/4033/RSA. [cit. 2024-12-22].

OPENMV. Deflate – deflate compression & decompression. Online. Deflate – deflate compression & decompression. 2025, 9.2. 2025. Dostupné z: https://docs.openmv.io/library/deflate.html. [cit. 2025-04-10].

ORACLE. Multithreading Concepts. Online. Multithreading Concepts. 2008. Dostupné z: https://docs.oracle.com/cd/E18752_01/html/816-5137/mtintro-25092.html. [cit. 2024-11-24].

ORACLE. Semaphores. Online. Semaphores. 2010. Dostupné z: https://docs.oracle.com/cd/E19683-01/806-6867/sync-27385/index.html. [cit. 2025-04-13].

ORACLE. Thread Pools. Online. Thread Pools. 2024. Dostupné z: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html. [cit. 2025-04-13].

SSL Protocols. Online. SSL Protocols. 2013. Dostupné z: https://hstechdocs.helpsystems.com/manuals/globalscape/archive/me3/index.htm. [cit. 2024-12-22].

SSL. SSL/TLS Handshake: Ensuring Secure Online Interactions. Online. SSL/TLS Handshake: Ensuring Secure Online Interactions. 2023. Dostupné z: https://www.ssl.com/article/ssl-tls-handshake-ensuring-secure-online-interactions/. [cit. 2024-12-22].

SSL. What is an SSL/TLS Certificate? Online. What is an SSL/TLS Certificate?. 2024. Dostupné z: https://www.ssl.com/article/what-is-an-ssl-tls-certificate/. [cit. 2024-12-22].

TEOFILO, Reyonald. Atomics in C++ — What is a std::atomic? Online. Atomics in C++ — What is a std::atomic? 2023. Dostupné z: https://ryonaldteofilo.medium.com/atomics-in-c-what-is-a-std-atomic-and-what-can-be-made-atomic-part-1-a8923de1384d. [cit. 2025-04-13].

Tutorial: Logging with journald. Online. Tutorial: Logging with journald. 2025, 19.3. 2025. Dostupné z: https://sematext.com/blog/journald-logging-tutorial/. [cit. 2025-05-05].

YORU, Yilmaz. What Are The Differences Between Mutex And Shared Mutex In C++? Online. What Are The Differences Between Mutex And Shared Mutex In C++? 2023. Dostupné z: https://learncplusplus.org/what-are-the-differences-between-mutex-and-shared-mutex-in-c/. [cit. 2025-04-13].

Přílohy

Příloha A – Kompletní projekt se zdrojovým kódem

Příloha B – Soubor s instrukcemi odkazem pro stáhnutí apliance virtuálního počítače s připraveným prostředím a projektem