Česká zemědělská univerzita v Praze

Technická fakulta

Katedra elektrotechniky a automatizace



Diplomová práce

Návrh a realizace kontrolního systému na WiFi síti

Martin Novák

© 2025 ČZU v Praze

**Čestné prohlášení**

Prohlašuji, že svou diplomovou práci "Návrh a realizace kontrolního systému na WiFi síti" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne datum odevzdání \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Poděkování**

Rád(a) bych touto cestou poděkoval(a) jméno vedoucího, případně dalších osob, a informace, za co děkujete.

Návrh a realizace kontrolního systému na WiFi síti

**Abstrakt**

Souhrn práce (cca 15 řádek textu).

**Klíčová slova:** klíčová slova (cca 10)

**Design and implementation of a control system on a WiFi network**

**Abstract**

Anglický překlad českého souhrnu

**Keywords**: klíčová slova anglicky

**Obsah**

[1 Úvod 1](#_Toc193743644)

[2 Cíl práce a metodika 2](#_Toc193743645)

[2.1 Cíl práce 2](#_Toc193743646)

[2.2 Metodika 2](#_Toc193743647)

[3 Přehled řešené problematiky 3](#_Toc193743648)

[3.1 OSI model 3](#_Toc193743649)

[3.1.1 Fyzická vrstva 3](#_Toc193743650)

[3.1.2 Linková vrstva 4](#_Toc193743651)

[3.1.3 Síťová vrstva 5](#_Toc193743652)

[3.1.4 Transportní vrstva 5](#_Toc193743653)

[3.1.5 Relační Vrstva 5](#_Toc193743654)

[3.1.6 Prezentační vrstva 5](#_Toc193743655)

[3.1.7 Aplikační vrstva 6](#_Toc193743656)

[3.2 Protokoly 6](#_Toc193743657)

[3.2.1 IP 6](#_Toc193743658)

[3.2.2 DHCP 6](#_Toc193743659)

[3.2.3 UDP 7](#_Toc193743660)

[3.2.4 TCP 7](#_Toc193743661)

[3.2.5 HTTP a HTTPS 8](#_Toc193743662)

[3.3 Datové formáty 10](#_Toc193743663)

[3.3.1 XML 10](#_Toc193743664)

[3.3.2 JSON 11](#_Toc193743665)

[3.3.3 CSV 12](#_Toc193743666)

[3.4 Wi-Fi 12](#_Toc193743667)

[3.4.1 QAM 15](#_Toc193743668)

[3.4.2 Šifrování komunikace 16](#_Toc193743669)

[3.4.3 Spektrální rozprostření 16](#_Toc193743670)

[3.4.4 OFDM 18](#_Toc193743671)

[3.4.5 MIMO 18](#_Toc193743672)

[3.5 Jednočipové počítače 19](#_Toc193743673)

[3.5.1 ESP8266 20](#_Toc193743674)

[3.6 Návrhové a architektonické vzory 22](#_Toc193743675)

[3.6.1 Zapouzdření 22](#_Toc193743676)

[3.6.2 N-vrstvá architektura 22](#_Toc193743677)

[3.6.3 Dependency injection 22](#_Toc193743678)

[3.6.4 Data Transfer Object (DTO) 23](#_Toc193743679)

[3.6.5 MVVM 23](#_Toc193743680)

[4 Vlastní řešení 26](#_Toc193743681)

[4.1 Hlavní uzel 27](#_Toc193743682)

[4.1.1 Komunikační vrstva 27](#_Toc193743683)

[4.1.2 Logická vrstva 29](#_Toc193743684)

[4.1.3 Uživatelské rozhraní 40](#_Toc193743685)

[4.2 Uzly 44](#_Toc193743686)

[4.2.1 Společná část 45](#_Toc193743687)

[4.2.2 Uzel 1 47](#_Toc193743688)

[4.2.3 Uzel 2 47](#_Toc193743689)

[4.2.4 Uzel 3 47](#_Toc193743690)

[5 Výsledky a diskuse 48](#_Toc193743691)

[5.1 Podkapitola úroveň 2 48](#_Toc193743692)

[5.1.1 Podkapitola úroveň 3 48](#_Toc193743693)

[5.1.2 Podkapitola úroveň 3 48](#_Toc193743694)

[5.2 Podkapitola úroveň 2 48](#_Toc193743695)

[6 Závěr 49](#_Toc193743696)

[7 Seznam použitých zdrojů i](#_Toc193743697)

[8 Přílohy viii](#_Toc193743698)

**Seznam obrázků**

Odkazovaný seznam obrázků

**Seznam tabulek**

Odkazovaný seznam tabulek

**Seznam použitých zkratek**

Soupis a definování zkratek (vyskytuje-li se jich v textu velké množství)

# Úvod

Text text text text text text text text text text text text text text text text text text text text text text text.

# Cíl práce a metodika

## Cíl práce

Text…

## Metodika

Text…

# Přehled řešené problematiky

## OSI model

Model OSI (Open System Interconnection) je teoretickým modelem vyvinutým v roce 1984 mezinárodní organizací pro standardizaci (ISO), definující protokoly pro komunikaci různých zařízení na síti. Jedná se o sedmivrstvou architekturu (viz kapitola 3.6.2), která je vyobrazena na Obr. 1 během posílání HTTP (viz kapitola 3.2.5) dotazu. Výhodou je nezávislost jednotlivých vrstev na konkrétní implementaci ostatních, což usnadňuje případný vývoj nových technologií. Dále se snáze hledá příčina problémů s připojením. Ovšem v praxi se spíše využívá model TCP/IP (Transmission Control Protocol/Internet Protocol) slučující první a druhou vrstvu do síťového rozhraní a pátou až sedmou do aplikační vrstvy. Oproti OSI je postaven na reálných komunikačních protokolech používaných v síťových prvcích. [1–4]



Obr. OSI model[5]

### Fyzická vrstva

Na této vrstvě dochází k fyzickému přenosu dat mezi síťovými prvky. Mezi ně se počítají routery, repeatery, switche a huby. Patří sem také metalické a optické kabely, nebo radiové vlny přes která jsou data přenášena.[3]

Tato vrstva je zodpovědná za kódování a dekódování přenášených dat na nosný signál a synchronizaci mezi vysílající a přijímající stranou. Zvolené prvky určují maximální přenosovou rychlost a zda bude komunikace simplexní, polo duplexní nebo plně duplexní. Zvolená topologie sítě (Obr. 2) má vliv na spolehlivost, bezpečnost a škálovatelnost. Fyzická topologie je dána tím, jak jsou zařízení, označovaná jako uzly, vzájemně propojená. Ta se nemusí shodovat s logickou topologii, která je daná datovými toky. Ty mohou být všesměrové, nebo jeden s co nejmenším počtem prošlých uzlů potřebných do cílové destinace.[3, 6]



Obr. Typy topologií[6]

### Linková vrstva

Tato vrstva je zodpovědná za to, aby data dorazili do správného koncového zařízení. Kromě toho kontrolují, že data dorazili bez chyb. Toho je docíleno tím, že jsou data zabalena do rámce začínající adresou koncového zařízení a končící výsledkem CRC (cyclical redundancy checking) algoritmu. K adresaci zařízení využívá MAC (media access control) adresy. Dále tato vrstva má na starosti řízení datového toku, což zahrnuje určování velikosti jednotlivých rámců a určení, které zařízení momentálně řídí komunikaci.[1, 3, 7]

Algoritmus CRC slouží k detekci chyb během datového přenosu. Při odesílání je z dat vytvořen kontrolní součet o fixní velikosti. Po přijetí dat je postup zopakován a výsledek je porovnán s přijatou hodnotou. Pokud jsou shodné, byl přenos úspěšný. K výpočtu je využíváno dělení binárních polynomů. Mezi hlavní výhody této metody patří snadná implementace a rychlost výpočtu. Dále dokáže detekovat jak náhodné chyby, tak shluky chyb. Tato metoda je oblíbená pro svou robustnost a vysokou přesnost. Nevýhodou je, že se jednou pouze o detekční mechanismus, ale ne o sebe opravný kód. Množství chyb, které je možné detekovat, je určen zvoleným charakteristickým polynomem.[8]

Podoba rámce a velikost jeho datové část je dána použitou fyzickou vrstvou (Obr. 3). Pro metalické kabely se datová část pohybuje od čtyřiceti šesti do patnácti set bytů, zatím co pro Wi-Fi se rozmezí pohybuje od nuly do dvou tisíc tří set dvanácti bytů. Rozdělením paketů ze síťové vrstvy (viz kap. 3.1.3) na menší části se snižuje pravděpodobnost kolize na přenosovém mediu.[9]



Obr. Rámec 802.3 vs 802.11[9]

### Síťová vrstva

Úkolem této vrstvy je dostat data z jednoho zařízení do jiného, aniž by se tato zařízení musela nacházet ve stejné síti. Kromě toho také hledá nejkratší cestu, kterou paket musí urazit, aby se dostal do cílové destinace. K adresaci na této vrstvě se nejčastěji využívá IPv4 (Internet Protocol version 4), ale existují i jiné alternativy. Na této vrstvě pracují routery a switche. [1, 3, 4]

### Transportní vrstva

Tato vrstva na straně odesilatele data vyšší vrstvy rozloží na části nazývané segmenty a na straně příjemce opět složí do původní podoby. Součástí tohoto procesu je kontrola, že všechna data dorazila v pořádku a případné opakování komunikace. Použitý protokol a jeho implementace určují, zda se při chybě bude opakovat pouze celý přenos, pouze jeho část nebo bude chyba tolerována. K adrese síťové vrstvy přidává port, který operačnímu systému říká, které aplikaci má přijatá data předat [10]. Tímto je zajištěno, že stejné spojení může být používáno více aplikacemi současně. Transportní vrstva také řídí rychlost přenosu, aby v případě rozdílných rychlostí připojení na straně příjemce a odesilatele, nebyla jedna strana přehlcena. [1, 3, 4]

### Relační Vrstva

Úkolem této vrstvy je navazování, spravování a ukončování relací mezi zařízeními. Během komunikace jsou zařízení synchronizována a vytváří si záchytné body, takže pokud dojde k přerušení spojení, nemusí opakovat celou komunikaci, ale pouze část od posledního záchytného bodu. Tato vrstva má také na starosti autorizaci a zabezpečení.[1, 3, 4]

### Prezentační vrstva

Úkolem prezentační (někdy nazývané překladová) vrstvy je příprava dat aplikační vrstvy k odeslání na straně odesilatele a následná uvedení do čitelného stavu na straně příjemce. Toto zahrnuje šifrování, kompresy a přizpůsobení datového formátu.[1, 3, 4]

### Aplikační vrstva

Tato vrstva je nejblíže uživateli a umožňuje aplikacím volám API endpointy. Samotná aplikace není součástí vrstvy, ale poskytuje protokoly umožňující aplikacím komunikovat s ostatními zařízeními na síti. Tím je uživateli přenášet soubory, zprávy, ověřovat zařízení, vzdáleně ovládat jiná zařízení a získávat data z databází. [1, 3, 4]

## Protokoly

Protokol je sada pravidel, definující strukturu přenášených dat a průběh komunikace mezi elektronickými zařízeními. Pokud odesílající i přijímající strana používají stejný protokol, je možné zajistit efektivní a spolehlivou komunikaci, protože obě strany interpretují data stejným způsobem a vědí, jak se chovat v případě chybového stavu. Různé způsoby propojení zařízení mají rozdílné protokoly. Často je protokol používaný aplikací zabalen do jednoho či více protokolů sloužícího k přenosu (například u síťové komunikace je protokol aplikační vrstvy v datové části protokolu transportní vrstvy, který je obalen protokoly síťové a linkové vrstvy). Je nutné dělat kompromisy mezi spolehlivostí a rychlostí přenosu.[11]

### IP

Protokol IP (Internet Protocol) fungující na třetí vrstvě (viz kap. 3.1.3) OSI modelu, sloužící k směrování packetů napříč sítí. Pro tento účel slouží IP adresa, která je pro každé zařízení připojené do dané síti unikátní. V současné době se jako adresa stále využívá IPv4 s dvě na třicátou druhou možných adres tedy přibližně čtyři miliardy, které jsou zapisována jako čtveřice čísel v rozsahu 0-255 oddělené tečkou. Od roku 1998 je hotový protokol IPv6 s dvě na sto dvacátou osmou adres, což je přibližně tři sta čtyřicet sextilionů. IPv6 adresa je zapsaná jako osm hexadecimálních čísel v rozsahu 0000-FFFF oddělených dvojtečkou. Ačkoliv s počtem adres je problém již tři desetiletí, změna stále neproběhla, protože by bylo nutné nahradit celou infrastrukturu, což je velice nákladné. [12, 13]

Aby mohl internet dále fungovat bylo potřeba udělat opatření, které sníží počet potřebných adres na internetu. Tím že seznamu všech možných adres část vyhradí se pro podsítě, se umožňující, aby se tyto adresy opakovali. K určení, zda jsou zařízení ve stejné podsíti, slouží masky, které v binárním zápisu mají v místě, kde se musí shodovat jedničku a v části adresy určující konkrétní zařízení nulu. Routery a modemy mají dvě adresy. Jednu pro vnitřní síť (obvykle značenou jako LAN = Local Area Network) a jednu pro vnější síť (obvykle značenou jako WAN = Wide Area Network). Přijde-li packet s adresou odpovídající masce vnitřní sítě, je přesměrován do zařízení nacházejícího se ve stejné síti. V opačném případě je pomocí NAT (Network Address Translation) nahrazena adresa zařízení ve vnitřní síti na vnější adresu routeru a paket je odeslán mimo lokální síť. Kromě umožnění připojení více zařízení, než kolik je IPv4 adres zvyšuje NAT bezpečnost sítě. Jelikož všechna zařízení jsou na WAN viditelná pod jednou adresou, je pro útočníky obtížné zjistit podobu vnitřní sítě. [14, 15]

### DHCP

Protokol DHCP (Dynamic Host Configuration Protocol) umožňuje automatickou konfiguraci IP adres v síti. Bez DHCP by bylo nutné manuálně přidávat a odebírat zařízení ze seznamu adres a na zařízení nastavovat adresu brány (lokální IP adresa routeru), masku sítě a zajistit že adresa zařízení je v síti unikání. DHCP má k dispozici seznam dostupných adres, které přiřazuje nově připojeným zařízením. Když se zařízení odpojí je adresa opět dostupná a je možné ji přiřadit jinému zařízení. Jelikož je proces automatizovaný je eliminována lidská chyba a je usnadněna správa sítě. [16]

### UDP

Jeden ze dvou hlavních protokolů čtvrté vrstvy (viz kap. 3.1.4) OSI modelu je UDP (User Datagram Protocol), sloužících ke komunikaci na síti. Má velice jednoduchý princip, kdy pakety pošle do cílové destinace bez navazování spojení, či ověřování, že všechny dorazily v pořádku. Je vhodný především v situacích, kdy je důležitější rychlost než spolehlivost, nebo když se nehodí očekávat odpověď. Typickým příklad použití je přehrávání audia a videa, video hovory a online hry, kdy opakované vysílání ztracených paketů již nemá smysl. Aplikace ovšem musí počítat se situacemi, kdy některé z paketů budou ztraceny, duplikovány, nebo dorazí v jiném pořadí, než byly odeslány. [17]

### TCP

Druhým z hlavních protokolů čtvrté vrstvy OSI modelu je TPC (Transmission Control Protocol). Na rozdíl od UDP je zde zajištěno, že když pakety dorazí ve špatném pořadí budou seřazeny správně. V případě, kdy je paket ztracen, požádá o opakované poslání konkrétního paketu. Před zahájením komunikace je navázáno spojení pomocí třístupňového ověření (anglicky Three-way handshake) kdy, jak je vidět na Obr. 4 jedna strana žádá o navázání spojení, druhá potvrdí žádost a současně požádá o spojení, které první strana potvrdí. Toto je provedeno pomocí bitů *SYN* a *ACK* v TCP hlavičce (viz Obr. 5), kde datová část bývá obvykle prázdná. Jakmile je navázáno spojení začíná odesílající strana posílat pakety, po jejich obdržení přijímající strana pošle potvrzení. Pokud do stanovené doby nedorazí potvrzení, předpokládá se, že byl paket ztracen a je zopakování jeho odeslání. Obdrží-li příjemci paket s vyšším číslem, než které očekává pošle potvrzení očekávaného, čímž dá odesilateli najevo, že má špatné pořadí a potřebuje znovu poslat chybějící. Příjemce si může podle pořadových čísel obdržené pakety seřadit do správného pořadí a rozeznat duplicity. Pokud chce jedna ze stran spojení ukončit, zopakuje podobný postup jako při navazování spojení, ale místo *SYN* je v logické jedničce bit *FIN*.[18, 19]



Obr. Třístupňové ověřování [18]



Obr. TCP hlavička [18]

### HTTP a HTTPS

Protokol HTTP (Hypertext Transfer Protocol) fungující na sedmé vrstvě (kap. 3.1.7) OSI modelu je základem výměny dat na internetu. Jedná se klient-server protokol, kdy klient pošle požadavek na server, který ho zpracuje a pošle zpět odpověď. Byl vyvinut počátkem devadesátých let dvacátého století jako rozšířitelný protokol, což kromě umožňuje kromě textu posílat i obrázky, videa a další datové soubory. Nové funkce lze snadnou doplnit přidáním nového atributu do hlavičky dotazu. HTTP je bez stavový protokol, ale umožňuje využít cookies soubory, které jsou uloženy u klienta a v případě potřeby mohou být přiloženy k dotazu. Ke komunikaci se využívá TCP (kap 3.2.4) protokol, kvůli vytváření spojení. [20]

Mezi klientem a serverem mohou být proxy servery, které dotazy pouze přeposílají, nebo mají jednu či více funkcí. První možnou funkcí je cache, která má uložené odpovědi pro časté dotazy, takže není potřeba zatěžovat server [21]. Druhou je odfiltrování potencionálně škodlivých dotazů. Třetí možnou funkcí je load balancing, kdy klient volá proxy server, který pak podle vytížení jednotlivých serverů zvolí, na který z nich bude dotaz přeposlán, například podle lokace nebo zajištění rovnoměrného rozložení zátěže [22]. Čtvrtou funkcí je autorizace dotazů, aby se ke zdrojům nedostala neoprávněná osoba. Poslední z běžně využívaných funkcí je logování dotazů, které mohou být zpětně použity k analýze. [20]

Verze HTTP/1.1 a starší jsou v podobě která je čitelná pro lidi. Od verze HTTP/2.0 jsou zprávy zabaleny do rámců, které umožňují kompresy a multiplexing. Struktura zprávy se liší v závislosti na tom, zda se jedná o dotaz, nebo odpověď (viz Obr. 6 a Obr. 7). U dotazu je nutné uvést o jaká metoda se provést. Nejběžnější jsou GET pro načtení dat a POST pro odeslání dat v těle dotazu. Cesta adresa od kořenového adresáře k zdroji nebo endpointu, o který klient žádá. Hlavička obsahuje dodatečné informace pro server, jako je například autorizace, očekávaný jazyk, způsob kódování a další. Obdobný význam má hlavička odpovědi pro klienta, ale místo metody a cesty obsahuje status kód a zprávy. Kód je tříciferné číslo, u něhož stovky určují kategorii a zbylé dvě číslice konkrétní stav. Jednička jsou informační zprávy, ale nejsou využívány tak často jako ostatní. Dvojka na začátku znamená, že dotaz byl v pořádku zpracován. Trojka znače přesměrování dotazu jinam. Čtyřka znamená chybu na straně klienta, zatímco pětka je chyba na straně serveru. [20, 23, 24]

|  |  |
| --- | --- |
| Obr. HTTP-dotaz [20] | Obr. HTTP-odpověď [20] |

Jelikož HTTP je nešifrované, je možné komunikaci odchytit a přečíst si obsah. Z tohoto důvodu bylo vytvořeno HTTPS (Hypertext Transfer Protocol Secure), které využívá SSL/TLS (Secure Sockets Layer/Transport Layer Security), jenž jsou založeno na asymetrické kryptografii, kdy data zašifrovaná pomocí veřejného klíče mohou být dešifrována pouze soukromím klíčem [25]. Aby bylo možné navázat spojení musí server mít platní certifikát vystavený nezávislou certifikační agenturou. V opačném případě klient ukončí komunikaci. Během navazování spojení je proveden TLS handshake, který ve verzi 1.2 probíhá tak, že klient pošle serveru seznam podporovaných šifer a server odpoví co během komunikace budou používat. Obě tyto zprávy obsahují náhodné číslo, které druhá strana použije k vygenerování klíče například pomocí RSA nebo Diffie-Hellman algoritmu. Toto číslo dále brání útočníkovi použití zprávy odchycené v minulosti. Server poté pošle svůj certifikát obsahující veřejný klíč a klientovu zprávu zašifrovanou soukromým klíčem. Klient použije veřejný klíč certifikační agentury, která měla certifikát vydat k ověření jeho pravosti. Poté klíčem serveru dešifruje zprávu, čímž ověří že odesilatel disponuje příslušným soukromím klíčem. Server dále pošle zprávu, kterou oznamuje, že poslal všechny potřebné údaje. Klient pošle svůj premaster secret, oznámení konce nešifrované komunikace a zašifrované shrnutí dosavadní komunikace. Server také pošle zašifrované shrnutí. Pokud se tyto dvě shrnutí liší, znamená to, že někdo sedí uprostřed a další komunikace není bezpečná. Od této chvíle může probíhat bezpečná komunikace. TLS 1.3 tuto výměnu zkracuje a zakazuje použití šifer, které již byly prolomeny, ale mnoho serverů a klientů stále využívá TLS 1.2, které je zpětně kompatibilní se staršími verzemi. [26–31]



Obr. HTTPS komunikace [27]

## Datové formáty

Za běhu programu jsou situace, kde je třeba objekty v paměti uložit či přenést do jiného programu. Tomuto procesu se říká serializace. Dochází během něho k zachycení aktuálního stavu objektu, který je reprezentován kombinací hodnot v jeho vlastnostech. Serializovat je možné pouze data nikoliv metody. Reverzní operaci, kdy je tato reprezentace převedena zpět na objekt, se nazývá deserializace. Nejčastější formáty jsou často součástí systémových knihoven daného jazyka, nebo existuje knihovna třetí strany. Je důležité, aby se shodovala struktura v programu i serializované verze, protože v opačném případě může při deserializace nastat chyba. [32]

Data je možné přenášet a ukládat v binární nebo textové podobě. Při použití binární podoby je zpracování rychlejší, ale obsah je pro člověka nečitelný a všechny zúčastněné strany musí znát význam jednotlivých bitů. Textová podoba je čitelná pro všechny, což umožňuje snadnou editaci a jednodušší hledání příčin chyb, protože si programátor může lehce ověřit, zda mají data očekávanou podobu. Nevýhodou je nutná konverze do příslušných datových typů. [33]

### XML

Značkovací jazyk XML (eXtensible Markup Language) popisuje strukturu dat. Oproti některým jiným značkovacím jazykům neobsahuje informaci o jejich významu. Ten musí znát aplikace, což znamená, že XML vytvořený jedním programem, může být pro jiný nečitelný. Na Obr. 9 je ukázka XML s kolekcí psů. Dokumenty obvykle začínají nepovinou značkou obsahující verzi a kódování. Veškerý obsah musí být zabalen do jednoho kořenového prvku. Jednotlivé značky mohou být rozšířeny o atributy obsahující doplňující informace o textu, který je jimi ohraničen. Nejčastěji se jedná o identifikátory nebo vzhled. Hodnoty jsou zapisovány do uvozovek. XML a značkovací jazyky na něm založené podporují komentáře, které jsou při zpracování ignorovány. Dále je možné přidat sekci *CDATA*, jejíž obsah je ponechán nezměněn, což je využíváno, pokud je třeba uložit text obsahující značkovací jazyk, který byl jinak zpracován. [33]

Při zpracování XML jsou rozlišovány dva základní typy. SAX (Simple API for XML) projde dokument pouze jednou a v závislosti na právě přečtené značce vyvolá příslušnou událost. Tento přístup vyžaduje méně paměti, ale aplikace si musí pamatovat vztahy mezi jednotlivými daty. Oproti tomu DOM (Document Object Model) uchovává celý dokument ve stromové struktuře. Tento přístup potřebuje více paměti, ale kdykoli se dá přistoupit k jakémukoliv prvku včetně jeho kontextu. DOM je využíván například u webových stránek, ODF (OpenDocument Format používaný v OpenOffice), Open XML (využívaný v Microsoft Office od verze 2007), SVG (Scalable Vector Graphics) nebo .NET aplikacích využívajících XAML (Extensible Application Markup Language). [34]



Obr. Příklad XML

### JSON

Formát JSON (JavaScript Object Notation) vznikl původně pro převod objektů v JavaScript do textového řetězce. JSON podporuje pouze základní datové typy jako jsou textové řetězce, čísla, logické hodnoty a objekty, či pole z těchto typů sestavené. V případě potřeby je možné do objektu vnořit další objekt. Zápis je tvořen páry skládajících se z názvu a hodnoty oddělených dvojtečkou. jednotlivé páry jsou od sebe odděleny čárkou. Veškerý obsah musí být obalen složenými závorkami označující objekt, nebo hranatými závorkami značící kolekci. JSON byl implementován mnoha jazyky jako jsou například C, C++, C#, Java, Python a mnoho dalších. Navzdory svému názvu není závislí na konkrétním jazyku, což ho dělá ideální volbou pro sdílení dat mezi programy napsaných v různých technologiích. Za nevýhodu by se dalo označit nemožnost používat komentáře. Na Obr. 10 je příklad zápisu kolekce psů. [35, 36]



Obr. Příklad JSON

### CSV

Formát CSV (Comma-Separated Values) je používaný k ukládání tabulek. Jedná se o jednoduchý a hojně rozšířený formát pro import a export dat. Každý řádek textu odpovídá jednomu řádku v tabulce. Jak název napovídá sloupce jsou většinou oddělovány čárkou, ale v některý případech (například kvůli českým desetinným číslům s desetinou čárkou místo tečky jako se používá v angličtině) je nutné použít jiný oddělovací znak (obvykle středník nebo svislítko) [37]. První řádek se většinou využívá k pojmenování jednotlivých sloupců. Na Obr. 11 je příklad z tabulky psů. Oproti ostatním formátům má výhodu v menší velikosti, neboť význam hodnoty je definován pouze jednou nikoli pro každou instanci. Toto sebou ovšem nese nevýhodu, že jeden soubor může obsahovat pouze záznamy stejného typu, jelikož v opačném případě se nebudou shodovat sloupečky. CSV má nezastupitelné využití při exportu dat z databází a v situacích kdy se předpokládá, že data budou zpracovávána uživatelem například pomocí nástrojů jako je Microsoft Excel. [38]



Obr. Příklad CSV

## Wi-Fi

Wi-Fi je uživatelsky přívětivější název pro bezdrátovou síťovou technologii definovanou standardem IEEE (Institute of Electrical and Electronics Engineers) 802.11, popisující první a druhou vrstvu OSI modelu (viz Kap. 3.1), jehož první verze vznikla v roce 1997. Větší rozšíření Wi-Fi nastalo po uvedení Apple AirPort v roce 1999 využívající 802.11b, který je někdy také označovaný jako Wi-Fi 1. Původní verze Wi-Fi měla maximální šířku pásma pouze 2 Mb/s a využívala frekvenční pásmo 2,4 GHz. 802.11b fungoval na stejné frekvenci, ale zvýšil přenosovou rychlost na 11 Mb/s. kromě vyšší rychlosti také využíval modulační schéma DSSS/CCK (Direct-Sequence Spread Spectrum/Complementary Code Keying) snižující vliv rušení způsobeného mikrovlnnými troubami, bezdrátovými telefony a jinými zdroji elektromagnetického záření. Ve stejném roce vyšel také standart 802.11a pracující ve frekvenčním pásmu 5 GHz s maximální rychlostí 54 Mb/s. Bylo zde také poprvé představeno OFDM (Orthogonal frequency-division multiplexing). 5 GHz má oproti 2,4 GHz výhodu vyšší rychlosti, ale za cenu kratšího dosahu. V roce 2003 byl představen standart 802.11g využívající technologie 802.11a na 2,4 GHz síti. V následujících letech přibyly další verze (viz Tab. 1) s vyšší přenosovou rychlostí, dosahem a pokročilými technologiemi umožňující vyšší spolehlivost, bezpečnost a komunikaci více zařízení současně. [39–41]

Tab. Verze Wi-Fi [39, 42]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| standart | Wi-Fi | rok | Frekvence [GHz] | Přenosová rychlost (teoretická) | Šířka kanálu [MHz] |
| 802.11 | Wi-Fi 0 | 1997 | 2,4 | 2 Mb/s | 20 |
| 802.11a | Wi-Fi 2 | 1999 | 5 | 54 Mb/s | 20 |
| 802.11b | Wi-Fi 1 | 1999 | 2,4 | 11 Mb/s | 20 |
| 802.11g | Wi-Fi 3 | 2003 | 2,4 + 5 | 54 Mb/s | 20 |
| 802.11n | Wi-Fi 4 | 2009 | 2,4 + 5 | 600 Mb/s | 20, 40 |
| 802.11ac | Wi-Fi 5 | 2013 | 5 | 3,5 Gb/s | 20, 40, 80, 160 |
| 802.11ax | Wi-Fi 6 | 2019 | 2,4 + 5 | 9,6 Gb/s | 20, 40, 80, 160 |
| 802.11ax | Wi-Fi 6E | 2021 | 2,4 + 5 + 6 | 9,6 Gb/s | 20, 40, 80, 160 |
| 802.11be | Wi-Fi 7 | 2024 | 2,4 + 5 + 6 | 46 Gb/s | 20, 40, 80, 160 |

Wi-Fi má v daném frekvenčním pásmu vymezený určitý rozsah frekvencí, které jsou rozděleny na 22 MHz úseky nazývané kanály. Dostupnost těchto kanálů se liší v závislosti na regulacích telekomunikačních úřadů jednotlivých států. Wi-Fi v pásmu 2,4 GHz může teoreticky mít až čtrnáct kanálů. V České republice je stejně jako ve většině Evropy a Spojených státech možné využít třináct kanálů, odpovídající frekvencím od 2,4000 do 2,4835 GHz. Jednotlivé kanály mají však rozestupy pouze 5 MHz, což znamená že sousední čtyři kanály na obě strany jsou vzájemně rušeny (viz Obr. 12). Pro vyšší datovou propustnost je možné zvolit i jinou šířku (viz Tab. 1), ale za cenu ztráty zpětné kompatibility se staršími zařízeními a menší počet vzájemně nerušených kanálů. [43, 44]



Obr. Překryv kanálů 2,4 GHz [43]

K realizace bezdrátové sítě neboli WLAN (Wireless Local Area Network) je potřeba zařízení nazývané AP (Access Point). Jedná se zařízení vysílající bezdrátový signál, který mohou zachytit koncová zařízení (označovaná jako stanice či zkráceně STA) v dosahu. K připojení do této sítě je potřeba znát SSID (Service Set IDentifier) a heslo (pokud není síť nezaheslovaná). SSID je možné zadat ručně, pokud ho uživatel zná předem, nebo ho získat ze speciálních paketů nazývaných beacon (někdy také SSID broadcast), které AP pravidelně vysílá na všech kanálech. Obvykle bývá součástí routeru, ale může se jednat i o samostatné zařízení. Síť může být tvořena jedním či více AP, která jsou propojena kabelem. [45, 46]

Oproti rámce pro Ethernet (IEEE 802.3), kterému k úspěšnému doručení stačí pouze dvě MAC adresy (viz Obr. 3), obsahuje Wi-Fi rámec (jehož podoba je detailněji popsána na Obr. 13) čtyři adresy. O jejich významu rozhodují devátý a desátý bit hlavičky, které obsahující informaci o směru toku dat (viz Obr. 14). V závislosti na situaci se může jednat o MAC adresu zařízení, nebo BSSID (Basic Service Set IDentifier) sítě vysílané určitým AP (viz Tab. 2). Rámce mohou mít několik významů, které určují třetí až osmí bit hlavičky. Bit *More Frag* slouží jako indikátor, zda byl paket rozdělen na více rámců (viz Kap. 3.1.2). IEEE 802.11 obsahuje také úsporný režim, kdy koncové zařízení vypne napájení antény za účelem úspory energie. V případě změny tohoto stavu posílá koncové zařízení AP rámec, který neobsahuje žádná data. Bit *Pwr Mgmt* říká, zda po odvysílání tohoto rámce bude zařízení aktivní, nebo úsporném režimu. S tím souvisí i další bit určující, zda má být rámec odvysílán, nebo uložen do doby, než bude cílové zařízení probuzeno. [47]

|  |  |
| --- | --- |
| Obr. Rámec Wi-Fi [47] | Obr. Význam DS bitů [47] |

Tab. Význam adres v Wi-Fi rámci [47]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| To DS | From DS | ADR 1 | ADR 2 | ADR 3 | ADR 4 | Situace na Obr. 14 |
| 0 | 0 | Cílová adresa | Zdrojová adresa | BSSID | - | Beacon |
| 0 | 1 | Cílová adresa | BSSID | Zdrojová adresa | - | (1) AP1 posílá data STA1 |
| 1 | 0 | BSSID | Zdrojová adresa | Cílová adresa | - | (2) STA2 posílá data AP1 |
| 1 | 1 | Cílová BSSID | zdrojová  BSSID | Cílová adresa | Zdrojová adresa | (3) AP1 posílá data AP2 |

Od 802.11ax se pro zjednodušení pro koncové uživatele místo značení verze pomocí standardu používá číslo generace (viz Tab. 1). Ačkoliv finální verze IEEE standardu vyšla až v roce 2021, Wi-Fi Aliance vydávala certifikáty již od roku 2019. Primárním cílem nové verze je zvýšení schopnosti současně komunikovat s více uživateli v prostředích s velkým množstvím zařízení, jako jsou například sportovní stadiony a dopravní uzly. Díky technologii OFDMA (Orthogonal Frequency Division Multiple Access) je možné jednotlivé subcarriery (viz Kap. 3.4.4) rozdělit na menší úseky, které mohou být přiřazeny jednotlivým zařízením. Zavedením plánování komunikace je snížen počet kolizí na síti, čímž je zvýšena datová propustnost, neboť není potřeba opakovat vysílání. Zlepšena byla také bezpečnost použitím WPA3 využívajícího SAE (Simultaneous Authentication of Equals). Kromě toho se také prodloužila výdrž baterií napájených zařízení, neboť nyní místo soustavného kontrolování, zda mu někdo něco neposílá, je probuzeno až v případě potřeby. 802.11ax obsahuje také Wi-Fi 6E pracující v nově uvolněném frekvenčním pásmu 6 GHz umožňující přenášet větší množství dat. [40, 48–50]

### QAM

Modulace QAM (Quadrature Amplitude Modulation) je způsob, jak do bezdrátového signálu zakódovat více informací. Tato modulace mění amplitudu a fázi signálu. Během modulace jsou data rozdělena na polovinu. Obě tyto části jsou modulovány pomocí sinusoid, které jsou vůči sobě posunuty o devadesát stupňů (viz Obr. 15). Polovina obsahující LSB (Least Significant Bit) nemá fázové posunutí a označuje se proto jako I (In phase). Polovina obsahující MSB (Most Significant Bit) se označuje jako Q (Quadrature). Obě tyto sinusoidy jsou poté sečteny, čímž je získán výsledný signál k odvysílání (viz Obr. 17). Možné stavy se dají znázornit pomocí dvourozměrného grafu, kde na ose x je I a na ose y je Q (viz Obr. 16). Množství možných hodnot je přímo v názvu použité modulace. Například 256-QAM znamená, že signál může nabývat dvě stě padesát šest různých stavů, tedy přenáší osm bitů. Z tohoto značení je výjimkou QPSK, který odpovídá 4-QAM. Ovšem s vyšším počtem možných stavů se zvyšuje také jejich hustota, což znamená že v případě rušení nemusí být symbol správně rozpoznán. Proto vyšší QAM je možné použít pouze na kratší vzdálenosti.[51]



Obr. Schéma QAM modulátoru [51]

|  |  |
| --- | --- |
| Obr. Graf 16-QAM [51] | 16-QAM diagram  Obr. Signál 16-QAM [52] |

### Šifrování komunikace

Jelikož data nejsou data vysílána do zařízení přímo, ale je možné je zachytit jinou anténou poblíž, je nutné provádět šifrovány, aby k nim nezískala přístup neoprávněná osoba. Prvním protokolem byl WEP (Wired Equivalent Privacy), který používal šedesáti čtyř nebo sto dvaceti osmi bitový statický klíč. Ten byl stejný pro všechny zařízení, a tudíž chránil pouze před útočníky, kteří nebyli k síti připojeni. Kromě toho nová zařízení mají dostatek výkonu k prolomení šifry a není proto doporučeno WEP nadále používat. V roce 2003 s 802.11g přišlo WPA (Wi-Fi Protected Access) řešící zranitelnost statických klíčů využitím TKIP (Temporal Key Integrity Protocol). Ten je generován pro každý paket. Jelikož je klíč jednorázový, má útočník méně informací použitelných k zjištění klíče. Dále WPA obsahuje mechanismus k ověření integrity dat v případě, že s nimi bylo manipulováno. O rok později bylo představeno WPA2 využívající AES (Advanced Encryption Standard). Navíc byla vylepšena i autorizace, kdy kromě hesla používaného v soukromém režimu, přibyl enterprise režim využívající EAP (Extensible Authentication Protocol), který ověřuju identitu vůči serveru. Nejnovější verze WPA3 z roku 2018 vnikl kvůli objevený zranitelnostem WPA2 (CVE-2017-13077 až CVE-2017-13088 [53]). Klíče jsou nyní unikátní pro každý přenos a mají sto devadesát dva bitů v osobním režimu a dvě stě padesát šest v enterprise režimu. [54]

### Spektrální rozprostření

Jedná se o metody používané k snížení vlivu rušení a zvýšení bezpečnosti bezdrátově přenášených signálů. Základem je výzkum z roku 1941, ve které se herečka Hedy Lamarr a pianista George Antheil snažili najít způsob, jak zabránit rušení signálu pro radiem řízená torpéda. Americká armáda toto řešení však odmítla. Dnes je využíváno pro Wi-Fi, Bluetooth, mobilní sítě a GPS (Global Positioning System). Hlavní myšlenkou je rozprostřít signál přes více frekvencí, díky čemuž je potřeba menší energie, neboť v případě interference nejsou ovlivněny všechny a není tedy nutné, aby byl signál silnější než šum. S tím souvisí složitější odposlech komunikace, protože bez znalosti příslušných frekvencí se signál od šumu nedá odlišit (platí obzvláště pro DSSS). Toto navíc umožňuje ve stejném frekvenčním pásmu vysílat více signálů. [44, 55]

Metoda rozprostření DSSS (Direct Sequence Spread Spectrum) kombinuje data s pseudonáhodným kódem o vyšší frekvenci, než jsou data. Jednotlivé hodnoty tohoto kódu se nazývají chipy. Jeden bit je přenášen pomocí jedenácti chipů, které jsou s daty zkombinovány pomocí funkce XOR (viz Obr. 18). oproti ostatním spektrálním rozprostření má výhodu vyšší odolnosti proti šumu, ale za cenu potřeby širšího frekvenční pásma, kvůli čemuž má méně dostupných kanálů. [44, 55]



Obr. DSSS [55]

Metoda FHSS (Frequency-Hopping Spread Spectrum) je možné použít jako alternativu k DSSS, či je zkombinovat dohromady. Oproti DSSS nemanipuluje přímo s daty, ale provádí skoky mezi sedmdesáti osmy frekvenčními pásmy (viz Obr. 19). Jelikož každých pár bitů mění frekvenci, útočník tak není schopen zachytit celou zprávu. V případě rušení v daném rozsahu, může díky úzkým kanálům stále využívat ty, které nejsou rušené. Má menší datovou propustnost, protože na přeskok potřebuje více času. [44, 55]



Obr. FHSS [55]

### OFDM

OFDM (Orthogonal Frequency-Division Multiplexing) je způsob, jak vyřešit problém s odrazy signálu, které komplikují rozeznaní jednotlivých bitů, kvůli echu. Data jsou rozdělena mezi více samostatně modulovaných signálů označovaných subcarriers, které jsou voleny tak, aby v momentě, kdy je daná frekvence na vrcholu byly všechny ostatní v nule. Poté co je provedena modulace, jsou signály sečteny a odeslány pomocí antény. Na straně přijímače je signál pomocí rychlé Fourierovi transformace (FFT = Fast Fourier Transform) opět rozložen a demodulován (viz Obr. 20). [56]



Obr. OFDM přijímač [57]

### MIMO

Mechanismus MIMO (Multiple-Input Multiple-Output) umožňuje v jeden okamžik na straně vysílače i přijímače využit více antén. Poprvé bylo představeno v 802.11n v podobě SU-MIMO (Single User MIMO). podobně jako jeho předchůdci, kdy pouze jedna ze stran měla dvě antény (viz Obr. 21), sloužilo k vyšší spolehlivosti. SU-MIMO využívá všechny antény pro stejný signál, tudíž má přijímač více informací umožňující vyčistit data od šumu, nebo pokud má vyšší šanci obdržet data, je-li signál rušen či odražen od překážky. Dále může být anténa navíc být využita k zvýšení rychlosti přenosu tím, že jsou data rozdělena na více částí a každá poslána jako samostatný signál. S příchodem 802.11ac byl tento mechanismus rozvinut do podoby MU-MIMO (Multi User MIMO) umožňující každé anténě AP komunikovat s jiným zařízením. Počet souběžných komunikací je limitován schopnostmi AP. Tento údaj je vyjadřován pomocí *MxN*, kde M je počet antén pro vysílání a N je počet antén pro příjem. Obvykle umí jedna anténa plnit obě funkce. [58–60]



Obr. SISO, SIMO, MISO, MIMO [60]

## Jednočipové počítače

V oblasti obvodů s vysokým stupněm integrace (VLSI = Very large Scale Integration), tedy integrovaný obvod, jenž obsahují více zařízení, je několik pojmů, které si jsou velice blízké a někdy dochází k jejich záměně. [61, 62]

Mikroprocesor je označení integrovaného obvodu obsahující vykonávající a řídící jednotku, jenž dohromady tvoří CPU (Central Processing Unit). Jejich umístění do jednoho čipu zvyšuje spolehlivost, neboť je sníženo množství míst, kde by mohl nastat problém při kombinaci více čipů. Prvním integrovaným obvodem tohoto typu byl čtyř bitový Intel 4004 z roku 1971 s frekvencí 740 kHz. Na Obr. 22 je vyobrazena struktura jednoduchého mikropočítače, tedy zařízení založeném na mikroprocesoru. [61, 63]

SoC (System-On-Chip) je integrovaný obvod obsahující všechny základní části počítače v jednom pouzdře. Tyto čipy obsahují procesor, cache, paměť a vstupně výstupní obvody. Toto umožňuje zjednodušení výroby a tím snížení nákladů. Dále zařízení využívající SoC mohou mít menší rozměry a spotřebu než ta používající více čipů. Často je využívána Von Neumannova architektura, kdy jsou instrukce i data umístěna do jedné paměti, která je přímo adresována CPU a označuje se jako primární nebo hlavní paměť. [61]

Mikrokontrolery neboli jednočipové počítače, často zkracované jako MCU (MicroController Unit), jsou speciální případem SoC, jenž nevyužívají externí DRAM (Dynamic Random Access Memory). Většinu vstupně výstupních obvodů a ROM (Read Only Memory) s programem, jenž mají vykonávat, obsahují přímo v sobě. Ke své funkci potřebují pouze zdroj hodinového signálu a napájení (viz Obr. 23). Ve většině případů obsahují časovače a převodníky analogového signálu na digitální (ADC = Analog-to-Digital Converter). Kromě těchto základních obvodů jsou typicky vybaveny sběrnicemi (např. I2C, SPI a další) umožňujícími připojit složitější sensory, Které nepoužívají pouze jednu logickou hodnotu, či napětí v rozmezí 0-3 V, ale komunikují pomocí zpráv tvořených jedním či více bajtů. Díky těmto vlastnostem jsou ideální pro úlohy vyžadující zpracování signálů v reálném čase. [61, 64]



Obr. Struktura jednoduchého mikropočítače [61]



Obr. Struktura mikrokontroleru [61]

### ESP8266

ESP8266EX od společnosti Espressif je SoC s QFN32-pin pouzdrem o rozměrech 5x5 mm kombinující vylepšenou verzi třiceti dvou bitového procesoru Tensilica L106 Diamond series, jehož maximální frekvence může být až 160 MHz, a 2,4GHz Wi-Fi 802.11 b/g/n s rychlostí až 72,2 Mb/s. Je možné ho použít buď samostatně, nebo jako periferii k jinému mikroprocesoru. K napájení lze využít napětí v rozmezí od 2,5 V do 3,6 V s průměrným odebíraným proudem 80 mA. Odebíraný proud závisí na stavu, v jakém se čip a Wi-Fi momentálně nachází. V hlubokém spánku, kdy jsou aktivní pouze hodiny, může proud klesnout až na úroveň 20 µA. ESP8266EX je vybaveno sedmnácti digitálními GPIO piny, z nich většina má ještě další funkci, na což musí být myšleno při návrhu řešení. Jednotlivé piny mohou být nastaveny jako pull-up nebo pull-down. Pro analogové periferie je možné využít jeden deseti bitový ADC převodník. Ke komunikaci s dalšími čipy je možné využít SPI, I2C nebo UART. ESP8266 má k dispozici dvě SPI schopné fungovat jako master nebo slave s frekvencí 20 MHz. Dále je k dispozici I2C s frekvencí 100 kHz. Pro komunikaci s počítačem je možné využít UART schopné dosáhnout rychlosti 115200 b/s. [65]



Obr. Blokový diagram ESP8266EX [65]

Nejčastěji se dají ESP8266EX sehnat již umístěny na desce plošných spojů společně s anténou, oscilátorem a FLASH pamětí (viz Obr. 25 a Příloha 2). Krystal oscilátoru může mít frekvenci 40, 26 nebo 24 MHz. Moduly se vyrábí v několika provedeních lišících se rozměry, počtem vyvedených pinů a anténou (viz Obr. 26), z čehož jsou nejpopulárnější ESP-01 a ESP-12E. K jejich popularitě výrazně přispívá nízká cena, Wi-Fi a kompatibilita s Arduinem. [66, 67]

|  |  |
| --- | --- |
| Obr. ESP-WROOM-S2 [67] | ESP8266 NodeMCU Development Boards Versions  Obr. Verze modulů [66] |

Pro snazší použití jsou moduly připájeny k vývojovým deskám. Ty jsou vybaveny USB konektorem pro nahrávání kódu a napájení. Mezi nejoblíbenější patří NodeMCU využívající ESP-12E. Ten dává programátorovi k dispozici 4 MB FLASH paměti, ADC a jedenáct GPIO. K nahrání kódu do paměti jsou využívány čipy CP2101 nebo CH340. Příloha 1 vyobrazuje jaké funkce mají jednotlivé piny této desky. [66]

## Návrhové a architektonické vzory

Návrhové a architektonické vzory jsou léty ověřené techniky pro řešení opakujících se problémů v objektově orientovaném programování. Nejedná se o konkrétní kód, ale jen o koncept. Z tohoto důvodu nejsou svázány s konkrétní technologií a je tak možné je použít v téměř libovolném jazyce. Výhodou takto pojmenovaných a popsaných postupů je, že je zná většina vývojářů po celém světě a při komunikaci stačí říci jaký vzor použít, bez nutnosti vysvětlovat detaily. Tyto dvě skupiny se od sebe liší oblastí, kterou pokrývají. Návrhové vzory se zabývají chováním jedné třídy, nebo její komunikaci s ostatními. Oproti tomu Architektonické vzory určují sktrukturu celého projektu a mají přímý vliv na jeho modularitu a škálovatelnost. [68–70]

### Zapouzdření

Tímto pojmem je obvykle myšlen jeden za základních pilířů objektově orientovaného programování, kdy třída skryje své hodnoty a metody používané pro vnitřní fungování a ostatním přístupní jen ty potřebné ke komunikaci. Tento přístup také pomáhá zajistit konzistenci, protože stav objektu může být upraven pouze zamýšleným způsobem. Toto lze přenést i do většího měřítka, kdy je aplikace rozdělena na více zapouzdřených částí. Aby ostatní části mohli komunikovat nepotřebují znát vnitřní fungování, ale pouze rozhraní.[71]

### N-vrstvá architektura

Pro složitější aplikace, nebo tam, kde se očekává potřeba měnit některé celky, se často na základě pokrývané oblasti rozděluje aplikace na části označované jako vrstvy. Obvykle se každá vrstva nachází ve vlastním projektu. Hlavní výhodou je přehledná struktura, ve které se snáze hledá. V kombinace se zapouzdřením také zvyšuje modularitu a bezpečnost. Jelikož okolní vrstvy vidí pouze rozhraní, a nikoliv konkrétní implementaci je snadné vrstvu nahradit jinou bez ovlivnění ostatních. Komunikace je obvykle omezena na vrstvy o jednu pod a nad čili případný útočník nemůže z nejvyšší vrstvy přistupovat přímo k nejnižší. Rozdělení vrstev sebou však nese komplikaci v podobě komunikace mezi nimi.[71, 72]

Nejběžnější je třívrstvá architektura. Nejvyšší vrstva komunikuje s uživatelem a podle typu aplikace se jedná o uživatelské rozhraní, nebo v případě API o endpointy. Prostřední a nejdůležitější vrstvou je business logika, která zpracovává požadavky od uživatele. Poslední vrstva se stará o přístup k datům. Tím může být například zápis do databáze, nebo komunikace s jiným systémem.[71]

### Dependency injection

Dependency injection je technika, která snižuje závislost třídy na jiné. Toto umožňuje aplikaci být více modulární, lépe testovatelná a snáze upravitelná.[73]

Pokud má třída například zpracovat data a výsledek uložit do databáze, při klasickém přístupu je pevně svázána s konkrétním databázovým systémem. V horším případě obsahuje všechen kód, čímž porušuje Single responsibility principle (S ze SOLID)[74]. V lepším případě je práce s databází umístěna do vlastní třídy, ale její instance je součástí objektu s logikou, který je zodpovědný za jeho správu. Oba tyto případy komplikují přechod z jednoho typu databáze na jiný a testování je velice obtížné, protože kód očekává připojení k funkční databázi.[73]

Aby se těmto problémům předešlo, je instance této pomocné třídy, která je obvykle označována jako služba, předávána zvenčí. Nyní za správu služby není zodpovědný objekt s logikou, ale Injector. Dále třída většinou není závislá na konkrétní třídě, ale na rozhraní definující metody, které je možné zavolat. Díky této abstrakci je možné snadno změnit implementaci. Mimo jiné je takto umožněno místo skutečné implementace použít testovací třídu, která pouze simuluje volání databáze. Služba je nejčastěji vkládána pomocí konstruktoru, ale může být také použita metoda.[73]

Pro drobné projekty může jako injector sloužit prosté zavolání konstruktoru z kódu[73]. Ve většině případů je použit framework, který automaticky řeší vytváření a předávání potřebných instancí. Může se jednat o knihovnu třetí strany, nebo v některých případech přímo o systémovou knihovnu. Od verzí *.NET Core 1.0* a *.NET Framework 4.5* mezi tyto jazyky patří také C#[75]. V závislosti na typu projektu je knihovna již importována, nebo je třeba dodat příslušný NuGet. Při přidávání služby do seznamu je možné definovat životnost instance. První možností je *Transient*, který je při každém zavolání vytvořen nový. Druhou možností je *Singleton*, jehož instance je vytvořena jen jednou. Poslední je *Scoped* využívaný v ASP.NET pro situace, kde je potřeba aby každé zavolání API mělo vlastní instanci. Od .NET 8.0 je přidán atribut *FromKeyedServices* umožňující zaregistrovat více implementace jednoho rozhraní odlišených klíčem a zvolit implementaci podle aktuální potřeby.[76]

### Data Transfer Object (DTO)

Data transfer Object je instance třídy sloužící k přenosu dat mezi systémy. Použití speciálních objektů umožňuje skrýt hodnoty používané k vnitřní funkci jedné strany, ale pro druhou stranu zbytečných nebo jejichž přenos by mohl být bezpečnostní hrozbou. Současně je takto snížen objem dat, který je nutné přenášet. Další výhodou je možnost naráz přenést údaje nacházející se na více místech a uspořádat je do vhodné struktury. Tyto objekty slouží k serializaci a deserializaci a neměli by obsahovat žádnou logiku.[77, 78]

### MVVM

Pro jednodušší vývoj a testování uživatelských rozhraní se využívají návrhové vzory MVC, MVP a MVVM. Všechny tři od sebe oddělují data, vzhled a logiku, čímž usnadňují udržení struktury a umožňují modulárnost aplikace. Liší se v datových tocích a závislostech jedné části na ostatních.[79]

Nejstarším z těchto návrhových vzorů je MVC (Model-View-Controller). Model obsahuje aplikační data a je zodpovědný za komunikaci s databází, serverem, či jinou externí částí aplikace. View má na starosti zobrazování dat z modelu uživateli. Controller reaguje na uživatelské akce a dává modelu a view pokyny k aktualizaci. Jak je vidět na Obr. 27 jednotlivé části jsou úzce provázány, což komplikuje testovatelnost a úpravy.[79, 80]



Obr. Datový tok MVC [79]

Většinu problémů MVC řeší MVP (Model-View-Presenter), kde view a model nekomunikují napřímo, ale přes presenter jako prostředníka (viz Obr. 28). Oproti MVC zde na uživatelské akce reaguje view, které informaci předává presenteru. Ten při vracení aktualizovaných dat z modelu může provést další zpracování. Díky většímu oddělení jednotlivých částí usnadňuje testování a úpravy.[79, 80]



Obr. Datový tok MVP [79]

Vzor MVVM (Model-View-ViewModel) je podobný MVP, ale view neobsahuje žádnou logiku a pouze vykresluje data, která dostane z viewModelu. Svůj obsah aktualizuje na základě eventu OnPropertyChanged (viz Obr. 29). Většina logiky se nachází ve viewModelu, který má také na starosti stav aplikace. Tento přístup umožňuje, aby více view bylo navázáno na jeden viewModel. Oproti svým předchůdcům je MVVM modulárnější, testovatelnější a snáze škálovatelný. Avšak za cenu vyšší komplexity tříd.[79, 80]



Obr. Datový tok MVVM [79]

# Vlastní řešení

Praktickou částí této práce je návrh a realizace řešení, které by umožnilo uživatelům s minimální či žádnou znalostí programovacích jazyků vytvořit automatizovanou úlohu. Řešení je navrženo tak, že existuje jeden či více hlavních uzlů, které vykonávají zadanou úlohu. Jako vstupy slouží data z uzlů se senzory. Jeden uzel může být součástí více úloh, tudíž iniciátorem komunikace je hlavní uzel (Obr. 30).



Obr. Sekvenční diagram: obecná komunikace s více hlavními uzly

Při realizaci je využívána abstrakce a není spoléháno na funkce specifické pro použité technologie, díky čemuž je snazší případná migrace. Dále je kladen důraz na modularitu, aby v případě že aktuální řešení nevyhovuje potřebám konkrétní úlohy bylo snadné danou část jednoduše nahradit bez ovlivnění zbytku systému.

Pro komunikaci byl zvolen protokol HTTP (viz Kap. 3.2.5) s obsahem ve formátu JSON (viz Kap. 3.3.2). Jelikož pro HTTP existuje mnoho nástrojů je jednoduché otestovat funkčnost uzlu. Dále je možné využití uzlů i mimo tento projekt. Z těchto důvodů byl upřednostněn textový formát před bitovým (viz Kap. 3.3), který vyžaduje funkční program schopný reprezentovat přijatá data. Z běžně využívaných formátů CSV vyžaduje tabulku, kde každý záznam musí mít stejný počet sloupců, a XML musí mít pro každou hodnotu otevírací a ukončovací značku, čímž zvětšuje požadavky na množství přenesených dat. Z tohoto důvodu jsou značkovací jazyky vhodné pro složité struktury psané programátorem (například vzhled uživatelské rozhraní), ale pro tuto situaci se více hodí JSON.

Z důvodů jako jsou například množství přenášených dat a princip fungování sensorů mohou mít uzly rozdílný počet endpointů. Proto všechny uzly mají endpoint *getInfo*, který vrátí kolekci s informacemi o dostupných endpointech, jako jsou URL, HTTP metoda, zda pouze vrací hodnoty, nebo je také nastavuje, jaké hodnoty vrací a zda očekává argumenty. V případě pomalých získávání hodnot také může obsahovat údaj o očekávaném zpoždění, aby hlavní uzel věděl, kdy ještě probíhá zpracování a kdy již uběhl časový limit znamenající problém se spojením. U vracených hodnot sdělí jejich název a datový typ. Argumenty navíc obsahují výchozí hodnotu a limity v jakých se může hodnota pohybovat. Proto je tento enpoint volán, když uživatel přidává nový uzel do systému, aby při zadávání logiky věděl, s jakými hodnotami může pracovat. Pro ověření funkčnosti hlavní uzel zkusí, zda je možné všechny zavolat. Tento postup sice brání přidání uzlů bez funkčního spojení, ale odchytí případné problémy již na začátku. Z kontroly jsou vynechány endpointy označené jako výstupní, aby změna hodnoty neměla nežádoucí účinky na systém. Tento postup je znázorněn na Obr. 31 pomocí sekvenčního diagramu.



Obr. Sekvenční diagram: přidání uzlu

## Hlavní uzel

Hlavní uzel je realizován jako počítačový program. Řešení je rozděleno na tři části (viz Kap. 3.6.2), které řeší komunikační, logickou a uživatelskou vrstvu. Každá vrstva má referenci jen na vrstvu pod ní. Toto řešení umožňuje snadnou změnu jednotlivých částí s minimálními zásahy do kódu.

Pro realizaci byl zvolen C#, jakožto hlavní programovací jazyk .NET ekosystému, který umožňuje knihovny využít na většině současně využívaných operačních systémech. Jako verze byla zvolena .NET 8.0, protože v době psaní této práce se jednalo o jedinou LTS (Long Term Support). Dále také podporuje *KeyedServices* (viz Kap. 3.6.3) umožňující snáze implementovat více typů uzlů. Tato volba je na úkor podpory starších operačních systémů (např. Windows 7 a verze Windows 10 starší než 21H2) a verzí programů třetích stran, které vyšly před listopadem 2023 [81–83]

Uživatelem zadaná logika je reprezentována jako kolekce generických objektů obsahujících operaci a instanci objektu reprezentujícího hodnotu uzlu. Jelikož se hodnota předává uvnitř objektu, je možné ji dynamicky měnit během vykonávací smyčky.

### Komunikační vrstva

Úkolem této vrstvy je zajistit komunikaci s uzly a poskytnout abstrakci pro ostatní vrstvy, aby ke své funkci nemuseli znát podrobnosti o způsobu komunikace. Tento způsob umožňuje v budoucnu nahradit HTTP jiným protokolem nebo využívat více protokolů či API třetích stran současně.

Vrstva je realizována jako knihovna tříd nazvaná *MainNode.Communication*. Většina tříd, které se zde nacházejí jsou DTO (viz Kap. 3.6.4). Ty jsou v následující vrstvě, odkud byla metoda, jenž je vrací volána, převedena na DO.

#### DTO třídy

Pro komunikaci s uzly slouží třídy plnící funkci DTO (viz Kap. 3.6.4), jenž se nacházející v namespace *MainNode.Communication.Dto*, a enumy, které se nacházejí v namespace *MainNode.Communication.Enums*. Ty jsou zobrazeny pomocí diagramu tříd na Obr. 32.



Obr. Diagram tříd MainNode.Communication.Dto a MainNode.Communication.Enums

Třída *ValuesDto* slouží k přenosu hodnot mezi Hlavním uzlem a ostatními uzly. Obsahuje pole pro hodnoty každého podporovaného datového typu. Ačkoliv se nejedná o nejintuitivnější řešení, byl tento přístup zvolen za účelem snížení množství přenášených dat při současném umožnění delších sebevysvětlujících názvů hodnot. Jelikož data v objektu reprezentující hodnotu na vyšších vrstvách musí být aktualizována, aniž by byl stávající objekt nahrazen, jsou možnosti buď hledat v kolekci posle jména, nebo spoléhat na jejich pořadí. Z důvodu rychlosti bylo upřednostněno pořadí před vyhledáváním, jelikož pravděpodobnost jeho změny je velice nízká.

Třída *ValueDto* obsahuje informace o jednotlivých hodnotách endpointu. Skládá se z názvu a enum *ValType*, jenž udává datový typ. Potomek *ValueArgDto* kromě těchto dvou údajů má navíc výchozí hodnotu a horní a dolní limit. Pro abstrakci protokolu je cesta k endpointu reprezentována pomocí rodičovské třídy *EndPointPath* a jejího potomka *HttpEndPointPath*, jenž kromě stringu s cestou obsahuje také HTTP metodu daného endpointu.

#### INodeCommunication

*INodeCommunication* je rozhraní pro třídu řešící komunikaci. Obsahuje adresu uzlu, se kterým bude komunikovat, a o jaký typ adresy se jedná, což je využito při deserializaci uloženého seznamu uzlů. Tato hodnota je nastavena instancí, jenž toto rozhraní implementuje. Metoda *Init* slouží k inicializaci objektů potřebných k samotné komunikaci. Tato logika nemohla být umístěna do konstruktoru, protože kvůli využití dependency injection (viz Kap. 3.6.3) je instance v některých situacích vytvořena dříve, než je známa adresa. Asynchronní Metoda *GetEndPoints* složí k zjištění seznamu dostupných endpointů (první zpráva mezi hlavním uzlem a uzlem na Obr. 31) a vrací kolekci *EndPointDto*, která může být *null*. Jak již bylo řečeno jedná se o surová data, která jsou zpracována až na vyšší vrstvě. Asynchronní metoda *GetValues* slouží ke komunikaci s uzly. Návratovou hodnotou je *nullable* objekt typu *ValuesDto*. Povinným parametrem je instance třídy *EndPointPath*, či jejich potomků. Dále má nepovinné celé číslo *delay* a instanci *ValuesDto*, jenž jsou obsaženy v objektu, který reprezentuje endpoint ve vyšší vrstvě a byl vytvořen na základě hodnot načtených pomocí *GetEndPoints.*



Obr. Diagram tříd INodeCommunication

*HttpNodeCommunication* je implementací (viz Obr. 33) právě popsaného rozhraní, které slouží ke komunikaci pomocí HTTP protokolu a dříve popsaných DTO objektů. K tomu využívá instanci třídy *System.Net.Http.HttpClient*, jenž je vytvořena v metodě *Init* s hodnotou *Timeout* nastavenou na deset hodin, aby bylo zajištěno, že bude delší nejpomalejší endpoint. Bylo vycházeno z předpokladu, že uzly budou posílat hodnoty v intervalech které se pohybují v řádek minut, až milisekund.

Implementace *GetValues* rozlišuje, zda volaný endpoint využívá *GET* nebo *POST* a zavolá podle toho pomocnou metodu. Jedná-li se o *GET* přidá argumenty do URL adresy. V případě *POST* provede serializaci. V obou případech je před odesláním HTTP dotazu vytvořena instance třídy *System.Threading.CancellationTokenSource*, jenž v případě, že metoda nestihne nedoběhnout dříve, než uplyne stanovený časový limitu, vyhodí *System.Threading.Tasks.TaskCanceledException*. Ta je zachycena a volání je vyhodnoceno jako chyba spojení. K nastavení časového limitu je využit parametr *delay*. Pokud pro daný endpoint není definován je použita výchozí hodnota 1 s. Tento postup je zvolen, protože *Timeout* může být nastaven pouze jednou, takže by bylo nutné vytvořit samostatného klienta pro každý endpoint. Pokud je volání úspěšné, je odpověď deserializována a vrácena.

#### CommunicationTypeResolver

Jedná se o pomocnou statickou třídu sloužící k vytvoření instance správné implementace rozhraní *INodeCommunication*. K tomu je využívána metoda *GetCommunicationType*, jenž má jako parametr textový řetězec, na jehož základě je rozhodnuto, jakou instanci je třeba vytvořit.

### Logická vrstva

Logická vrstva je jádrem této práce, neboť zde probíhá zpracování uživatelem zadané logiky. Ta je zadávána v podobě textu, aby byl proces stejný bez ohledu na to, zda je vyšší vrstva uživatelské rozhraní, nebo ASP.NET aplikace sloužící jako API komunikující s jinou aplikací. Tím je minimalizováno riziko, že by implementace vytvářela neočekávané stavy, protože pokud programátor, který vytváří vyšší vrstvu neobejde třídu na zpracování vstupu, budou neočekávané operace zachyceny. Je-li uživatelský vstup bez chyb, je vytvořena kolekce datových toků tvořených operacemi obsahujícími objekty reprezentující data z uzlů. Když uživatel spustí smyčku, jsou na začátku každé iterace aktualizovány hodnoty využitých endpoitů. Po jejich aktualizaci je provedeno vyhodnocení datových toků. Na konci smyčky jsou výsledky odeslány na příslušné endpointy. Po uplynutí nastaveného času je spuštěna další iterace smyčky.

Vrstva je realizována jako knihovna tříd nazvaná *MainNode.Logic*. Pro využití ve vyšší vrstvě má definována rozhraní, aby se snížila šance nesprávné manipulace. V této vrstvě jsou využívány repositáře, aby byla data na jednom místě a předešlo se tak více instancím, které mají reprezentovat stejnou informaci, ale mají odlišné hodnoty.

#### DO třídy

DO na této vrstvě jsou reprezentací DTO z komunikační vrstvy (viz 4.1.1.1), které využívají generiku, aby mohli obsahovat všechny podporované datové typy a poskytnou ostatním objektům přímo danou hodnotou. K převodu DTO na DO slouží statická třída *Mapper* obsahující metody, které se shodně jmenují *Map* a pro DTO parametr vrací jeho ekvivalentní instanci DO. Tento přístup minimalizuje závislost logické vrstvy na komunikační a v případě změny je třeba upravit pouze třídy *Mapper* a *Node*, který s touto vrstvou komunikuje (viz Kap. 4.1.2.3).



Obr. Diagram tříd ValueDo a potomci

Základem je abstraktní třída *ValueDo* (viz Obr. 34) a její stejnojmenný generický potomek reprezentující hodnotu uzlu. Abstraktní rodič je potřebný z důvodu, aby bylo možné tuto třídu použít jako parametr metod a vlastnost ostatních tříd i v situacích, kdy v době kompilace není možné určit datový typ hodnoty. Abstraktní rodič obsahuje pouze jméno *Name* a deklaraci metody *GetT*, která vrací generický datový typ. Potomek obsahuje definici oné metody a generickou vlastnost *Value*, obsahující již zmíněnou hodnotu. Pro účely výpisu a debugu obsahuje přetíženou metodu *ToString* vypisující celé jméno generického typu společně s jménem a hodnotou. Dále obsahuje zkrácenou verzi *ToStringShort* vypisující pouze krátký název typu a jméno. Od generické *ValueDo* dědí *ValueArgDo*, která přidává pouze vlastnosti *Default*, *Min* a *Max*, jenž jsou stejného datové typu jako *Value*.



Obr. Diagram tříd EndPointDo

Třída *ValuesDo* (Obr. 35) obsahuje informaci o aktuálních hodnotách endpointu. K tomu jí slouží listy *ValueDo<int>*, *ValueDo<float>* a *ValueDo<bool>*. Při načtení nových dat je aktualizována hodnota *Value* všech elementů v těchto kolekcích. Pro účely výpisu se zde nachází metoda *ToStringListShort*, jež zavolá *ToStringShort* nad každým elementem ve výše zmíněných kolekcích a jejich výsledky spojí do listu textových řetězců.

Třída *EndPointDo* (Obr. 35) uchovává informace o jednotlivých endpointech uzlu. Cesta potřebná k jeho zavolání je uložena ve vlastnosti *Path* typu *EndPointPath*, která slouží jako abstrakce komunikační vrstvy (viz Kap. 4.1.1.1). hodnoty a argumenty jsou reprezentovány instancemi třídy *ValuesDo*. Vlastnosti *Type* a *Delay* jsou identické s *EndPointDto*.

#### ConnectionStatus

Třída *ConnectionStatus* (viz Obr. 36) slouží k sledování stavu spojení s uzlem. K jeho reprezentaci slouží *ConnectionStatusEnum* jenž může nabývat hodnot *GOOD*, *WITH\_PROBLEMS*, *LOST*, *RECOVERING* a *UNKNOWN*. Při vytváření instance pomocí konstruktoru jsou nastaveny bezznaménkové celočíselné hodnoty *maxFails* a *recoveryThreshold*. První je počet neúspěšných pokusů v řadě, kdy se ze spojení s problémy stává ztracené spojení. Druhý je kolikrát v řadě musí být volání úspěšné, aby se ze zotavujícího spojení stalo opět dobré. V případě, že neexistují žádané informace o těchto hodnotách je stav neznámí, což nastává jen v případě načtení uloženého uzlu. Dojde-li k chybnému spojení, je do listu typu *ErrorData* přidán nový objekt, jenž obsahuje informaci o času, volaném endpointu a z jakého důvodu k chybě došlo.



Obr. Diagram tříd ConnectionStatus

#### Node



Obr. Diagram tříd Node

Třída Node (viz Obr. 37) obsahuje veškeré informace o uzlu a slouží jako prostředník mezi logickou a komunikační vrstvou. Toho je dosaženo pomocí private instance rozhraní *INodeCommunication* (viz Kap. 4.1.1.2). Vlastnost *Name* slouží k identifikaci při zadávání logiky a pokud ho uživatel nezmění, jeho výchozí hodnotou je node a pořadové číslo. Aby bylo možné odesílat dotazy, je třeba adresa uložená v proměnné *\_address* k niž je přistupováno pomocí vlastnosti *Address*. Při čtení pouze vrátí hodnotu, ale při zápisu současně zavolá metodu Init z komunikačního objektu. Adresa je textový řetězec, protože se jedná datový typ využívaný většinou tříd pro komunikaci. Ať už se jedná síťovou komunikaci s IP adresou a portem, sérovou linku s číslem portu, volání jiného procesu, databázi s ConnectionStringem nebo další. S adresou souvisí *AddressType*, jenž je také kombinací proměnné a vlastnosti. Ten je využíván při deserializaci aby bylo možné zjistit, která implementace *INodeCommunication* je potřeba vytvořit. Seznam všech endpointů dostupný pro uzel je uložen v kolekci *EndPoints*. Pro reprezentaci stavu spojení slouží instance třídy *ConnectionStatus*, která je vyřazena ze serializace, neboť nemá význam ukládat aktuální stav spojení, neboť při příštím spuštění bude situace zcela odlišná.

Třída Node má dva konstruktory. První má parametr *INodeCommunication*, využívající dependency injection (viz Kap. 3.6.3), volaný při přidávání nového uzlu, kdy je na základě typu implementace nastavena hodnota *AddressType*. Druhý je bezparametrický a je využívaný při deserializaci, kdy není možné volat konstruktor s parametry.

Asynchronní metoda *GetEndPoints* slouží k naplnění kolekce endpointů pomocí zavolání stejnojmenné metody v implementaci *INodeCommunication* a s využitím statické metody Mapper.Map přemapováním vrácených DTO, využívaných v komunikační vrstvě na DO, která jsou používány v logické vrstvě.

Asynchronní metoda *GetValues*, jejíž parametr je právě volaný endpoint, slouží k aktualizaci hodnot. Na začátku je zavolána *INodeCommunication.GetValues* vracející *ValuesDto* obsahující kolekce hodnot rozdělené podle datových typů. Pro každý list hodnot v *EndPointDo* je zavolána rozšiřující metoda *CopyValues*, která nahradí původní hodnoty novými. Tento postup je použit, neboť kvůli zachování referencí nemůže být objekt nahrazen novým. Pokud vše proběhne v pořádku, je v objektu *ConnectionStatus* pomocí metody *Success* komunikace označena za úspěšnou. V opačném případě je zavoláno *Failure*, čímž je aktualizován počet neúspěšných spojení a současně je zaevidován důvod proč se nezdařila. GetValues Je využívána jak pro vstupní, tak výstupní endpointy.

Asynchronní metoda *GetAllValues* slouží k ověření dostupnosti všech vstupních endpointů. Toho je dosaženo tak, že ze seznamu vybere ty, jejichž *Type* má hodnotu *EndPointType.GET*. ty jsou následně pomocí *GetValues* zavolány.

Pro volání pomalých endpointů, tedy těch, jenž mají nenulovou vlastnost *Delay*, slouží asynchronní metoda *ParalelCall*. Stejně jako jeho obdoba pro běžné volání využívá *INodeCommunication.GetValues*, ale neprovádí aktualizaci hodnot, aby se předešlo situaci, kdy by část výpočtů iterace proběhla se starými a část již s novými hodnotami.

#### EndpointVariables



Obr. Diagram tříd EdpointVariables

Účelem této třídy (Obr. 38) je sledovat aktualitu hodnot a sledovat stav načítání v případě endpointů na jejichž odpověď se čeká jednu či více iterací vyhodnocovací smyčky. Dále je takto nevolat endpointy jenž jsou v uzlu sice dostupné, ale k řešení požadované úlohy jsou nepotřebné a jejich volání by pouze prodloužilo načítací proces, zvýšilo množství dat, které je potřeba přenést a uzel by musel vynaložit energii na zpracování dotazu, což je problematické obzvláště v případě bateriového provozu.

Aby bylo vůbec možné hodnoty aktualizovat jsou potřebné reference na instance Node a EndPointDo. Pro sdělení informace o aktuálnosti hodnot slouží booleovská vlastnost *IsActual*, která je před zahájením načítání pomalého volání nastavena na false. Teprve poté co jsou hodnoty aktualizovány je nastavena na true. Podobnou funkci má *Loaded*, avšak zde je hodnota změněna již v okamžiku dokončení načítání, k čemuž dojde ještě během iterace vyhodnocovací smyčky. Opačný význam má *Loading* znamenající, že načítání teprve probíhá. Tyto dvě hodnoty jsou na sobě nezávislé, protože kdyby byla použita pouze jedna nebo by se jednalo o inverzi, tak by v okamžiku po spuštění smyčky mohlo docházet k nečekaným stavům, jelikož by nebylo možné dosáhnout stavu, kdy hodnota není načítána a současně není dokončeno získávání nové.

Pro načítání hodnoty pomalých endpointů je využívána asynchronní metoda *Load* volající *Node.ParalelCall*. V momentě, kdy je načítání dokončeno, ale čeká se do konec aktuální iterace vyhodnocovací smyčky, jsou nové hodnoty uloženy v proměnné typu *ValuesDto*, odkud budou během příští fáze načítání zkopírovány stejně jako je tomu bezprostředně po načtení u klasických endpointů využívající metodu Node.*GetValues* volanou stejnojmennou metodou této třídy.

#### Třídy datového toku

Aby bylo možno dynamicky měnit vyhodnocovanou logiku na základě uživatelského zadání, je nutné vytvořit objekty obsahující jak požadované operace, tak hodnoty, se kterými se mají provádět. Tuto funkci plní generická třída *Operation* a její potomci (viz Obr. 39). Pro reprezentaci prováděné operace je využíván delegát *System.Func*, jenž má dva parametry. Hodnota pro výpočet je uložena buď ve vlastnosti *Const* nebo *Ref*, v závislosti na tom, zda se jedná o konstantu nebo referenci typu *ValueDo*. Obě tyto hodnoty jsou nullable a nastavovány pomocí konstruktoru. Výsledek je počítán pomocí generické metody *Execute*, jenž jako parametr přijímá hodnotu nalevo od znaménka (Obr. 40).



Obr. Diagram tříd Operation



Obr. Kód metody Execute

Základní třída očekává, že trojice výsledek, levá a pravá hodnota jsou stejného datového typu. V závislosti na tom, která z vlastností je null dosadí hodnoty do delegáta a vrátí výsledek. Její stejnojmenný potomek má stejné chování, ale pravá hodnota od znaménka je jiného datové typu. Aby nedocházelo k nejasnostem, zda využít vlastnost rodiče nebo potomka, je využita schopnost jazyka C# přidáním klíčového slova *new* v definici překrýt stejnojmennou vlastnost rodiče.

K vytvoření datového toku hodnot slouží instance generické třídy *Flow* (Obr. 41) obsahující list operací. Aby bylo možné uložit všechny datové toky do jedné kolekce je nutné vytvořit abstraktního negenerického předka. Pro umožnění vytvoření instance na základě proměnné typu *System.Type* je v této třídě vytvořena statická metoda, jež na základě na této hodnoty vrátí generickou instanci. Každá instance *Flow* má jméno ve vlastnosti *Name*, aby bylo možné se na ní odkazovat při zadávání logiky. K vyhodnocení celého datového toku slouží generická metoda *Evaluate*. Ta na začátku vytvoří výchozí hodnotu datového typu tohoto toku, který slouží jako levá hodnota první operace, a uloží ji do proměnné pro výsledek. Poté projde celou kolekci a pro každý prvek zavolá metodu *Execute* s aktuálním výsledkem jako levou hodnotou. Po projetí celé kolekce vrátí hodnotu, jenž se aktuálně nachází v proměnné s výsledkem. Toto řešení sebou nese nutnost používání závorek pro určení pořadí operací. Pokud bude uživatel potřebovat tuto knihovnu implementovánu tak, že bude potřeba získat pouze hodnotu, je možné využít právě popsanou metodu *Evaluate*, ale je počítáno s variantou, kdy jeden datový tok je součástí dalšího a k tomu slouží metoda *Run*. Ta takto vypočtený výsledek vloží do *ValueDo* vlastnosti *Output*, kterou je možné použít jako referenci. Aby bylo možné sledovat, zda v této iteraci vyhodnocovací smyčky již byl výsledek datového toku vypočítán, je využívána třída *FlowResult* (Obr. 42), jejíž instance je vytvářena pomocí metody *GetResult*.



Obr. Diagram tříd Flow



Obr. Diagram tříd FlowResult

Stejně jako v případě *Flow* je *FlowResult* generická třída s negenerickou abstraktní rodičovskou třídou. Booleovská hodnota Finished vyjadřuje, zda v této iteraci již byl provedeno vyhodnocení datového toku. Jeho výsledek je uložen v proměnné. Pokud při zavolaní konstruktoru *Flow*, pro který je instance vytvářena nemá nastavenou instanci *ValueDo*, kam bude vkládat výsledek, je vytvořena nová, jenž má stejné jméno jako datový tok, ale na konec je doplněno \_out. V opačném případě je pouze uložena reference na tento objekt. K získání výsledku slouží vlastnost *Value*, která obsahuje pouze get. Pokud v této iteraci již došlo k vyhodnocení, je rovnou vrácen výsledek. V opačném případě je provedeno vyhodnocení. Poté jsou nastaveny hodnoty *Finished*, *IsActual* a *\_lastRun*. Poslední dvě zmíněné jsou využity v případě, kdy je pomocí vlastnosti *RunFrequency* nastaveno, aby vyhodnocení bylo prováděno jednou za určitý časový úsek. V takovém případě IsActual slouží jako indikátor, že výsledek nepochází z této iterace. Na začátku nové iterace vyhodnocovací smyčky je zavolána metoda *NewIteration*, která na základě aktuálního času, *\_lastRun* a *RunFrequency*, jejíž výchozí hodnota je 0 µs, nastavuje hodnoty *IsActual* a *Finished*. V případě že datový tok je současně výstupem systému, je použita metoda *BindOutput*, jenž nahradí vlastnost *Flow.Output* hodnotou endpointu.

Pro využití výsledku z datového toku v jiném slouží třída *SubFlowOperation*, která je potomkem *Operation*. její konstruktor přijímá *FlowResult* a *System.Func*, jehož levá hodnota a výsledek musí být stejného datového typu. Při výpočtu jako hodnotu na pravé straně operátoru využívá *FlowResult.Value*.

Speciálním případem je *MergeFlowOperation* sloužící k sloučení výsledků dvou datových toků. Oproti ostatním potomkům *Operation* výsledek, levá a pravá strana mohou být rozdílných datových typů. Ačkoliv metoda *Execute* má parametr, je zde pouze kvůli dědičnosti, ale k výpočtu není využit. Aby bylo možné tento výsledek použít pro výpočet, je potřeba současně použít instanci třídy *FlowMerge*, jenž je potomkem *FlowResult*. V konstruktoru je vytvořena na základě dvou *FlowResult* a *System.Func* vytvořena instance *MergeFlowOperation*. Vlastnost *Value* vypadá téměř totožně jako v rodičovské třídě, ale místo datového toku je zde vyhodnocována operace.

#### NodeRepository

Třída *NodeRepository* implementující rozhraní *INodeRepository* (viz Obr. 43), čímž je snížena závislost vyšší vrstvy na konkrétní implementaci, slouží jako globální úložiště instancí třídy *Node*. Očekává se, že tato třída bude využívána jako singleton. Jádrem této třídy je kolekce *Nodes*, obsahující seznam všech uzlů připojených do systému. Pro přidání nového uzlu slouží asynchronní metoda *AddNode*, jejímž parametrem je přidávaný uzel. Než dojde k přidání, je ověřena nenulovost adresy a jména. Jelikož je jméno využíváno jako identifikátor při zadávání logiky, musí být unikátní. Pokud je některý z těchto požadavků nesplněn, je vyhozena výjimka *System.ArgumentException* s odpovídajícím chybovou zprávou. V opačném případě je zavolána metoda *GetEndPoints*, aby bylo možné získat seznam dostupných endpointů. Po jejím dokončení je pomocí *GetAllValues* ověřena jejich dostupnost (viz Kap. 4.1.2.3). V případě, že je seznam prázdný, je vyhozena výjimka *MainNode.Exceptions.NoEndPointException*, protože nemá smysl přidával uzel, který nejde zavolat. Pouze pokud nedošlo k žádnému problému je uzel přidán a změněna hodnota počítadla. Jelikož je počet uzlů použit jako výchozí název uzlu, jedná se o statickou hodnotu, jenž není součástí instance. Toto je z důvodu předcházení cyklických referencí.

Pro Uložení seznamu slouží metoda *SaveNodes*, vracející serializovanou kolekci uzlů. Třída neřeší ukládání sama, protože *MainNode.Logic* je knihovnou a může být implementována v různých typech aplikace včetně webových. Z tohoto důvodu je zde řešena pouze serializace a deserializace, ale práce s perzistentním úložištěm je přenecháno vyšší vrstvě. Obdobně metoda *LoadNodes* má jako parametr textový řetězec ve formátu JSON. Po deserializaci na *List<Node>* se pokusí pomocí metody *AddNode* tyto uzly přidat. V případě neúspěchu je přidá do *Dictionary<Node, string>*, který je návratovou hodnotou této metody. Klíčem záznamu je přidávaný uzel a hodnotou je chybová hláška, proč se přidání nezdařilo.



Obr. Diagram tříd NodeRepository

#### FlowRepository

Třída *FlowRepository* implementující rozhraní *IFlowRepository* (viz Obr. 44) uchovává všechny datové toky, vstupy a výstupy potřebné k realizaci dané úlohy na jednom místě. Jednotlivé datové toky jsou ve formě *FlowResult* (viz Kap. 4.1.2.5) uloženy v kolekci *Results*. Pro uložení vstupních a výstupných hodnot slouží kolekce Inputs a Outputs, jenž jsou typu *Dictionary<EnpointLoadTypeEnum, List<EndpointVariables>>*. Klíčem je enum určující, zda se jedná o klasický endpoint, nebo o pomalý, jehož odpověď může trvat několik iterací vyhodnocovací smyčky. Hodnotami jsou kolekce *EndpointVariables* (Kap. 4.1.2.4) řešící aktualizaci hodnot.



Obr. Diagram tříd FlowRepository

K přidání nového datového toku slouží metoda *AddFlow* s parametrem typu *Flow*, která pomocí *GetResult* vytvoří instanci *FlowResult* a přidá ji do kolekce. Aby bylo možné využít tuto hodnotu jako referenci pro další výpočet, je tato instance současně také vrácena. Pro přidání vstupů a výstupů slouží metody *AddInput* a *AddOutput*, které jsou téměř identické, ale pracují s jinými kolekcemi. Podle toho, zda je hodnota *Delay* *null* či nikoli, je instance přidána do seznamu normálních nebo pomalých volání.

Pro získání požadovaného datového toku je využívána metoda *GetFlowByName*, jejímž parametrem je jméno tohoto toku. Pokud nebyl nalezen žádný výsledek, nebo naopak je jich více, je vyhozena výjimka System.Exception s příslušnou chybovou hláškou.

Tato třída také řeší provedení vyhodnocovací části iterace smyčky. Slouží k tomu metoda *Run*, která nejprve na všech prvcích zavolá metodu *NewIteration*, čímž dojde k vynulování příznaků, zda v této iteraci již bylo provedeno vyhodnocení. Poté ze všech prvků načte hodnotu vlastnosti *Value*. Pokud ještě nebyl datový tok vyhodnocen, tak k tomu dojde. Byl-li vyhodnocen při získávání některého předcházejícího, je pouze vrácen výsledek.

#### Zpracování uživatelem zadané logiky

Pro zpracování uživatelem zadaného textového řetězce do podoby, kterou je možné vyhodnotit pomocí výše popsaných tříd, byl zvolen deterministický konečný automat. Kombinace stavů jsou reprezentovány pomocí matice. Jelikož při návrhu bylo počítáno s možností v budoucnosti převést toto řešení na jednočipový počítač, které oproti klasickému osobnímu počítači má výrazně méně paměti, byla tomu přizpůsobena struktura. Oproti běžně využívanému řešení, kdy každý znak abecedy automatu má vlastní sloupec (nebo řádek v závislosti na způsobu zápisu), což by v tomto případě, při využití pouze ASCII znaků, vyžadovalo dva tisíce osm set šestnáct možných kombinací, byly znaky se stejným významem seskupeny (např. všechna písmena) a nepotřebné zcela vynechány. Tím se počet možných přechodů snížil na sto čtyřicet tři. Ačkoli je toto řešení v době provádění náročnější na výkon, je ušetřena zbytečně zabraná paměť, což vhledem k tomu, že některá vývojová prostředí pro jednočipové počítače omezují velikost programu na 32 kB nebo méně, je výrazný rozdíl. Pro snížení počtu stavů je využíván zásobník. Na Obr. 45 je diagram tříd, které tento stavový automat a zásobník reprezentují.



Obr. Diagram tříd stavový automat

Zásobník obsahuje objekty typu *StackValue* skládající se z dosud přečtených znaků, StackValueTypeEnum a cache. Enum určuje o jakou hodnotu se jedná (např. konstanta, jméno, uzlu, ...). Pro zaznamenání znaků je využit System.Text.StringBuilder, jenž je schopný dynamicky rozšiřovat svůj obsah bez zbytečného kopírování paměťových bloků, čímž je výrazně ušetřen čas procesoru. Cache je typu *nullable Object* a je využita v případě, že je z přečteného slova je možné vytvořit objekt, ale bez znalosti následujícího ho není možné dále zpracovat. Takovým případem může být přečtení hodnoty ednpointu, ale neznalost operátoru a druhé hodnoty potřebné k vytvoření operace.

Matice reprezentující stavoví automat je dvourozměrné pole *TransitionFunc*, jehož indexy jsou právě čtený znak převeden na číslo pomocí metody *getId* a enum *LCStateEnum* reprezentující stavy (viz Příloha 4 obsahující tabulku přechodů). Třída *TransitionFunc* reprezentuje přechodovou funkci a obsahuje následující stav, delegáta *System.Action<char, LCStateEnum, StackValueTypeEnum?>*, jenž bude proveden při přechodu do nového stavu a jaký typ záznamu bude přidán do zásobníku. Parametry delegáta jsou aktuální znak a stav společně s typem záznamu, jenž je součástí objektu reprezentující přechodovou funkci. Tato hodnota je potřebná, protože metody, na něž se delegáti odkazují, nejsou součástí této třídy.

Delegáti *System.Func* používaní v instancích *Operation* jsou bráni z *FuncRepo*, kde se nachází v *Dictionary<(Type, Type, string), Delegate>*, jenž je naplněn v konstruktoru. Klíčem je trojice datových typů obou hodnot a textové podoby operátoru. Třetí hodnotou je textový řetězec místo znaku, kvůli logickým operátorům, jenž jsou tvořeny dvěma znaky. Hodnota je definována pomocí lambda výrazů. K přistupování ke kolekci slouží metoda *GetFunction*, která v případě neexistujícího klíče vyhodí výjimku se zprávou obsahující informaci, která trojice nebyla nalezena.

Samotný převod logiky z textového řetězce na datové toky probíhá v instanci třídy *LoopCompiler* (viz Příloha 3), která je pro vyšší přehlednost rozdělena na více souborů, jenž mezi názvem třídy a koncovkou mají jakou část převodu řeší. V konstruktoru je zavolána metoda *InitTable*, která naplní tabulku přechodových funkcí. Pro zahájení převodu je zavolána metoda *Compile*, která zpracuje jeden datový tok. Ten může obsahovat další vnořené toky ohraničené závorkou, ale na ně se uživatel nebude moci odkazovat. Aby bylo možné zadat více datový toků, je nutné použít metodu *CompileMultiLine*, která textový řetězec rozdělí podle středníků a nových řádků. Dojde-li k chybě vyhodí *System.ApplicationException* se zprávou obsahující index datového toku a text výjimky, kterou současné vloží jako *innerException*, aby bylo možné dohledat příčinu jejího vzniku. Na začátku kompilace je vymazán zásobník a na jeho vrchol je vložen inicializační operátor, neboť kvůli principu vyhodnocování (viz Kap. 4.1.2.5) musí být první operací ekvivalent *0+b*. Poté se projde pomocí smyčky for celý textový řetězec. For byl zvolen místo foreach proto, aby bylo možné v chybové hlášce přesně určit místo, kde nastal problém. Na základě indexu vráceného metodou *getId* a číselného vyjádření *LCStateEnum*, který popisuje aktuální stav konečného automatu, je z tabulky získána přechodová funkce. V případě, že tato hodnota není zadána, je vyhozena *System.ApplicationException* oznamující uživateli že v tomto místě není takovýto znak očekáván a je poznačeno místo v řetězci kde se právě nachází. V opačném případě je provedena metoda, na niž ukazuje delegát a aktualizován stav konečného automatu. Po skončení smyčky je provedena funkce, jenž se v tabulce nachází na souřadnici označenou prázdným znakem a aktuálním stavem.

#### LoopExecutor

Instance třídy LoopExecutor (Obr. 46) je zodpovědná za provádění vyhodnocovací smyčky. Při zavolání metody *Start* je změněna hodnota vlastnosti *IsRunning* na *true* a vytvořena nová instance časovače *System.Threading.Timer*, jenž má nulové zpoždění a periodu nastavenou dle hodnoty vlastnosti *Period*, jejíž výchozí hodnota je 1000 ms. Delegát volaný po uplynutí periody je asynchronní metoda *Run*. Na začátku je ověřen zámek, zajišťující, aby nenastala situace, kdy je metoda spuštěna vícekrát. Před začátkem vyhodnocení je počkáno na provedení asynchronní metody *loadData* starající se o načtení aktuálních hodnot z uzlů. Vyhodnocení provedeno pomocí *FIowRepository.Run* (viz Kap. 4.1.2.7). Poté jsou výsledky odeslány do příslušných uzlů pomocí metody *writeData*. Po dokončení je vyvolána událost *LoopFinished*, na kterou se mohou napojit třídy z vyšší vrstvy a reagovat tak na dokončení smyčky. Nakonec je do debug terminálu zapsána informace o délce trvání a uvolněn zámek. Metody *loadData* a *writeData* jsou asynchronní, neboť v nich probíhá volání komunikačních metod, jež ze své podstaty musí být asynchronní, ale průběh metody *Run* je pozastaven, dokud nedoběhnou. Obě metody jsou téměř identické, ale pracují s jinou kolekcí. Normální endpointy jsou aktualizovány okamžitě. V případě pomalých je nejprve ověřeno, zda již uzel odpověděl. Pokud ano, proběhne aktualizace hodnot. Jestliže momentálně neprobíhá načítání nových hodnot, je zahájeno.



Obr. Diagram tříd LoopExecutor

### Uživatelské rozhraní

Pro nejvyšší vrstvu, se kterou interaguje uživatel byla zvolena desktopová aplikace. Při zvažování, jaký typ aplikace zvolit bylo myšleno na to, aby se dala jednoduše distribuovat a když uživatel změní velikost okna, tak se mu přizpůsobí. Z možností, jež jsou dostupné bez použití knihoven třetích stran jsou na výběr WinForm (Windows Forms), WPF (Windows Presentation Foundation) a UWP (Universal Windows Platform). WinForm není responzivní a prvky jsou umístěny prostřednictvím absolutních souřadnic, tudíž při změně velikosti okna buď nejsou viditelné, nebo je část okna prázdné. UWP je aplikace určená pro Microsoft Store, kvůli čemuž je pro distribuci nepraktické. Zbývá tedy WPF, jenž je responzivní a distribuuje se jako klasický *exe* soubor. Při realizaci bylo využíváno MVVM (viz Kap. 3.6.5), kdy vzhled je definován pomocí XAML a data s logikou jsou přidány pomocí provázání s ViewModely. Pro lepší čitelnost ViewModelů byl využit NuGet balíček *CommunityToolkit.Mvvm [84]*, který přidává atributy, díky nimž je logika aktualizace View při změně hodnoty vygenerována vývojovým prostředím do samostatných souborů mimo kód psaný programátorem.

#### Komponenty

Pro lepší čitelnost XAML definujících vzhled oken a možnost znovu použitelnosti na více oknech byly grafické prvky, které jsou tvořeny z více elementů nebo obsahují nějakou logiku, vytvořeny jako samostatné komponenty s vlastními ViewModely.

*EndpointView* (na prostředku Obr. 50) sloužící k zobrazení informací o endpointu je tvořen mřížkou s třemi řádky a dvěma sloupci, jež si rozdělí šířku na polovinu. První řádek o výšce 50 px obsahuje *Label*, jehož obsah je navázán na vlastnost *Address* nacházející se v *EndPointViewModel* (Obr. 47) a zabírá dva sloupce. Na druhém řádku se v obou sloupcích nachází kódem zadané popisy sloupců pro hodnoty a argumentu. Poslední řádek zabere veškeré dostupné místo. Zde se nachází *ListView* jejichž zdrojem dat jsou kolekce *Values* a *Arguments*. Ty jsou získány metodou *ValuesDo.ToStringListShort* zavolanou nad příslušnou kolekcí instance třídy *EndPointDo*. Jednotlivé hodnoty jsou zobrazeny pomocí *TextBlock*, která má menší rozměry než *Label*, zabaleném do *Border* se zaoblenými rohy.



Obr. Diagram tříd EndPointViewModel

*FlowListView* (vlevo dole na Obr. 52) je určen k zobrazení seznamu datových toků. Ty jsou reprezentovány třídou *FlowViewModel*, jenž má jméno a datový typ výstupní hodnoty. Tyto objekty se nachází v kolekci *Flows* uvnitř singleton instance *FlowListViewModel* (viz Obr. 48). Ta je vytvářena ze seznamu ve *FlowRepository* (viz Kap. 4.1.2.7) buď při prvním zavolání nebo jako reakce na použití metody *RefreshFlows*. K zobrazení je využíván *ListView* naplněný *Buttony*, jejichž barva textu a ohraničení je nastavena na základě datového typu a *TypeColorConverter*.



Obr. Diagram tříd FlowViewModel

*NodeListView* (vlevo nahoře na Obr. 52 a vlevo uprostřed na Obr. 54) složí k zobrazení připojených uzlů a jejich stavu. Má stejné rozložení jako *FlowViewModel*, ale tlačítko je celé podbarvené podle *NodeStatusConverter* a hodnoty *ConnectionStatus* nacházející se v *NodeViewModel* (viz Kap. 4.1.3.2 a Obr. 49). Při kliknutí je proveden *ShowInfoCommand*, jenž otevře nové okno *NodeInfoWindow*.



Obr. Diagram tříd NodeViewModel

#### Okna

NodeInfoWindow (Obr. 50) slouží k zobrazení informací o uzlu. Jeho ViewModel je instance třídy *NodeViewModel*. V horní části okna se nachází jméno uzlu s výškou 60 px. Pod ním je vložen ListView obsahující komponenty EndpointView s výškou 5\*, jehož zdrojem je kolekce *EndPoints*. Mezi elementy a okraji okna se nachází mezery vytvořené pomocí řádků tabulky. Jejich výška je 0,5\* což odpovídá přibližně 7,7 % výšky okna pro každou mezeru a 77 % pro list.



Obr. Okno s informacemi o uzlu

*AddNodeWindow* (Obr. 51) je okno pro přidání nového uzlu jehož logika se nachází v *NodeViewModel*. Pro snazší umístění prvků byla použita mřížka. Prvky jsou zarovnány na střed a dohromady na výšku zabírají 170 px a na šířku dvě čtvrtiny velikosti okna. Pro zadání názvu a adresy uzlu slouží dvě textová pole s popisem. Po zadání uživatel klikne na tlačítko, čímž je zavolán *ButtonClickCommand*. Tím je zavolána metoda *NodeRepository.AddNode* (viz Kap. 4.1.2.6), jež ověří validitu hodnot a přidá uzel do seznamu. Pokud je vše v pořádku, zobrazí okno *NodeInfoWindow* a aktualizuje kolekci v *NodeListViewModel*.



Obr. Okno pro vložení nového uzlu

Pro zadávání logikyse využívá okno *FlowEditWindow* (Obr. 52) jenž využívá ViewModel *FlowEditWindowViewModel* (Obr. 53). V levé části okna se nachází sloupec se šířkou 80 px obsahující komponenty *NodeListView* a *FlowListView*, aby uživatel viděl názvy, které může použít při zadávání logiky. Zbytek okna zabírá *TabControl*, jenž má momentálně pouze jednu záložku, kterou text. V ní se nachází *ToolBar* a *TextBox* vyplňující celou plochu. Do textového pole je možné psát více řádků, používat tabulátor a pokud se text nevejde na obrazovku zobrazí se scrollbar. Na liště se nachází tlačítko „compile“, která spouští *CompileFlowCommand*. Ta smaže všechny datové toky a zavolá metodu *LoopCompiler.CompileMultiLine*. v závislosti na výsledku zobrazí buď *MessageBox* se zprávou „compiled successfully”, nebo chybovou hlášku. Poté vyvolá aktualizaci grafické reprezentaci seznamu datových toků.



Obr. Okno pro zadávání logiky



Obr. Diagram tříd FlowEditViewModel

*MainWindow* (Obr. 54 a Obr. 55) je úvodní obrazovkou aplikace. Momentálně je využíváno pouze menu na levém okraji a zbytek je nevyužit. Zde se nachází dva *StackPanely* a *NodeListView*. První obsahuje tlačítka „start“ a „stop“ ovládající vyhodnocovací smyčku. Jejich viditelnost se mění podle hodnoty vlastnosti *IsLoopRunning* a za běhu aplikace je vidět právě jedno. Pod nimi se nachází tlačítko „edit“ otevírající okno *FlowEditWindow*. Druhý panel obstarává uzly a obsahuje tlačítka pro načtení a uložení uzlů z disku a pro přidání nového, které otevírá okno *AddNodeWindow*. Stav připojených uzlů je aktualizován pomocí metody, jenž je napojena na událost vyvolanou ukončením vyhodnocovací smyčky.



Obr. Úvodní obrazovka



Obr. Diagram tříd MainWindowViewModel

## Uzly

Jednotlivé uzly jsou realizovány pomocí vývojových desek NodeMCU pro ESP 8266 (viz kap. 3.5.1) a Arduino IDE. Z důvodu omezeného výkonu zde není striktní hranice mezi komunikační a logickou vrstvou. Kde je to možné, byla použita abstrakce, aby byl kód přenositelný na mikrokontrolery nepodporované Arduino IDE nebo využívající jiný způsob komunikace. Kvůli využití třídy *Optional*, umožňující existenci proměnných bez hodnoty, je nutné použít kompilátor podporující C++17 nebo novější.

Projekt je strukturován tak, že většina kódu se nachází ve složce *scr*, která je rozdělena na *Abstract*, *HW*, *Lib* a složky pro jednotlivé uzly. *Abstract* obsahuje abstrakce knihoven specifických pro Arduino, které využívá zbytek projektu. Složka *HW* obsahuje abstrakci pro práci s jednotlivými senzory, zobrazovacími prvky a akčními členy. V *Lib* se nachází sdílená logika a DTO. V kořenové složce se nachází soubor *ESP.ino*, jenž je vstupním bodem pro Arduino IDE. Dále se zde nachází *secret.h*, sloužící k zapsání klíčů a jiných citlivých údajů. Při vytvoření do něj byly zapsány zástupné hodnoty, aby se po stažení repositáře dal projekt zkompilovat, a následně byl vyřazen z verzování.

### Společná část

Pro analytické účely jsou během provozu uzlu posílána data na sériovou linku, což v případě neočekávaného chování usnadní hledání chyby. Po zapnutí je do terminálu odeslána zpráva „boot“ sloužící k označení začátku provozu. Toto je z důvodu, kdyby v terminálu zůstala komunikace z minulého připojení. Dále je vypsán aktuální stav paměti, aby bylo možné odhalit případné problémy z důvodu špatného rozmístění dat mezi zásobník a haldu. Poté proběhne inicializace specifická pro každý uzel. Po jejím dokončení je provedeno připojení k Wi-Fi za využití údajů nacházejících se v souboru *secret.h*, během něhož je vykonána funkce *WaitToConnect*. Ta je definována jako *weak*, což umožňuje, aby uzel měl jiné chování, než je výchozí. Pokud není definováno, je každých 500 ms ověřen stav připojení a dokud nedojde k připojení, je do terminálu posíláno „Connecting...“. Je-li připojování úspěšné, je vypsána IP adresa. Nakonec jsou přidány výchozí endpointy, kterými jsou *getInfo* a zavolání kořenové adresy, a do terminálu je napsáno „Server listening“ následováno časovým razítkem. Od této chvíle až do vypnutí, je v nekonečné smyčce očekáván dotaz určený k zpracování.

#### Abstract

Tato složka obsahuje hlavičkové soubory fungující jako rozhraní. Nachází se zde abstrakce pro komunikaci, logování, serializaci a deserializaci. Soubory *cpp* s implementací pro Arduino se nachází v příslušně pojmenované podsložce. V případě rozšíření řešení o jinou technologii bude přidána nová složka. Jelikož je jen jedna možnost, tak v tento okamžik nejsou použity preprocesory určující, jaká implementace hlavičkových souborů má být využita.

Aby bylo možné měnit způsob komunikace s uzlem je definována třída *CommunicationHandler*. Metoda *StartListening* slouží k přidání reakce na specifikovaný endpoint. Parametry jsou cesta a ukazatel na funkci, jenž se má zavolat jako reakce na jeho obdržení. K odeslání odpovědi slouží metody *SendOk* a *SendError* jejímž parametrem je textový řetězec. Při přijetí dotazu s argumenty, je využita metoda *GetBody*, která má jako parametry buffer a jeho velikost. Pro zpracování přijatých dotazů slouží metoda *Loop*, jenž je volána z nekonečné smyčky v *main*, které odpovídá Arduino funkce *loop*. V případě ESP je implementací využívána instance třídy ESP8266WebServer [85]. Jedná se o HTTP server poslouchající na portu 80, který je spuštěn v konstruktoru *CommunicationHandler*.

Jelikož každá platforma má vlastní knihovny pro serializaci a deserializaci, byla i pro tuto logiku vytvořena abstrakce, která je pro vyšší přehlednost rozdělena na dva soubory. Protože pří kompilaci C++ jsou hlavičkové soubory zkopírovány do cpp soborů, jenž jsou kompilovány samostatně, je nutné jednoznačně určit datový typ využívaný v dané situaci [86]. Z tohoto důvodu nebylo možné pro serializaci *ValueDto* využít generické funkce a místo toho jsou deklarovány pro každý ze tří podporovaných datových typů samostatně. V závislosti na množství potřebných dat jsou pro hodnoty a endpointy definovány funkce *Serialize*, *SerializeInfo* a *SerializeValue*. První vypisuje veškeré údaje a je využívána pro logování. Druhá vytváří odpověď pro endpoint *getInfo*. Poslední se používá při dotazu na hodnoty uzlu. Všechny tři právě popsané funkce mají jako parametry ukazatel na instanci *EndPointDto*, buffer a jeho velikost pro serializaci ednpointu a ukazatel na ValueDto pro hodnoty. Pro serializaci celé kolekce je využívána *SerializeEndpoints*. Při přijetí dotazu, jenž obsahuje argumenty, je použita funkce *Deserialize* jejíž parametry jsou *const char\** obsahující JSON a ukazatel na *EndPointDto*, kam se mají hodnoty zapsat. Arduino implementace využívá knihovnu *ArduinoJson* ve verzi 7.1.0 [87, 88].

Aby bylo možné provádět analýzu v případě chyby, je nutné za běhu programu někam zaznamenávat stavové informace. K tomu slouží funkce *Log* deklarované v hlavičkovém souboru *Logger.h*, které mají jako parametr buď *const char\** nebo buffer a počet znaků. Pro využití v souborech specifický pro Arduino jsou tyto dvě funkce deklarovány v *LoggerExtend.h* také jako String. Implementací je výpis do terminálu pomocí sériové linky.

#### Lib

V této složce se nachází soubory využívající pouze C++ a hlavičkové soubory z ostatních složek, jenž vytváří abstrakci nad platformě závislých knihovnách. Jedná se především o DTO se strukturou odpovídající těm v hlavním uzlu (viz Kap. 4.1.1.1) a logiku potřebnou ke komunikaci s ním.

Generická třída *ValueDto* má konstruktor s *const char\** *name*, jenž slouží k identifikaci hodnoty v hlavním uzlu či jiné aplikaci načítající hodnoty z tohoto uzlu, a generickou *val*, jenž obsahuje uloženou hodnotu. Metoda *GetType* vracející *ValTypeEnum* je využívána při volání endpointu *getInfo*. Ke své činnosti využívá podmínky obsahující *constexpr is\_same\_v*, což je výraz porovnávající dva datové typy, jenž je vyhodnocen již během kompilace. Tato funkce vyžaduje kompilátor podporující C++14 nebo novější. Pokud se jedná o jiný datový typ, než je povolený, vyhodí výjimku *runtime\_error*.

Třída *EndPointDto* obsahuje informace o endpointu. Pro zaznamenání adresy je využívána proměnná *const char\* URL*. Aby bylo možné při zavolání *getInfo* sdělit jaká HTTP metoda má být použita, je třeba zadat hodnotu *HttpEnum*. Jelikož C++ nevyžaduje název enum před jeho hodnotou, bylo před využitím již existujícího enum upřednostněno vytvoření vlastního, neboť by při změně knihovny nebo migraci na jinou technologii mohlo docházet k nečekanému chování, z důvodu rozdílného pořadí hodnot nebo konfliktu s jinou knihovnou. Pokud je k zpracování dotazu více času, je zadána hodnota *Delay*, která zpracovávající aplikaci říká, kdy se ještě nejedná o chybu spojení. Tento údaj je deklarován jako *optional<int>*, čímž je umožněno, aby jinak hodnotový datový typ neměl zadanou hodnotu. Aby hlavní uzel či jiná aplikace pracující s tímto uzlem věděla, zda se jedná o vstupní či výstupní endpoint, je definován enum *EndPointType*. Pro uložení hodnot a argumentů složí *vectory*. Na rozdíl od hlavního uzlu je zde k dispozici omezené množství paměti, takže zde nejsou kolekce hodnot podle účelu sloučeny do dvou objektů, ale nachází se přímo v *EndPointDto* rozlišeny prefixem *Val\_* nebo *Arg\_*. Kromě bezparametrického konstruktoru má třída dva parametrické, lišící se parametrem *delay*. Povinnými hodnotami jsou HTTP metoda a URL adresa. Nepovinným údajem je EndPointType, jenž v případě nevyplnění má hodnotu *EP\_TYPE\_GET*, znamenající, že endpoint slouží k získání hodnot.

Hlavičkový soubor *Node.h* obsahuje deklarace funkcí *NodeInit* a *printEndpoint*. *NodeInit* slouží k definici endpointů specifických pro daný uzel a nachází se v souborech pro konkrétní uzel umístěných ve složkách o úroveň výše. Který z nich bude kompilován se vybírá na základě definice nacházející se na začátku *Node.h* a preprocesorů. Funkce *printEndpoint* je využívána jako kontrola při vytvoření nového endpointu a je definována v souboru *NodeShared.cpp* tak, že provede serializaci veškerých údajů a zapíše je do logu.

Pár hlavičkové souboru a zdrojového kódu *SharedHttpEndpoints* obsahují přidání základních endpointů, kterými jsou *getInfo* a volání bez cesty. Funkce, jenž se provede při zavolání *getInfo*, zavolá *SerializeEndpoints* výsledek zapíše do logu a odešle jako odpověď. V případě, že se za adresou uzlu nenachází žádná cesta, je do logu zapsána autorizační hlavička a tělo dotazu. Poté je odeslána odpověď „hello world“. Tento endpoint je využíván při testování komunikace, kdy není žádoucí, aby se uzel pokoušel o zpracování právě přijatých hodnot.

### Uzel 1

První realizovaný uzel je vybaven sensorem DHT11 a monochromatickým OLED displejem. DHT11 od společnosti Adafruit je levný teploměr a vlhkoměr komunikující pomocí protokolu 1wire. Ačkoliv se s přesností teploty ± 2 °C jedná spíše o orientační hodnotu, byl tento sensor ponechán pro své dynamické vlastnosti. Jelikož je po odečtení hodnoty potřeba počkat dalších 2000 ms, než je možno získat další, slouží tento sensor jako ukázka, jak si hlavní uzel poradí s pomalu odpovídajícím uzlem. Pro komunikaci jsou využívány knihovny *DHT sensor library* a *Adafruit Unified Sensor[89, 90]*. Bílý monochromatický OLED displej s rozlišením 128x64 px je řízen čipem SSD1306, který umožňuje komunikaci pomocí I2C. Pro ovládání jsou využívány knihovny *Adafruit GFX Library* a *Adafruit\_SSD1306* [91, 92]. [93–95]

Pro abstrakci uvnitř *Node1.cpp* byla vytvořena třída *DhtWrapper*. Kromě snazší přenositelnosti je důvodem k abstrakci fakt, že knihovna po dobu 2000 ms od posledního čtení vrací stejnou hodnotu, ale časové razítko není zvenčí dostupné. Pro získání hodnot slouží metoda *ReadRaw*, jenž získá teplotu a vlhkost ihned po sobě a uloží je do proměnných. Poté porovná své časové razítko a pokud uplynul daný limit, aktualizuje ho. Pro přístup k takto přečteným hodnotám slouží metody *GetTemp* a *GetHumid* a *GetDataAge*. Pokud je potřeba aby data byla aktuální, je zavolána metoda *WaitForNewestData*, která počká do uplynutí zbývajícího času a poté teprve proběhne četní.

Tento uzel má přepsanou společnou funkci *WaitToConnect*, aby během čekání na připojení nejen vypisovala tečky do terminálu, ale také na displeji blikal symbol Wi-Fi. Po připojení se vedle něj vypíše IP adresa.

Jsou definovány endpointy *getDhtValuesNew* a *getDhtValuesAny*, které vrací teplotu, vlhkost a stáří dat. Liší se tím, zda jsou poslány hodnoty bez ohledu na stáří, nebo je čekáno na čerstvé. Poté co je odeslána odpověď, jsou tyto údaje vypsány na displej včetně časové značky počítané od doby zapnutí napájení.

### Uzel 2

text

### Uzel 3

text

# Výsledky a diskuse

## Podkapitola úroveň 2

Text…

### Podkapitola úroveň 3

Text…

### Podkapitola úroveň 3

Text…

## Podkapitola úroveň 2

Text…

# Závěr

Text…

# Seznam použitých zdrojů

[1] What is OSI Model | 7 Layers Explained. *GeeksForGeeks* [online]. [vid. 2025-01-28]. Dostupné z: https://www.geeksforgeeks.org/open-systems-interconnection-model-osi/

[2] Difference Between OSI Model and TCP/IP Model - GeeksforGeeks. *GeeksForGeeks* [online]. [vid. 2025-01-28]. Dostupné z: https://www.geeksforgeeks.org/difference-between-osi-model-and-tcp-ip-model/

[3] MICHAEL GOODWIN a CHRYSTAL R. CHINA. What Is the OSI Model? | IBM. *IBM* [online]. [vid. 2025-01-28]. Dostupné z: https://www.ibm.com/think/topics/osi-model

[4] What is the OSI Model? | Cloudflare. *Cloudflare* [online]. [vid. 2025-01-28]. Dostupné z: https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/

[5] *Bytebytego Big Archive System Design 2023* [online]. 2023 [vid. 2025-01-26]. Dostupné z: https://blog.bytebytego.com/p/free-system-design-pdf-158-pages

[6] MICHAEL GOODWIN, GITA JACKSON a TASMIHA KHAN. What Is Network Topology? | IBM. *IBM* [online]. [vid. 2025-01-28]. Dostupné z: https://www.ibm.com/think/topics/network-topology

[7] What Is a Data Link Layer? | Coursera. *Coursera* [online]. [vid. 2025-02-02]. Dostupné z: https://www.coursera.org/articles/data-link-layer

[8] What is Cyclic Redundancy Check (CRC) and How Does it Work? | Lenovo US. *Lenovo* [online]. [vid. 2025-02-02]. Dostupné z: https://www.lenovo.com/us/en/glossary/cyclic-redundancy-check/?orgRef=https%253A%252F%252Fwww.perplexity.ai%252F

[9] Difference Between Packet And Frame - PyNet Labs. *PyNet Labs* [online]. [vid. 2025-02-08]. Dostupné z: https://www.pynetlabs.com/what-is-the-difference-between-packet-and-frame/

[10] What is Ports in Networking? - GeeksforGeeks. *GeeksforGeeks* [online]. [vid. 2025-02-05]. Dostupné z: https://www.geeksforgeeks.org/what-is-ports-in-networking/?ref=header\_outind

[11] What is Protocol? A Guide to Understanding | Lenovo US. *Lenovo* [online]. [vid. 2025-02-10]. Dostupné z: https://www.lenovo.com/us/en/glossary/what-is-protocol/?orgRef=https%253A%252F%252Fwww.perplexity.ai%252F

[12] IPv4 vs. IPv6: what are the differences in 2025? - Surfshark. *SurfShark* [online]. [vid. 2025-02-11]. Dostupné z: https://surfshark.com/blog/ipv4-vs-ipv6

[13] What is the Internet Protocol? | Cloudflare. *Cloudflare* [online]. [vid. 2025-02-10]. Dostupné z: https://www.cloudflare.com/learning/network-layer/internet-protocol/

[14] KRISTOFER KOISHIGAWA. Subnet Cheat Sheet – 24 Subnet Mask, 30, 26, 27, 29, and other IP Address CIDR Network References. *FreeCodeCamp* [online]. [vid. 2025-02-12]. Dostupné z: https://www.freecodecamp.org/news/subnet-cheat-sheet-24-subnet-mask-30-26-27-29-and-other-ip-address-cidr-network-references/

[15] ADITYAPRATAPBHUYAN. Understanding Network Address Translation (NAT) in Networking: A Comprehensive Guide - DEV Community. *Dev.to* [online]. [vid. 2025-02-13]. Dostupné z: https://dev.to/adityapratapbh1/understanding-network-address-translation-nat-in-networking-a-comprehensive-guide-8bo

[16] Dynamic Host Configuration Protocol (DHCP) | Microsoft Learn. *Microsoft Learn* [online]. [vid. 2025-02-13]. Dostupné z: https://learn.microsoft.com/en-us/windows-server/networking/technologies/dhcp/dhcp-top

[17] What is the User Datagram Protocol (UDP)? | Cloudflare. *Cloudflare* [online]. [vid. 2025-02-14]. Dostupné z: https://www.cloudflare.com/learning/ddos/glossary/user-datagram-protocol-udp/

[18] *Transmission Control Protocol (TCP) (článek) | Khan Academy* [online]. [vid. 2025-02-14]. Dostupné z: https://cs.khanacademy.org/computing/informatika-pocitace-a-internet/x8887af37e7f1189a:internet/x8887af37e7f1189a:tcp-protokol/a/transmission-control-protocol--tcp

[19] *What is TCP/IP? | Cloudflare* [online]. [vid. 2025-02-14]. Dostupné z: https://www.cloudflare.com/learning/ddos/glossary/tcp-ip/

[20] An overview of HTTP - HTTP | MDN. *Mozilla Developer Network* [online]. [vid. 2025-02-15]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[21] Caching - IBM Documentation. *IBM* [online]. [vid. 2025-02-19]. Dostupné z: https://www.ibm.com/docs/en/was-nd/8.5.5?topic=discussions-caching

[22] HTTP Load Balancing | NGINX Documentation. *NGINX* [online]. [vid. 2025-02-19]. Dostupné z: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

[23] HTTP response status codes - HTTP | MDN. *Mozilla Developer Network* [online]. [vid. 2025-02-15]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[24] RUFAI MUSTAPHA. What is HTTP? Protocol Overview for Beginners. *freeCodeCamp* [online]. [vid. 2025-02-14]. Dostupné z: https://www.freecodecamp.org/news/what-is-http/

[25] How does public key cryptography work? | Public key encryption and SSL | Cloudflare. *Cloudflare* [online]. [vid. 2025-02-17]. Dostupné z: https://www.cloudflare.com/learning/ssl/how-does-public-key-encryption-work/

[26] ARTHUR BELLORE. The TLS Handshake Explained. *auth0* [online]. [vid. 2025-02-15]. Dostupné z: https://auth0.com/blog/the-tls-handshake-explained/

[27] BYTEBYTEGO. *SSL, TLS, HTTPS Explained - YouTube* [online]. [vid. 2025-02-15]. Dostupné z: https://www.youtube.com/watch?v=j9QmMEWmcfo

[28] COMPUTERPHILE a DR. MIKE POUND. *TLS Handshake Explained - Computerphile - YouTube* [online]. [vid. 2025-02-15]. Dostupné z: https://www.youtube.com/watch?v=86cQJ0MMses&t=4s

[29] How does public key cryptography work? | Public key encryption and SSL | Cloudflare. *Cloudflare* [online]. [vid. 2025-02-15]. Dostupné z: https://www.cloudflare.com/learning/ssl/how-does-public-key-encryption-work/

[30] What is SSL/TLS Certificate? - SSL/TLS Certificates Explained - AWS. *Amazon Web Services* [online]. [vid. 2025-02-15]. Dostupné z: https://aws.amazon.com/what-is/ssl-certificate/

[31] HTTP vs HTTPS - Difference Between Transfer Protocols - AWS. *Amazon Web Services* [online]. [vid. 2025-02-15]. Dostupné z: https://aws.amazon.com/compare/the-difference-between-https-and-http/

[32] BAELDUNG. What Are Serialization and Deserialization in Programming? | Baeldung on Computer Science. *Baeldung* [online]. [vid. 2025-02-22]. Dostupné z: https://www.baeldung.com/cs/serialization-deserialization

[33] SATRAPA, Pavel. Jazyky pro popis dat. In: *Jazyky pro popis dat* [online]. B.m.: TUL, nedatováno [vid. 2025-02-21]. Dostupné z: https://www.nti.tul.cz/~satrapa/vyuka/xml/prednaska01.pdf

[34] SATRAPA, Pavel. API pro XML. In: *Jazyky pro popis dat* [online]. B.m.: TUL, nedatováno [vid. 2025-02-20]. Dostupné z: https://www.nti.tul.cz/~satrapa/vyuka/xml/prednaska12.pdf

[35] ŠIMON RAICHL. Lekce 7 - Formát JSON. *ITnetwork.cz* [online]. [vid. 2025-02-21]. Dostupné z: https://www.itnetwork.cz/javascript/oop/objekty-json-a-vylepseni-diare-v-javascriptu

[36] JSON. *JSON.org* [online]. [vid. 2025-02-21]. Dostupné z: https://www.json.org/json-en.html

[37] PETR SEDLÁČEK. Lekce 2 - REST API, SOAP, GraphQL a JSON. *ITnetwork.cz* [online]. [vid. 2025-02-21]. Dostupné z: https://www.itnetwork.cz/javascript/nodejs/rest-api-soap-graph-a-json

[38] Exploring Why CSV is a Popular File Format and How to Manage it | Lenovo UK. *Lenovo* [online]. [vid. 2025-02-21]. Dostupné z: https://www.lenovo.com/gb/en/glossary/csv/?orgRef=https%253A%252F%252Fwww.perplexity.ai%252F

[39] Standardy Wi-Fi: IEEE 802.11ac, 802.11ax a standardy bezdrátového připojení | Dell Česká republika. *Dell* [online]. [vid. 2025-02-22]. Dostupné z: https://www.dell.com/support/contents/cs-cz/article/product-support/self-support-knowledgebase/networking-wifi-and-bluetooth/wi-fi-network-standards-overview

[40] IEEE SA - The Evolution of Wi-Fi Technology and Standards. *IEEE* [online]. [vid. 2025-02-23]. Dostupné z: https://standards.ieee.org/beyond-standards/the-evolution-of-wi-fi-technology-and-standards/

[41] Different Wi-Fi Protocols and Data Rates. *Intel* [online]. [vid. 2025-02-23]. Dostupné z: https://www.intel.com/content/www/us/en/support/articles/000005725/wireless/legacy-intel-wireless-products.html#primary-content

[42] What is WiFi 6E? | TP-Link. *TP-Link* [online]. [vid. 2025-02-24]. Dostupné z: https://www.tp-link.com/us/wifi-6e/

[43] Brief introduction of Wireless Channel, Channel Width and DFS | TP-Link Norway. *TP-Link* [online]. [vid. 2025-02-28]. Dostupné z: https://www.tp-link.com/no/support/faq/4309/

[44] KLEMENT, Doc Phdr Milan. Univerzita Palackého v Olomouci Technologie bezdrátových sítí základní principy a standardy [online]. 2019 [vid. 2025-02-26]. Dostupné z: https://www.pdf.upol.cz/fileadmin/userdata/PdF/katedry/ktiv/Studijni\_materialy/Klement/2019/TBS\_2019\_skripta.pdf

[45] What is a Wireless Access Point (WAP)? Benefits & How It Works | Lenovo US. *Lenovo* [online]. [vid. 2025-02-28]. Dostupné z: https://www.lenovo.com/us/en/glossary/wireless-access-point/?orgRef=https%253A%252F%252Fwww.perplexity.ai%252F

[46] STA Access - NetEngine AR600, AR6100, AR6200, and AR6300 V300R019 CLI-based Configuration Guide - WLAN-FAT AP - Huawei. *Huawei* [online]. [vid. 2025-02-28]. Dostupné z: https://support.huawei.com/enterprise/en/doc/EDOC1100112363/75acc8a8/sta-access

[47] 802.11 Standards - NetEngine AR600, AR6100, AR6200, and AR6300 V300R019 CLI-based Configuration Guide - WLAN-FAT AP - Huawei. *Huawei* [online]. [vid. 2025-02-09]. Dostupné z: https://support.huawei.com/enterprise/en/doc/EDOC1100112363/b1db415/80211-standards

[48] Wi-Fi CERTIFIED 6TM coming in 2019 | Wi-Fi Alliance. *Wi-Fi Alliance* [online]. [vid. 2025-02-24]. Dostupné z: https://www.wi-fi.org/news-events/newsroom/wi-fi-certified-6-coming-in-2019

[49] LINUS TECH TIPS. *Just how FAST is WiFi 6? - YouTube* [online]. [vid. 2025-02-24]. Dostupné z: https://www.youtube.com/watch?v=Mx5-T8ZwxbU

[50] What Is Wi-Fi 6? - Intel. *Intel* [online]. [vid. 2025-02-24]. Dostupné z: https://www.intel.com/content/www/us/en/gaming/resources/wifi-6.html

[51] ANTHONY M. BRUNO. What is Quadrature Amplitude Modulation (QAM)? *CWNP* [online]. [vid. 2025-03-02]. Dostupné z: https://www.cwnp.com/qam-basics/

[52] *What Is QAM? How Does QAM Work? - Huawei* [online]. [vid. 2025-03-02]. Dostupné z: https://info.support.huawei.com/info-finder/encyclopedia/en/QAM.html

[53] WPA2 Security (KRACKs) Vulnerability Statement | TP-Link Baltic. *TP-Link* [online]. [vid. 2025-03-02]. Dostupné z: https://www.tp-link.com/baltic/support/faq/1970/

[54] IRMA ŠLEKYTĖ. WEP, WPA, WPA2, and WPA3: Main differences | NordVPN. *NordVPN* [online]. [vid. 2025-03-02]. Dostupné z: https://nordvpn.com/blog/wep-vs-wpa-vs-wpa2-vs-wpa3/

[55] An Introduction to Spread-Spectrum Communications | Analog Devices. *Analog Devices* [online]. [vid. 2025-02-25]. Dostupné z: https://www.analog.com/en/resources/technical-articles/introduction-to-spreadspectrum-communications--maxim-integrated.html

[56] LESLIE A. RUSCH. GEL7114 - Module 4.12 - OFDM introduction. In: *GEL-7114 Digital Communications* [online]. B.m.: Universite Laval, 2020 [vid. 2025-03-01]. Dostupné z: https://www.youtube.com/watch?v=i3LBGw8Yle4

[57] BHARDWAJ, Manushree, Arun GANGWAR a Devendra SONI. A Review on OFDM: Concept, Scope & its Applications. *IOSR Journal of Mechanical and Civil Engineering (IOSRJMCE)* [online]. nedatováno, **1**(1), 7–11 [vid. 2025-03-03]. Dostupné z: www.iosrjournals.orgwww.iosrjournals.org

[58] RF ELEMENTS S.R.O. *Inside Wireless: MIMO Introduction - Multiple Input Multiple Output - YouTube* [online]. [vid. 2025-02-26]. Dostupné z: https://www.youtube.com/watch?v=T7NyrG4\_RSI

[59] What Is MIMO? From SISO to MIMO - Huawei. *Huawei* [online]. [vid. 2025-03-02]. Dostupné z: https://info.support.huawei.com/info-finder/encyclopedia/en/MIMO.html

[60] Detailed explanation of MU-MIMO technology and the application of MU-MIMO in WiFi6. *FS* [online]. [vid. 2025-03-02]. Dostupné z: https://www.fs.com/blog/demystifying-mumimo-technology-in-wifi-6-115.html

[61] GREAVES, David J. *Modern System-on-Chip Design on Arm* [online]. B.m.: ARM, nedatováno [vid. 2025-03-03]. ISBN 978-1-911531-37-1. Dostupné z: https://armkeil.blob.core.windows.net/developer/Files/pdf/ebook/arm-modern-soc-design-on-arm.pdf

[62] VLSI | Analog Devices. *Analog Devices* [online]. [vid. 2025-03-05]. Dostupné z: https://www.analog.com/en/resources/glossary/vlsi.html

[63] JOSH SCHNEIDER a IAN SMALLEY. What is a microprocessor? | IBM. *IBM* [online]. [vid. 2025-03-05]. Dostupné z: https://www.ibm.com/think/topics/microprocessor

[64] JOSH SCHNEIDER a IAN SMALLEY. What is a microcontroller? | IBM. *IBM* [online]. [vid. 2025-03-05]. Dostupné z: https://www.ibm.com/think/topics/microcontroller

[65] ESPRESSIF SYSTEMS. *ESP8266EX Datasheet* [online]. 2023 [vid. 2025-03-06]. Dostupné z: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex\_datasheet\_en.pdf

[66] *Getting Started with ESP8266 NodeMCU Development Board| Random Nerd Tutorials* [online]. [vid. 2025-03-07]. Dostupné z: https://randomnerdtutorials.com/getting-started-with-esp8266-wifi-transceiver-review/

[67] ESPRESSIF SYSTEMS. *ESP8266 Hardware Design Guidelines Version 2.8* [online]. 2024 [vid. 2025-03-07]. Dostupné z: https://www.espressif.com/sites/default/files/documentation/esp8266\_hardware\_design\_guidelines\_en.pdf

[68] What’s a design pattern? *Refactoring Guru* [online]. [vid. 2025-01-25]. Dostupné z: https://refactoring.guru/design-patterns/what-is-pattern

[69] Why should I learn patterns? *Refactoring Guru* [online]. [vid. 2025-01-25]. Dostupné z: https://refactoring.guru/design-patterns/why-learn-patterns

[70] Difference Between Architectural Style, Architectural Patterns and Design Patterns - GeeksforGeeks. *GeeksForGeeks* [online]. [vid. 2025-01-26]. Dostupné z: https://www.geeksforgeeks.org/difference-between-architectural-style-architectural-patterns-and-design-patterns/

[71] STEVE “ARDALIS” SMITH. *Architecting-Modern-Web-Applications-with-ASP.NET-Core-and-Azure* [online]. 2023 [vid. 2025-01-21]. Dostupné z: https://dotnet.microsoft.com/en-us/download/e-book/aspnet/pdf

[72] RITVIK GUPTA. Software Architecture Patterns: What Are the Types and Which Is the Best One for Your Project | Turing. *Turing* [online]. [vid. 2025-01-26]. Dostupné z: https://www.turing.com/blog/software-architecture-patterns-types

[73] Dependency Injection(DI) Design Pattern - GeeksforGeeks. *GeeksForGeeks* [online]. [vid. 2025-01-19]. Dostupné z: https://www.geeksforgeeks.org/dependency-injectiondi-design-pattern/

[74] Single Responsibility in SOLID Design Principle - GeeksforGeeks. *GeeksForGeeks* [online]. [vid. 2025-01-19]. Dostupné z: https://www.geeksforgeeks.org/single-responsibility-in-solid-design-principle/

[75] NuGet Gallery | Microsoft.Extensions.DependencyInjection 1.0.0. *NuGet* [online]. [vid. 2025-01-23]. Dostupné z: https://www.nuget.org/packages/Microsoft.Extensions.DependencyInjection/1.0.0#supportedframeworks-body-tab

[76] Dependency injection - .NET | Microsoft Learn. *Microsoft Learn* [online]. [vid. 2025-01-23]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

[77] Create Data Transfer Objects (DTOs) | Microsoft Learn. *Microsoft Learn* [online]. [vid. 2025-01-24]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5

[78] BAELDUNG. The DTO Pattern (Data Transfer Object) | Baeldung. *Baeldung* [online]. [vid. 2025-01-24]. Dostupné z: https://www.baeldung.com/java-dto-pattern

[79] Difference Between MVC, MVP and MVVM Architecture Pattern in Android - GeeksforGeeks. *GeeksForGeeks* [online]. [vid. 2024-11-26]. Dostupné z: https://www.geeksforgeeks.org/difference-between-mvc-mvp-and-mvvm-architecture-pattern-in-android/

[80] NIMROD KRAMER. Android Architecture Patterns: MVC vs MVVM vs MVP. *daily.dev* [online]. [vid. 2025-01-03]. Dostupné z: https://daily.dev/blog/android-architecture-patterns-mvc-vs-mvvm-vs-mvp

[81] RICH LANDER. core/release-notes/8.0/supported-os.md at main · dotnet/core · GitHub. *GitHub* [online]. [vid. 2025-03-10]. Dostupné z: https://github.com/dotnet/core/blob/main/release-notes/8.0/supported-os.md

[82] . NET and .NET Core official support policy. *Microsoft* [online]. [vid. 2025-03-10]. Dostupné z: https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core

[83] core/release-notes/6.0/supported-os.md at main · dotnet/core · GitHub. *GitHub* [online]. [vid. 2025-03-10]. Dostupné z: https://github.com/dotnet/core/blob/main/release-notes/6.0/supported-os.md

[84] NuGet Gallery | CommunityToolkit.Mvvm 8.2.2. *NuGet* [online]. [vid. 2025-03-16]. Dostupné z: https://www.nuget.org/packages/CommunityToolkit.Mvvm/8.2.2?\_src=template

[85] *esp8266/Arduino: ESP8266 core for Arduino* [online]. [vid. 2025-03-19]. Dostupné z: https://github.com/esp8266/Arduino

[86] *26.1 — Template classes – Learn C++* [online]. [vid. 2024-09-19]. Dostupné z: https://www.learncpp.com/cpp-tutorial/template-classes/

[87] *ArduinoJson: Efficient JSON serialization for embedded C++* [online]. [vid. 2025-03-20]. Dostupné z: https://arduinojson.org/?utm\_source=meta&utm\_medium=library.properties

[88] *How to upgrade from ArduinoJson 6 to 7 - YouTube* [online]. [vid. 2024-07-21]. Dostupné z: https://www.youtube.com/watch?v=eE6\_77YIkzI

[89] *adafruit/Adafruit\_Sensor: Common sensor library* [online]. [vid. 2025-03-24]. Dostupné z: https://github.com/adafruit/Adafruit\_Sensor

[90] *adafruit/DHT-sensor-library: Arduino library for DHT11, DHT22, etc Temperature & Humidity Sensors* [online]. [vid. 2025-03-24]. Dostupné z: https://github.com/adafruit/DHT-sensor-library

[91] *adafruit/Adafruit-GFX-Library: Adafruit GFX graphics core Arduino library, this is the „core" class that all our other graphics libraries derive from* [online]. [vid. 2025-03-24]. Dostupné z: https://github.com/adafruit/Adafruit-GFX-Library

[92] *adafruit/Adafruit\_SSD1306: Arduino library for SSD1306 monochrome 128x64 and 128x32 OLEDs* [online]. [vid. 2025-03-24]. Dostupné z: https://github.com/adafruit/Adafruit\_SSD1306

[93] *Overview | DHT11, DHT22 and AM2302 Sensors | Adafruit Learning System* [online]. [vid. 2025-03-24]. Dostupné z: https://learn.adafruit.com/dht

[94] GM electronic | Modul teploměru a vlhkoměru s DHT11. *GME* [online]. [vid. 2025-03-24]. Dostupné z: https://www.gme.cz/v/1508421/modul-teplomeru-a-vlhkomeru-s-dht11

[95] OLED displej 0,96 palce. *GME* [online]. nedatováno [vid. 2025-03-24]. Dostupné z: https://img.gme.cz/files/eshop\_data/eshop\_data/9/772-153/dsh.772-153.1.pdf

[96] *ESP8266 Pinout Reference: How To Use ESP8266 GPIO Pins* [online]. [vid. 2025-03-19]. Dostupné z: https://electropeak.com/learn/esp8266-pinout-reference-how-to-use-esp8266-gpio-pins/

# Přílohy

Odkazovaný seznam příloh



Příloha Piny NodeMCU [96]



Příloha Schéma zapojení ESP8266EX [67]



Příloha Diagram tříd LoopCompiler



Příloha Tabulka stavů konečného automatu