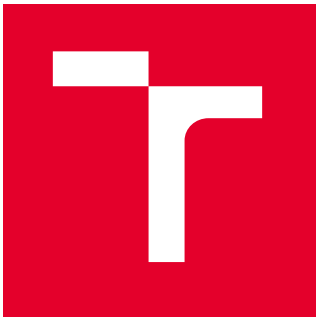


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

ROZHRANÍ IEC-104 PRO PLC KONTROLERY ŘADY UNIPI PATRON

IEC-104 INTERFACE FOR UNIPI PATRON PLCS

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **Slávek Rylich**

AUTHOR

VEDOUCÍ PRÁCE — **doc. Mgr. Karel Slavíček, Ph.D.**

SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Telekomunikační a informační systémy**

Ústav telekomunikací

Student: Slávek Rylich ***ID:*** 223284

Ročník: 3 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Rozhraní IEC-104 pro PLC kontrolery řady UniPi Patron

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je vývoj softwarového modulu pro PLC kontroler řady UniPi Patron, který bude implementovat protokol IEC-104. Úkolem je analýza protokolu IEC-104, návrh struktury softwarového modulu a jeho praktická implementace na PLC kontroleru řadu UniPi Patron. Data přijímaná tímto protokolem budou ukládána ve vhodném databázovém systému (Grafana/Nodered/Mervis), jehož výběr bude proveden v rámci bakalářské práce.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 28.5.2024

Vedoucí práce: doc. Mgr. Karel Slavíček, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

ABSTRAKT

Cílem této bakalářské práce je navrhnout, implementovat a ověřit modul, který umožní efektivní integraci protokolu IEC-104 do existujícího PLC kontroléru řady UniPi Patron. Protokol IEC-104 je standardem v energetickém a průmyslovém sektoru pro spolehlivou a vysokorychlostní komunikaci mezi zařízeními. Implementace tohoto protokolu do kontroléru UniPi Patron rozšiřuje jeho možnosti v oblasti sběru dat a reálně časové výměny informací, což přispívá k lepšímu monitorování a řízení průmyslových procesů. Výsledkem práce je funkční modul, běžící na PLC kontroléru Unipi Patron, který umožňuje spolehlivou komunikaci s řídicími SCADA systémy prostřednictvím protokolu IEC-104, čímž přispívá k rozšíření možností tohoto řídicího systému v průmyslových aplikacích.

KLÍČOVÁ SLOVA

APCI, APDU, ASDU, IEC-104, Patron, PLC, protokol dálkového řízení, průmyslová komunikace, Python, RTU, SCADA, Unipi

ABSTRACT

The aim of this bachelor's thesis is to design, implement and verify a module that will enable the effective integration of the IEC-104 protocol into an existing PLC controller of the UniPi Patron series. The IEC-104 protocol is a standard in the energy and industrial sectors for reliable and high-speed communication between devices. The implementation of this protocol in the UniPi Patron controller expands its capabilities in the area of data collection and real-time information exchange, which contributes to better monitoring and control of industrial processes. The result of the work is a functional module running on the Unipi Patron PLC controller, which enables reliable communication with control SCADA systems through the IEC-104 protocol, thus contributing to the expansion of the capabilities of this control system in industrial applications.

KEYWORDS

APCI, APDU, ASDU, IEC-104, Patron, remote control protokol, industrial communication, PLC, Python, RTU, SCADA, Unipi

Vysázeno pomocí balíčku `thesis` verze 4.09; `https://latex.fekt.vut.cz/`

RYLICH, Slávek. *Rozhraní IEC-104 pro PLC kontrolery řady UniPi Patron*. Semestrální práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2024. Vedoucí práce: doc. Mgr. Karel Slavíček, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Slávek Rylich
VUT ID autora:	223284
Typ práce:	Semestrální práce
Akademický rok:	2023/24
Téma závěrečné práce:	Rozhraní IEC-104 pro PLC kontrolery řady UniPi Patron

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu doc. Mgr. Karlu Slavíčkovi, Ph.D. za odborné vedení a konzultace k práci. A také panu Ing. Martinu Třískovi za konzultace, trpělivost, a hlavně podnětné návrhy k práci.

Obsah

Seznam obrázků

Úvod

V průmyslových prostředích, kde se kladou vysoké nároky na spolehlivost, kompatibilitu a efektivitu komunikace mezi řídicími systémy, se stává nezbytným vývoj specifických rozhraní pro propojení a integraci různých zařízení. Tato práce se zaměřuje na vývoj softwarového modulu implementujícího rozhraní IEC-104 pro PLC kontrolér řady Patron společnosti unipi technology s.r.o., s cílem rozšířit jeho schopnosti komunikace a interoperability v průmyslovém prostředí.

IEC-104 představuje standardizovaný protokol pro komunikaci mezi zařízeními v energetice a průmyslu. Jeho implementace do PLC kontroléru řady unipi Patron přináší významné možnosti v oblasti sběru a výměny dat v reálném čase, což zlepšuje možnosti monitorování a řízení průmyslových procesů.

Cílem této práce je analyzovat požadavky protokolu IEC-104 a navrhnout softwarový modul, který umožní efektivní implementaci tohoto rozhraní do existujícího PLC kontroléru řady unipi Patron. Modul bude vyvinut s ohledem na jeho spolehlivost, bezpečnost a schopnost manipulace s daty v souladu se standardy IEC-104.

Struktura této bakalářské práce bude systematicky pokrývat každý krok procesu vývoje softwarového modulu rozhraní IEC-104 pro PLC kontrolér řady unipi Patron. Obsah práce bude rozdělen do několika hlavních sekcí, které zahrnou:

1. Analýzu protokolu IEC-104: Identifikaci klíčových požadavků a specifikací protokolu pro návrh modulu.
2. Návrh a architekturu modulu: Detailní popis návrhu struktury a rozhraní softwarového modulu pro efektivní integraci IEC-104 do PLC kontroléru řady unipi Patron.
3. Implementaci a testování: Proces implementace navrženého modulu s důrazem na ověření jeho funkcionality a spolehlivosti prostřednictvím testování.
4. Zhodnocení výsledků a závěr: Zhodnocení dosažených výsledků.

1 Seznámení s IEC 60870-5

IEC (International Electrotechnical Commission – Mezinárodní elektrotechnická komise) definuje sadu pravidel 60870 pro systémy a zařízení pro dálkové ovládání se sériovým přenosem bitově kódovaných dat pro sledování a řízení geograficky rozlehlých procesů. Část 5 (v názvu normy 60870-5) pak definuje přenosové protokoly mezi řídícími a řízenými stanicemi RTU (Remote Terminal Unit – Vzdálená koncová jednotka), které mezi sebou udržují stále navázané datové spojení.

Norma IEC 60870-5 se rozděluje na více částí, které popisují základní parametry a struktury pro zajištění komunikace. Jejich základní popisy podle [2]:

- IEC 60870-5-1 (1990) - Formáty přenosového rámce
  - Určuje základní požadavky pro služby zajišťující spojení aplikací dálkového ovládání na fyzické a linkové vrstvě. Specifikuje normy pro kódování, formátování a synchronizaci datových rámců pevné i proměnné délky, které splňují požadavky na integritu dat.
- IEC 60870-5-2 (1992) - Procedury linkové přenosu
  - Platí pro zařízení a systémy dálkového ovládání s kódovaným bitovým sériovým přenosem dat pro monitorování a řízení geograficky rozšířených procesů.
- IEC 60870-5-3 (1992) - Obecná struktura aplikačních dat
  - Specifikuje pravidla pro strukturu aplikačních datových jednotek v přenosových rámcích dálkového ovládání.
- IEC 60870-5-4 (1993) - Definice a kódování aplikačních informačních prvků
  - Uvádí pravidla pro definování informačních prvků a představuje soubor informačních prvků, zejména digitálních a analogových procesních proměnných často používaných v aplikacích dálkového ovládání jako jsou celá čísla, kladná celá čísla, čísla s pohyblivou řádovou čárkou nebo také časové značky.
- IEC 60870-5-5 (1995) - Základní aplikační funkce
  - Definuje společnou normu pro interoperabilitu mezi kompatibilními zařízeními dálkového ovládání.
- IEC 60870-5-6 (2006)
  - Specifikuje metody pro testování shody zařízení dálkového ovládání mezi automatizačními systémy rozvoden a systémy dálkového ovládání, včetně předních funkcí SCADA (Supervisory Control And Data Acquisition - Dispečerské řízení a sběr dat). Usnadňuje interoperabilitu tím, že poskytuje standardní metodu testování implementací protokolů. Očekává se, že minimalizuje riziko neinteroperability.

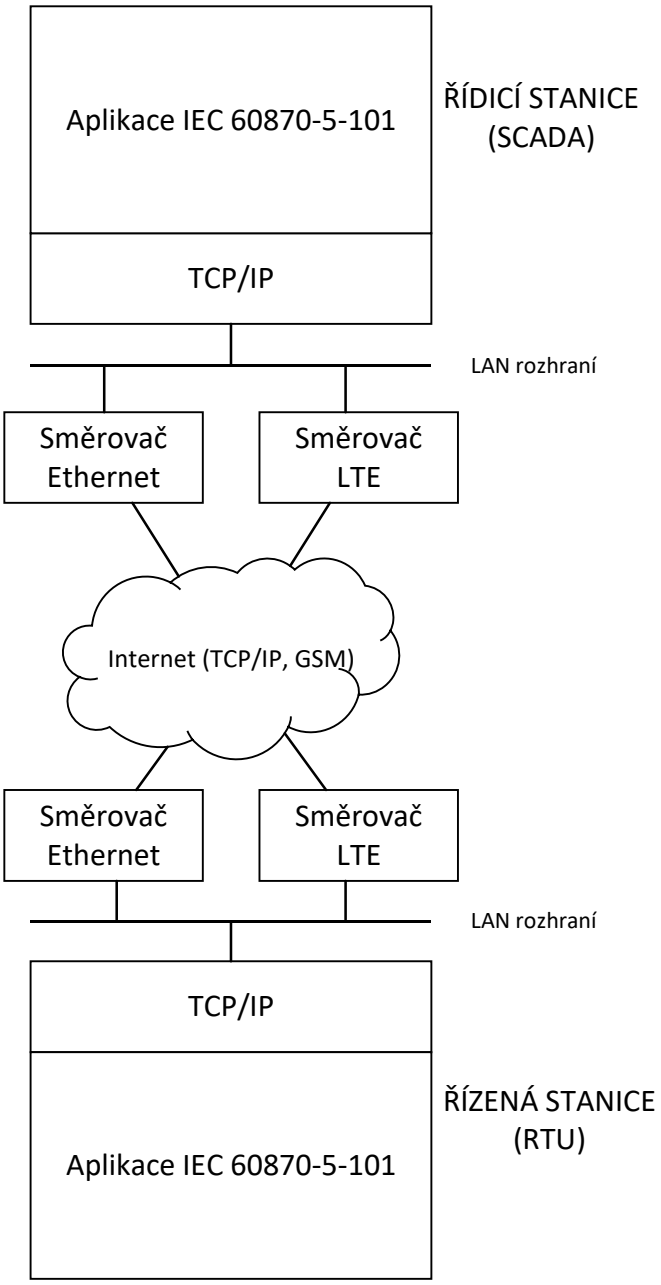
- IEC TS 60870-5-7
  - Bezpečnostní rozšíření pro protokoly IEC 60870-5-101 a IEC 60870-5-104 (použití IEC 62351)
- IEC 60870-5-101 (1995) - Společná norma pro základní úkoly dálkového ovládání
  - Definuje doplňkový standard dálkového ovládání, který umožňuje interoperabilitu mezi kompatibilními zařízeními pro dálkové ovládání. Definovaný doplňkový standard dálkového ovládání využívá standardy řady dokumentů IEC 60870-5.
- IEC 60870-5-102 (1996) - Společná norma pro přenos integrovaných součtových hodnot v elektrizačních soustavách
  - Standardizuje přenos integrovaných součtů představujících množství elektrické energie přenesené mezi energetickými společnostmi nebo mezi energetickými společnostmi a nezávislými výrobci v síti nízkého nebo vysokého napětí.
- IEC 60870-5-103 (1997) - Společná norma pro informační rozhraní ochran
  - Platí pro ochranná zařízení s kódovaným bitovým sériovým přenosem dat pro výměnu informací s řídicími systémy. Definuje doprovodný standard, který umožňuje interoperabilitu mezi ochranným zařízením a zařízeními řídicího systému v rozvodně.
- IEC 60870-5-104 (2006) - Síťový přístup pro IEC 60870-5-101 používající normalizované transportní profily
  - Definuje doplňkový standard dálkového ovládání, který umožňuje interoperabilitu mezi kompatibilními zařízeními pro dálkové ovládání přes internetovou síť využívající protokoly TCP/IP.

Dále se budeme zaměřovat především na část normy IEC 60870-5-104.

1.1 IEC 60870-5-104

IEC 60870-5-104 (dále jen IEC-104) definuje společnou normu pro dálkové ovládání, využívá normy ze souboru IEC 60870-5. Specifikace v této části představují kombinaci aplikační vrstvy z IEC 60870-5-101 (dále IEC-101) a transportních funkcí poskytovaných sadou protokolů TCP/IP. U TCP/IP lze používat různé typy sítí obsahující X.25, FR (Frame Relay), ATM (Asynchronous transfer mode – Vysokorychlostní síťová architektura) a ISDN (Integrated Services Digital Network – Datoví síť integrovaných služeb). To jsou však historické protokoly, v dnešní době převážně IEEE 802.3 (Ethernet). U TCP/IP je možno kombinovat použitím stejných definic různé ASDU (Application Service Data Unit – Jednotka dat aplikačního služby),

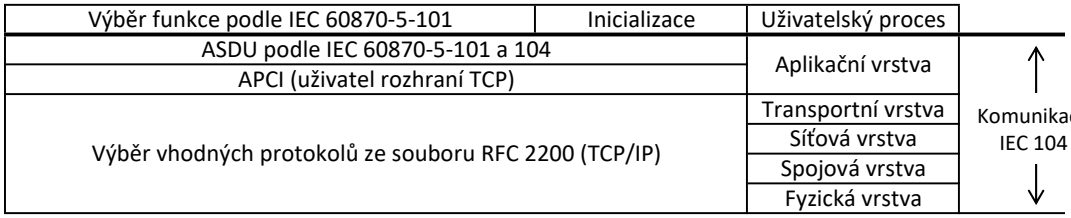
které jsou definované v jiných společných normách IEC 60870-5 (například 60870-5-102), toto však dále tato část neuvádí. Níže, na obrázku 1.1 je ukázkový příklad obecné architektury, kde řídicí a řízená stanice komunikují sítí klasického ethernetu a mají k dispozici záložní linku přes GSM (Global System for Mobile Communications). Dále následuje popis struktury protokolu objektů přenášených zpráv podle [3].



Obr. 1.1: Příklad architektury IEC-104 se zálohovanou linkou

1.1.1 Struktura protokolu

Níže uvedený obrázek 1.2 znázorňuje obecnou strukturu protokolu koncového zařízení vůči normalizovanému modelu OSI (Open Systems Interconnection – Propojení otevřených systémů). Vytváří se spojení IEC-104 a přenášená data jsou vybrané funkce ze souboru podle IEC-101. Tím jsou pak ovládány vzdálené procesy v reálném čase. K samotnému spojení IEC-104 se využívá služeb TCP/IP. K funkcím pro ovládání procesů blíže v [4].

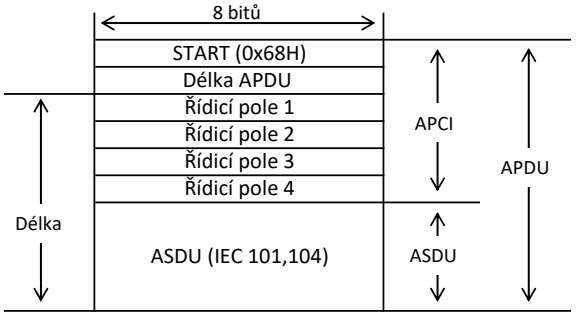


Obr. 1.2: Obecná struktura komunikačního koncového zařízení

1.1.2 APCI

Přenášené zprávy protokolu IEC-104 se nazývají APDU. Tyto zprávy obsahují dvě pole, hlavičku APCI a ASDU. APCI slouží k zajištění spolehlivosti spojení díky řídicím informacím v tomto poli. ASDU nese samotné funkce IEC-101. Přenášet se mohou dva typy APDU. APDU pevné a proměnné délky, viz obrázek 1.3 a 1.4.

Obr. 1.3: Formát APDU pevné délky



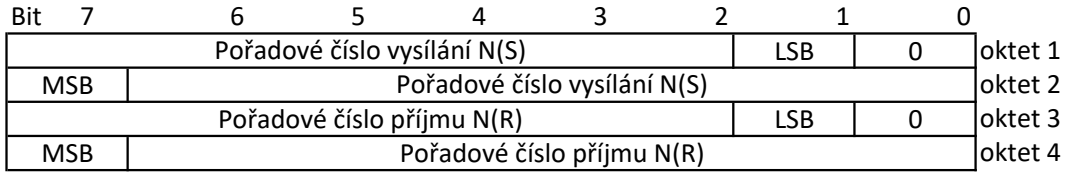
Obr. 1.4: Formát APDU proměnné délky

V případě přenosu APDU pevné délky, tedy pouze samotná hlavička APCI, je jím řízeno spojení IEC-104. Pokud jsou součástí APDU i data, tedy ASDU, jsou předána pro zpracování další aplikaci v řízené stanici. Struktura hlavičky zahrnuje hexadecimální hodnotu 0x68 identifikující začátek rámce, pole s délkou APDU v

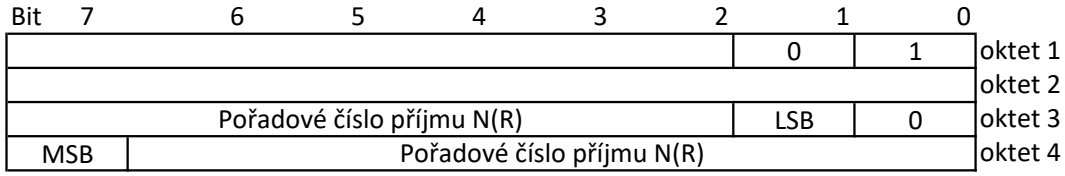
bytech a 4 B řídicích polí. V informaci o délce v druhém bytu jsou zahrnuta i řídicí pole hlavičky. Maximální hodnota délky je tedy 253 B (start byte a pole s délkou se nezahrnuje). Jednotlivá řídicí pole pak přenáší informace o číslování pro ochranu dat před ztrátou a duplicitou dat nebo řídicí funkce, budou popsané dále. Pro APDU obsahující data následuje za čtvrtým řídicím polem pole ASDU.

K rozlišení přenosu, zda se jedná o APDU pevné délky k řídicím účelům nebo o APDU s daty, slouží tři formáty rámců. Struktury hlaviček těchto rámců lze vidět na obrázcích 1.5, 1.6 a 1.7. Formáty jsou:

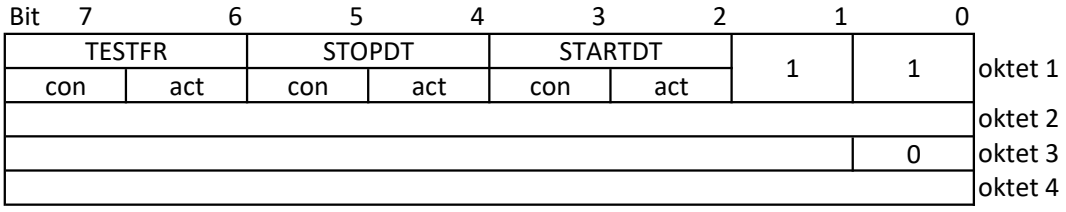
- I-formát - hlavička obsahuje číslování rámců, za hlavičkou následují data (ASDU)
- S-formát - potvrzovací rámec APDU pevné délky
- U-formát - přenos nečíslované řídicí funkce APDU pevné délky.



Obr. 1.5: I-formát APCI



Obr. 1.6: S-formát APCI



Obr. 1.7: U-formát APCI

Jednotlivé formáty se rozlišují prvními dvěma bity v prvním bajtu řídicího pole každého APDU od LSB, jak je vidět na obrázcích 1.5, 1.6 a 1.7 výše.

Ochrana proti ztrátě a duplicitě dat

K ochraně proti ztrátě a duplicitě dat slouží bajty s pořadovým číslem vysílání a přijmu v I-formátu a S-formátu viz obrázky 1.5 a 1.6. Hodnoty čítače pro vysílání i příjem se zvyšují s odeslanými nebo přijatými zprávami APDU. Vysílací strana si uchovává v paměti zatím nepotvrzené rámce. Úspěšně přijaté rámce se potvrzují každý zvlášť nebo několik současně[1], vždy však nejvyšší hodnotou úspěšně přijatého rámce. Vysílací strana si pak uvolňuje z paměti ty rámce, jejichž pořadové číslo příjmu je <= počtu přijatých rámců I-formátu. V případě delšího přenosu pouze z jednoho směru je druhou stranou vyslán S-formát, aby na vysílací straně nedošlo k přeplnění paměti nebo časové prodlevě viz dále. Toto potvrzování je použito na obou stranách komunikace.

Řízení přenosu STARTDT a STOPDT

K řízení přenosu slouží informace v hlavičce U-formátu, viz obrázek 1.7. Jednotlivé bity v hlavičce signalizují zahájení či ukončení přenosu nebo testování spojení. Vždy je vyslána výzva v podobě bitu act příslušného pole STARTDT, STOPDT nebo TESTFR a je očekáváno jeho potvrzení bitem con. Testováním spojení je zde myšleno udržení navázaného spojení, aby nedošlo k jeho ukončení způsobené časovou prodlevou.

1.1.3 ASDU

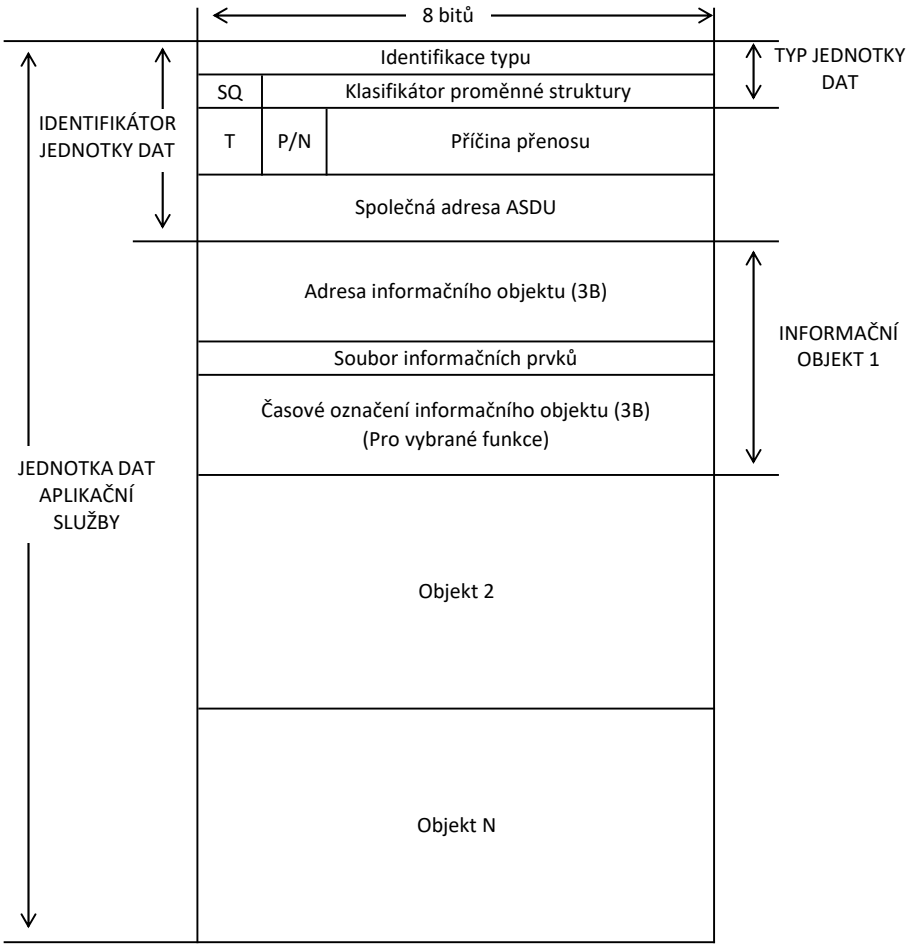
Data přenášená protokolem IEC-104 jsou umístěna v poli ASDU. Zde se přenáší jednotlivé funkce definované IEC-101. Skládá se ze dvou hlavních částí. První je identifikátor jednotky dat složen z přenášeného typu funkce, klasifikátoru struktury, příčinou přenosu a společné adresy ASDU. Identifikátor jednotky dat může mít celkově[2] 4-6 B. Druhá část jsou již přenášené funkce ve formě objektů. Těchto objektů může být umístěno v jednom ASDU až 127. V každém objektu se nachází jeho adresa a volitelně informační prvky, povely, měřené hodnoty, kvalitativní bity, časové razítka a jiné prvky v závislosti na vybrané funkci. Bližší popis všech funkcí v [4]. Na obrázku 1.8 jsou znázorněna jednotlivá pole struktury ASDU [3]. Úkolem této práce není implementace funkcí z IEC-101, proto budou vybrané pouze tři funkce pro demonstraci funkčnosti modulu s IEC-104. Tyto funkce jsou:

[1]Počet rámců, které lze potvrdit najednou definuje parametr w viz [3]. Výchozí hodnota je nastavena na 8 rámců.

[2]Je možné přidat 1 B k příčině přenosu, který bude identifikovat původce volání. Také lze použít 1 nebo 2 B pro společnou adresu ASDU. Tato volba je pevně daná pro systému.

Typ	Popis	Označení
<13>	:= měřená hodnota, krátké číslo s pohyblivou řádovou čárkou	M_ME_NC_1
<45>	:= jednoduchý povel	C_SC_NA_1
<102>	:= příkaz čtení	C_RD_NA_1

Tab. 1.1: Tabulka s výběrem funkcí z IEC-101 použitých v této práci



Obr. 1.8: Struktura ASDU

1.2 Komunikační procedury

V normě [3] je definováno základní chování při komunikaci řídicí a řízené stanice. V této kapitole je základní popis, jak taková komunikace probíhá a jsou znázorněny důležité situace. To zahrnuje inicializaci spojení, zahájení a ukončení přenosu dat a ukončení spojení. Řídicí stanice je klient, řízená stanice je server. Navázat spojení může jak klient, tak server[3]. Server naslouchá na portu TCP 2404. Spojení obvykle

[3]V případě inicializace spojení ze strany serveru musí být pevně stanovený parametr IP adresa a port a daný systém tuto možnost musí mít implementovanou.

inicializuje klient. Ten si standardně volí porty libovolně z dynamického rozsahu (49152-65535). Po úspěšné inicializaci klient ihned zahajuje přenos zasláním STARTDT act. Spojení trvá nezbytně nutnou dobu pro přenos. Ukončit spojení mohou jak klient, tak server, viz obrázek 1.9. V horní polovině obrázku je také znázorněno navazování TCP spojení ze strany klienta a ukončení spojení z obou stran a v dolní polovině pak zahájení vícenásobného spojení od klienta. Aktivní a pasivní ukončení znamená, že jedna strana zavře soket a druhá jej musí akceptovat, což má za následek čtyřcestný handshake, jak je vidět na obrázku 1.9. V případě, že je mezi stranami aktivní přenos, jedna ze stran, která chce ukončit spojení je nucena nejprve vyslat ukončovací signál STOPDT act a teprve potom může uzavřít soket (toto na obrázku není vyznačeno). Druhá strana nemá jinou možnost než ho potvrdit STOPDT con a uzavřít soket.

1.2.1 Vícenásobná spojení

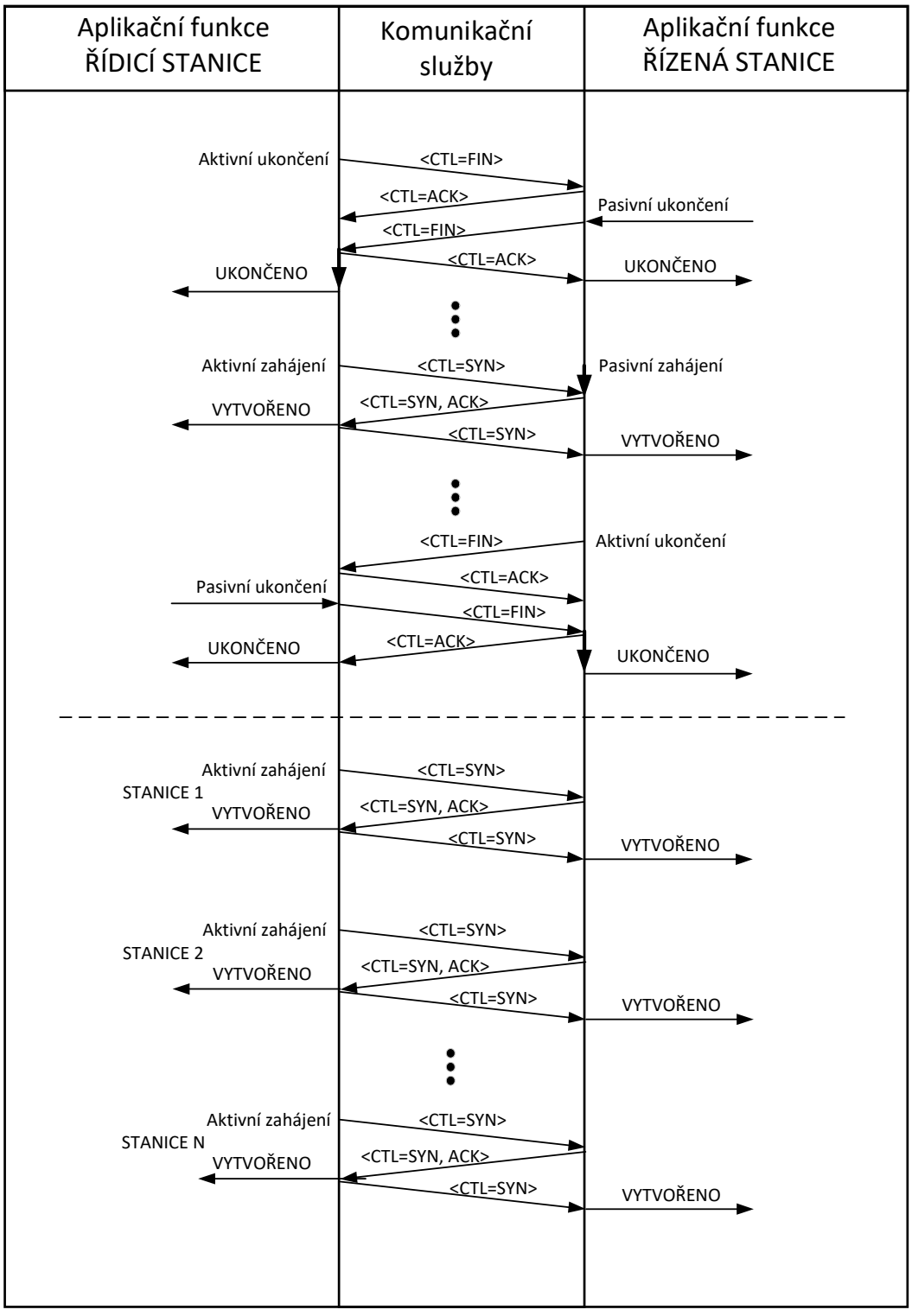
Tato norma dále definuje vícenásobná spojení. Není definovaný omezený počet těchto spojení, avšak reálnou hranicí je kapacita daného systému. Pro vícenásobná spojení platí, že přenos dat (inicializován STARTDT act bitem) probíhá pouze po jednom spojení a ostatní se udržují pouze navázaná (testovaná TESTFR act a TESTFR con). Během spojení mohou nastat výpadky, které mají za následek ukončení spojení. Důvodem může být nespolehlivý spoj nebo chybně nastavené parametry pro spojení, jako jsou například krátké časové prodlevy. Inicializaci více spojení a následné přepínání mezi nimi, ať už při výpadku nebo vynucením softwarově, je vyvoláno vždy řídicí stanicí. Blíže o záložních spojení a zálohovaných skupinách v [3]. V rámci této práce je v implementaci zahrnuta podpora vícenásobných spojení.

1.2.2 Časové prodlevy

Pro účely zajištění stability komunikace jsou definované čtyři typy časových prodlev:

- Timeout t0 - definuje délku doby navazování TCP spojení, po jeho uplynutí, během něhož se spojení úspěšně nenavázalo (důvodem může být, restart vzdálené strany), je navazování spojení zrušeno a po určité době[4] se pokus opakuje.
- Timeout t1 - časová prodleva nečinnosti (bez přijaté zprávy - U-formát, I-formát, S-formát), po jeho uplynutí je vyvoláno aktivní ukončení
- Timeout t2 - časová prodleva pro vyslání S-formátu rámce v případě, že se ve vyrovnávací paměti nachází nepotvrzené rámce
- Timeout t3 - po uplynutí je vyslán TESTFR act pro udržování spojení

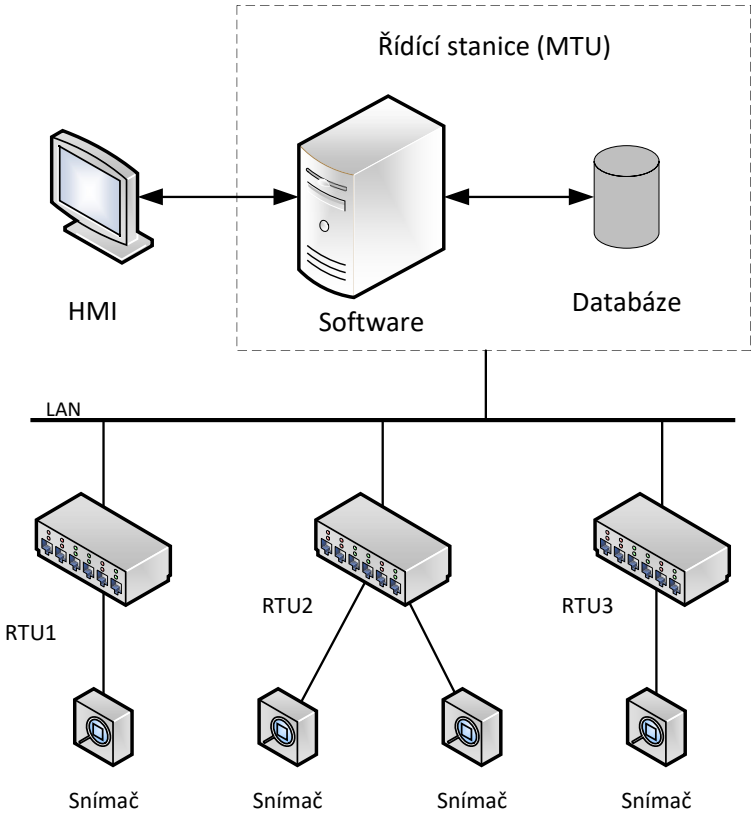
[4] V normě [3] není specifikováno.



Obr. 1.9: Ukázka navazování a ukončení spojení

2 SCADA systémy

SCADA je pojem zahrnující sloučení softwarových programů a hardwaru, které dohromady tvoří systém pro vzdálenou dispečerskou činnost. Hardware tvoří MTU neboli řídicí stanice a RTU neboli řízená stanice. Softwarové vybavení se skládá z HMI, databáze a SCADA softwaru. Toto spojení HW a SW poskytuje ohromnou výhodu centrálního řízení a sběr dat pro rozsáhlé a komplexní systémy. V této práci přiblíženo k ovládání částí distribuční sítě a rozvoden v energetickém průmyslu. Tento systém však může být použit i v jiných odvětvích. Na trhu se objevuje velká řada, jak dodavatelů samotné technologie, tak softwarových vývojářů, kteří poskytují tyto systémy. Příkladem mohou být Promotic, SCADA servis, ELVAC, Mervis a mnoho dalších. Pro vzdálené řízení a sběr dat využívají více komunikačních protokolů. Tato práce je zaměřena na protokol IEC-104.



Obr. 2.1: Ukázka zapojení topologie SCADA systému podle [1]

Obrázek 2.1 slouží jako příklad SCADA systému, na kterém je znázorněno HMI formou monitoru pro jednoduché ovládání systému člověkem. V rámci řídicí stanice

je instalovaný program společně s lokální databází. Přes rozhraní LAN[1] jsou připojené řízené stanice RTU, které slouží ke sběru dat ze snímačů a poskytování těchto hodnot pro MTU. Snímače reprezentují nějakou reálnou měřenou veličinu. Místo snímače by mohly být i ovládané prvky jako třeba pohon, relé a jiné.

Dále v této práci mohou být použity pojmy MTU, klient nebo SCADA, všechny však znamenají řídicí stanici. Naopak řízenou stanicí jsou myšleny RTU, server nebo taky slave zařízení. Jednotlivá názvosloví pocházejí buďto z oblasti komunikačních architektur nebo z oblasti průmyslových automatizačních systémů, principiálně však popisují ta stejná zařízení.

2.1 RTU

Zařízení pro vzdálené řízení, měření nebo sběr dat. Může vystupovat v mnoha oborech průmyslové automatizace, zejména v energetice. V sobě má řídicí procesorovou jednotku, implementované programovací logické funkce a programovatelné vstupy a výstupy. Zpravidla obsahuje více komunikačních rozhraní pro komunikaci s jinými typy zařízení nebo zařízeními jiných výrobců. V energetice vystupuje RTU jako koncové komunikační zařízení, které vykonává řídicí signály. Řídicími signály mohou být povely pro ovládání vstupů a výstupů nebo pro ovládání vnitřních stavů procesů. Jiné názvy pro toto zařízení mohou být také komunikační jednotka nebo koncentrátor dat. Rozlišují se dva směry komunikace. Směr řídicí je od SCADA aplikace a směr monitorovací opačný.

2.1.1 Využití s IEC-104

RTU mohou mít v sobě zabudováno více protokolů pro komunikaci. Jedním z nich je právě IEC-104 používaný především v energetice pro automatizaci distribučních sítí. RTU čtou stavy vypínačů, odpojovačů a alarmů ze vstupů a ovládají vypínací prvky, přepínají odbočky trafostanic pro regulaci napětí a přepínají na záložní trasy v případě výpadku VN. Dále měří napětí, proudy, výkony, účiníky a další veličiny pro řízení sítě. Celý proces řídí SCADA systémy, které centrálně řídí a monitorují síť, vyhodnocují statistiky a vizualizují data. V rozvodnách se běžně používají dva typy schémat zapojení RTU [5].

[1]Zde není myšleno LAN jen jako (Local Area Network – Místní síť), protože ovládané RTU SCADA systémem mohou být rozmístěné po celém světě, ale jako síť pod jednou správou, například pomocí tunelů VPN.

Výhody použití RTU systémů

- Funkce jednotlivých RTU poskytují možnost zapojení centralizovaným nebo necentralizovaných systémem. Liší se robustností konkrétního řešení.
- Jedná se o certifikovaná zařízení splňující vysoké požadavky pro využití v distribuční soustavě.
- Podporují mnoho komunikačních protokolů a rozhraní pro konkrétní účely.
- Nabízí možnost připojení záložní linky, v dnešní době nejčastěji LTE modem.
- Mohou zastávat řídicí funkci nebo jen jako datový koncentrátor.
- Možnost automatizace některých havarijních funkcí (přepnutí na záložní linku).

2.1.2 Příklad RTU jednotky od společnosti ELVAC

Příkladem RTU je jednotka RTU7MC3-D od společnosti ELVAC. Společnost se zabývá poskytováním inženýrsko dodavatelských služeb v oblasti průmyslové automatizace, průmyslových a speciálních PC systémů, silnoproudé elektrotechniky a jednoúčelových strojů. Velká část produktů této společnosti pochází z vlastního vývoje. RTU7MC3-D je pokročilá univerzální komunikační jednotka s vestavěným počítačem s Linuxovým jádrem. HW jednotky je specializovaný pro použití v rozvodnách a jiných objektech pro distribuci elektrické energie. K dispozici jsou dva nezávislé ethernetové porty 10/100Mbps s podporou VLAN. Dvě rozhraní RS-485 a jedno RS-232 jako konzola. Dále integrovaný LTE modem s dvěma anténami. Jsou implementovány sady komunikačních protokolů IEC 60850, IEC 60870-5-101, IEC 60870-5-103, IEC 60870-5-104, DNP3, MODBUS TCP/RTU, DLMS, OPC UA, SNMP a další pokročilé protokoly pro komunikaci přes počítačovou síť. Jednotka je dále vybavena čtyřmi digitálními vstupy a také slotem na SD kartu. Je uzpůsobená pro montáž na DIN lištu nebo svisle či vodorovně na panel v elektrické skříni. viz katalogový list [6]. Na obrázku 2.2 je fotografie jednotky ze stránek dodavatele [5].

Obr. 2.2: RTU7MC3-D od společnosti ELVAC

3 Návrh modulu pro komunikaci IEC-104

Cílem této kapitoly je zpracování návrhu pro následnou implementaci komunikačního protokolu IEC-104 pro PLC kontrolér unipi Patron M567. Využití souhrnu poznatků předešlých kapitol, samotné normy [3] a následné řešení aplikace.

3.1 unipi technology

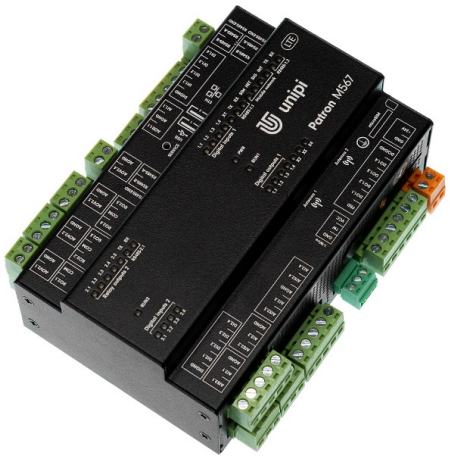
Společnost zabývající se vývojem programovatelných logických kontrolérů (PLC), IoT zařízení, různých senzorů a dalších systémů v tomto oboru. Využitelnost jejich produktů je široká. Uplatní se v menších aplikacích jako je smart home, ve větších pak v řízení budov BMS (Building Management System – Řídicí systém budov), průmyslu nebo ve vlastních projektech. unipi vyvíjí řady kontrolérů Neuron, Patron, Iris, unipi 1.1, Gate nebo další různé rozšiřující moduly a brány [7] sloužící ke konkrétním aplikacím. Jednotlivé řady se liší vzhledem, velikostí, instalovaným softwarem, a především hardwarovou výbavou. Všechny však spojuje jádro operačního systému postavené na Linuxu. To činí tyto systémy ojedinělými a robustními pro své aplikace. Díky otevřenosti kódu si každý může finální konfiguraci přizpůsobit dle potřeby.

3.1.1 unipi Patron M567

Pro naši aplikaci poslouží jednotka unipi Patron M567. Je to programovatelný logický kontrolér postavený na vlastním výpočetním modulu společnosti unipi technology. Možnosti jeho využití jsou v automatizaci, ovládání, monitorování či MaR. Disponuje 4jádrovým procesorem i.MX 8M Mini (Arm Cortex A53) o frekvenci 1,8GHz, RAM 1 GB a 8 GB eMMC flash paměti. Tyto hlavní parametry z něj činí velmi výkonnou programovatelnou jednotku. Další předností je softwarové vybavení. Jádro této jednotky je postaveno na Linuxu, konkrétně Debian 12, který je dále speciálně upraven a vybaven ovladači pro různé typy vstupně výstupních periferií, viz tabulka 3.1. Ve výchozím nastavení běží nad Linuxem softwarová platforma Mervis, která slouží pro tvorbu programů pro ovládání a automatizaci výše zmíněných systémů. Tato platforma je uzavřená, co se týče rozšiřování o vlastní implementace a málo flexibilní pro vývoj nových technologií a architektur. Například neumožňuje napojení na Grafanu, která je v poslední době velmi populární. Díky tomu Mervis nebude v této práci využit. Možnost, která bude využita, a která jednotky unipi dělá ojedinělými je vlastní softwarové řešení. Tato možnost spočívá v tom, že vlastní implementace využívá holý operační systém společně s ovladači k HW. V tabulce níže je souhrn specifikací jednotky. Uvedené informace jsou ze stránek dodavatele [7].

Tab. 3.1: Parametry unipi Patron M567

Souhrn specifikací:	
CPU	i.MX 8M Mini (Arm Cortex A53)
Jader	4
Frekvence CPU	1,8 GHz
RAM	1 GB
Flash paměť	8 GB
OS	Debian GNU/Linux 12 (bookworm)
I/O	
DI	8
DO	4
RO	5
AI	5
AO	5
Komunikační rozhraní	
Ethernet	1
USB	2
RS485	3
RS232	1
1-wire	1
LTE modem	1



Obr. 3.1: Jednotka unipi Patron M567

3.2 Popis funkce

Pro demonstraci je aplikace přirovnána k reálnému využití v řízení rozvodny. Řídicí stanice nebo SCADA je program běžící na PC v řídicím centru s přístupem pro obsluhu. RTU nebo také řízená stanice, v našem případě Patron M567, je v elektrickém rozvaděči se zapojenými periferiemi typicky v centrálním bodě rozvodny. Řídicí a řízená stanice jsou propojeny prostřednictvím internetové sítě.

Obsluha chce přistoupit k RTU a vyčíst z nich data nebo sepnout zátěž. Využívá k tomu vizuální rozhraní SCADA systému. SCADA vytvoří požadavek ze souboru funkcí IEC-101 na vyčtení dat nebo s povelem sepnutí a pomocí svého klienta IEC-104 ho odešle na RTU. Na RTU běží server IEC-104, který čeká na přijetí dat od klienta. Server přijatá data předá aplikaci, která má na starosti obsluhu IEC-101. Ta může nebo nemusí vygenerovat odpověď s daty a předat je opět modulu IEC-104 k odeslání zpět do SCADA systému. V reálné aplikaci by to mohli být požadavky např. pro vyčtení elektroměru, vyčtení stavů pojistek, různých ochran, spuštění nějakého pohonu apod.

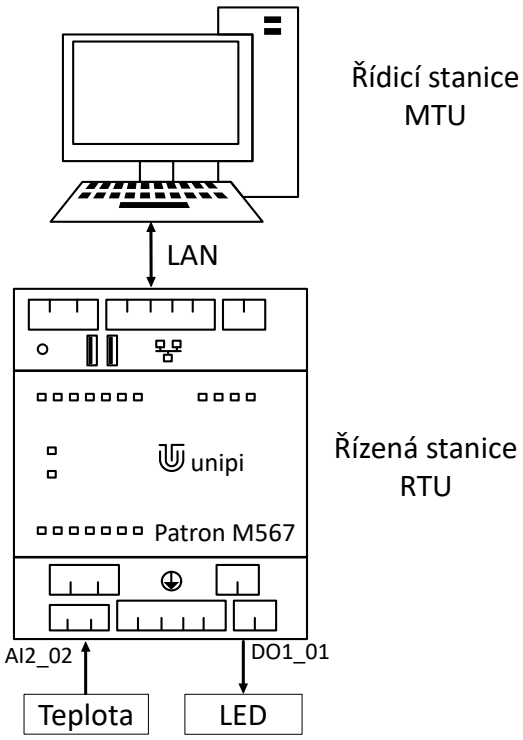
Pro demonstraci v této práci budou použity klient IEC-104[1] s pevně zadanými daty (požadavky ze souboru funkcí IEC-101 viz tabulka 1.1), které budou odesílány do Patrona s běžícím serverem IEC-104. Na Patronu se data zpracují a vygenerují se odpovědi. Vyčtená hodnota bude datového typu float z analogového vstupu a ovládaným prvkem bude LED dioda na digitálním výstupu.

3.2.1 Fyzická topologie

Fyzickou topologii tvoří jednotka unipi Patron M567 v roli řízené stanice RTU a PC s modulem klienta IEC-104, v roli řídicí stanice MTU. Fyzické zapojení příkladu je znázorněno na obrázku 3.2.

K RTU jsou zapojeny ovládané periférie, tedy LED dioda a snímač teploty. MTU a RTU jsou propojeny napřímo datovým kabelem. Ovládaná LED dioda je připojena na digitální výstup DO1_01 a teplotní čidlo na analogový vstup AI2_02.

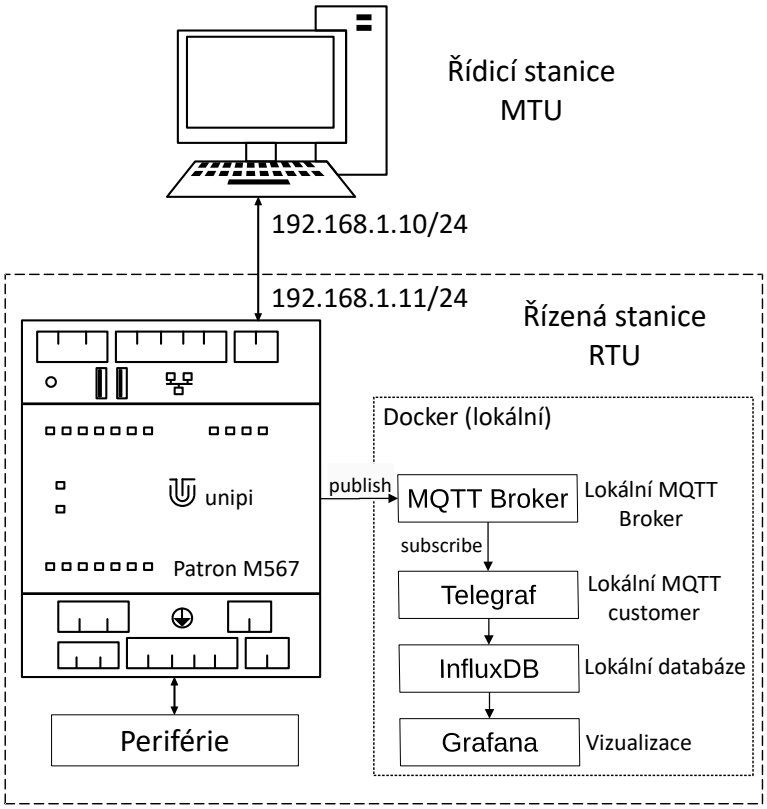
[1]Klient není předmětem této práce, proto zde vystupuje pouze jako samotný modul klienta IEC-104. Klient je vyvýjen společně se serverm viz 4.4.



Obr. 3.2: Fyzická topologie

3.2.2 Logická topologie

Pro účely návrhu budou jednoduše obě vystupující stanice zapojeny ve stejné místní síti LAN. Logická topologie i se znázorněnými IP adresami je na obrázku 3.3. IP adresa řídicí stanice bude 192.168.1.10, a řízené stanice IP 192.168.1.11. Řízená stanice naslouchá standardně na portu 2404 a čeká na zahájení spojení od klienta. Ten volí porty dynamicky dle standardního připojení přes TCP. Na obrázku lze vidět i tok dat do MQTT Brokeru a z něj do dalších nástrojů, blíže jsou popsané v další kapitole 3.2.3.



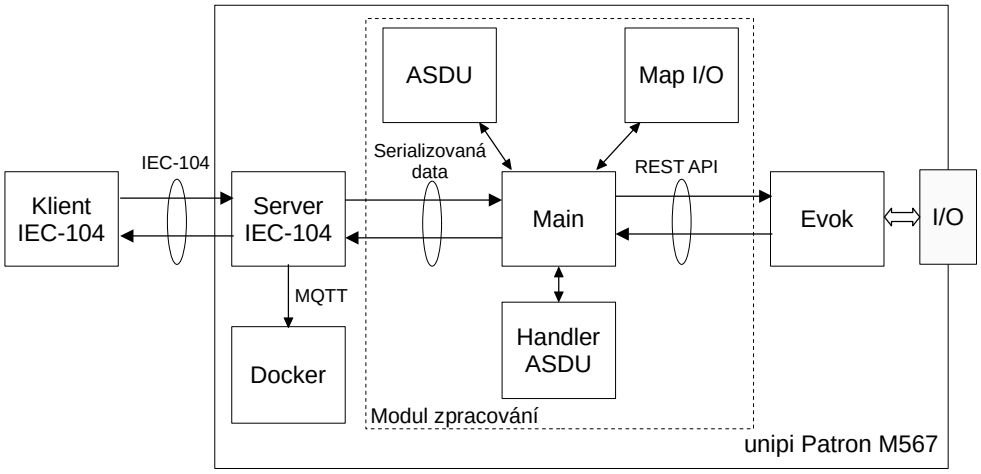
Obr. 3.3: Logická topologie

3.2.3 Aplikační vrstva

Úkolem aplikační vrstvy je převzít přijaté rámce a podle informací v APCI rozlišit jednotlivé rámců, zda se jedná o řídicí nebo datové rámce. Přenášená data jsou vybrané funkce definované v IEC-101. V praktické ukázce budou data předána do modulu zpracování, který dekóduje vybrané funkce, přiřadí jim mapovaný I/O a provede akci spojenou s vyčtením teploty nebo sepnutí LED. Na obrázku 3.4 je zobrazené navržené blokové schéma všech modulů uvnitř jednotky unipi Patron M567 a naznačena jejich spolupráce a výměna dat. Jednotlivé bloky budou popsány v kapitole 4.2.6.

Modul serveru zároveň posílá příchozí požadavky do MQTT Brokera, odkud si je dále odebírá Telegraf a ukládá je do databáze InfluxDB. Z databáze InfluxDB jsou data stahovány nástrojem Grafana k vizualizaci. MQTT Broker, Telegraf, InfluxDB a Grafana běží ve virtualizačním nástroji Docker jako kontejnery pro jejich snadnější správu. Kvůli omezeným HW zdrojům jednotky unipi Patron a povaze vnitřního úložiště není tato volba vhodná k reálnému použití. V praxi by databázový systém byl umístěn v datacentru nebo v cloudu, avšak v této práci postačí lokálně jako možnost kontroly úspěšně přenesených dat protokolem IEC-104. Volba

MQTT protokolu a MQTT Brokeru je v jejich jednoduchosti a nenáročnosti na výkon jednotky. Na broker se pak můžou jednoduše napojit i jiné aplikace, které mohou sbírat publikované zprávy k vlastnímu zpracování. Volba databáze InfluxDB je pro její výhodu optimalizovaného ukládání a analýzu časových řad ve spojení s Grafanou jako vizualizačním nástrojem. Protože InfluxDB nemá možnost přímého napojení na MQTT Broker, je mezi ním a databází vložen nástroj Telegraf. Tyto nástroje a způsoby, jak si mezi sebou předávají data, je zobrazeno na obrázku 3.3. Pro účely této práce běží nástroje lokálně v Dockeru. Mohou být ale umístěny i mimo jednotku, stejně jako je naznačeno na obrázku 3.3, např. v cloudu.



Obr. 3.4: Blokové schéma modulů uvnitř jednotky unipi Patron M567

4 Implementace - Praktické zpracování

Cílem této kapitoly je představit praktické zpracování a implemenaci softwarového modulu protokolu IEC-104 v roli řízené stanice (server) pro PLC kontrolér řady unipi Patron M567. Výsledek praktického zhotovení práce je funkční linuxový démon[1] naslouchající na portu 2404 a připraven komunikovat s klientem. V kapitole jsou uvedeny ukázky kódů důležitých metod, delší metody jsou vloženy mezi přílohy nebo jsou popsány vývojovými diagramy.

4.1 Přehled

Tento softwarový modul je serverová implementace protokolu IEC-104 v programovacím jazyce Python. Programovací jazyk Python byl pro tento projekt zvolen z důvodu jeho univerzálnosti, relativní snadnosti a široké dostupnosti příkladů, které usnadňují vývoj. Modul umožňuje komunikaci s klientskými zařízeními IEC-104. Data přenášená protokolem jsou zasílána prostřednictvím protokolu MQTT do MQTT Brokera a následně ukládány do databáze pomocí nástroje Telegraf, odkud jsou přístupná pro další aplikace. Kromě ukládání do databáze jsou příchozí data předávány k dalšímu zpracování, které je realizováno speciálním modulem pro zpracování pouze pro účely této práce. Celý modul IEC-104 je pak určen pro provoz na platformě Linux a je kompatibilní s verzí Pythonu 3.9 a vyšší.

4.2 Architektura

V této kapitole bude zobrazen přehled tříd, stručně popsány hlavní komponenty modulu, jejich obsluha v čase, zobrazen diagram datového toku z pohledu příchozích dat, stavový diagram spojení a představen modul zpracování.

4.2.1 Přehled tříd

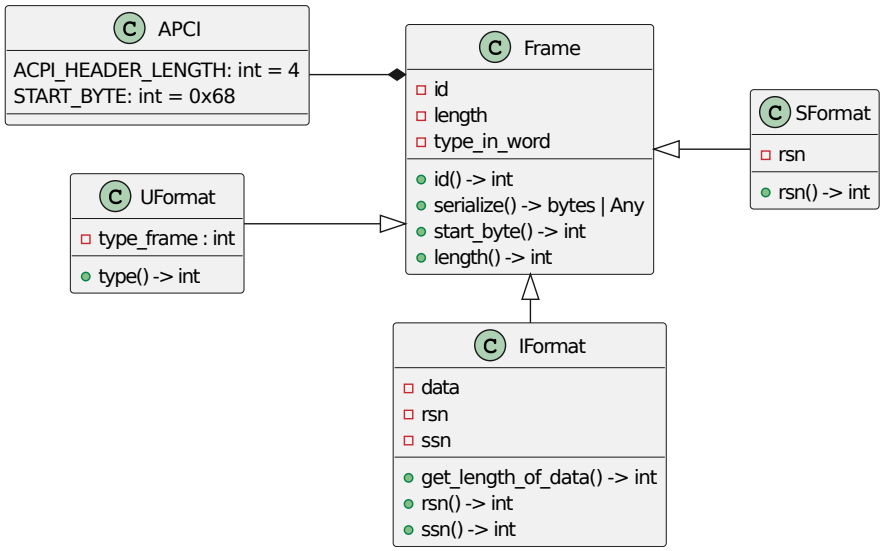
Celý modul je rozdělen do následujících tříd:

- *APCI - Formáty řídicích informací v hlavičce přenášeného rámce*
- *ClientManager - Spravuje všechna spojení mezi klientem a serverem*
- *ConfigLoader - Načítá parametry z konfiguračního souboru*
- *Frame - Definuje společné atributy pro jednotlivé rámce*
- *IDataSharing - Rozhraní pro ukládání přenášených dat*
- *IFormat - Definuje parametry I-formátu*
- *MQTTProtocol - Zasílání přenášených dat do MQTT Brokera*

[1] Démon je označení běžící služby na pozadí v unixových systémech.

- *PacketBuffer - Vyrovnávací fronta pro potvrzování přenesených rámců*
- *Parser - Stará se o vytvoření objektů pro jednotlivé formáty rámců*
- *ServerIEC104 - Udržuje server pro přijetí nových spojení od klienta*
- *Session - Zajišťuje příjem a odesílání rámců*
- *SFormat - Definuje parametry S-formátu*
- *ConnetionState - Definuje vnitřní stav spojení*
- *TransmissionState - Definuje vnitřní stav přenosu*
- *Timer - Signalizuje vypršení jednotlivých časových intervalů*
- *UFormat - Definuje parametry U-formátu*

Z nichž ServerIEC104, ClientManager a Session jsou hlavními komponenty modulu. Na obrázku 4.1 je vidět menší část objektového modelu tříd, který zachycuje využití dědičnosti pro objekt rámce. Zbytek objektového modelu je součástí přílohy A.1. V něm je možné nalézt ostatní třídy, kromě těch na obrázku 4.1 a jejich závislosti. Třídy v obou objektových modelech nezobrazují všechny své atributy a metody, jen ty důležité, z důvodu zjednodušení a přehlednosti.



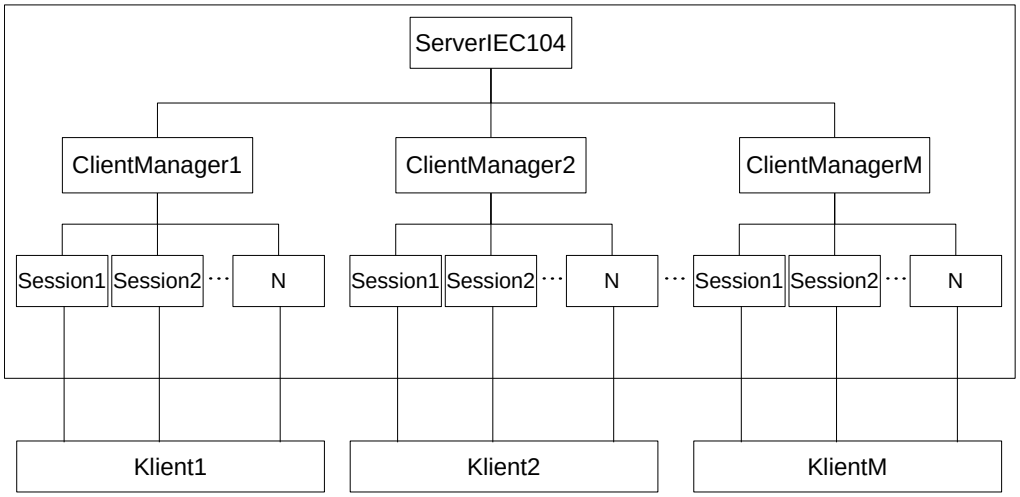
Obr. 4.1: Objektový model

4.2.2 Hlavní komponenty

Hlavními komponenty jsou třídy:

- ServerIEC104 - Vytváří soket pro naslouchání na TCP portu 2404 a obsluhuje příchozí spojení od klienta
- ClientManager - Spravuje všechna spojení mezi klientem a udržuje vnitřní stavy komunikace jako jsou počítadla a časovače
- Session - Má v sobě metody pro přijetí a odeslání rámců

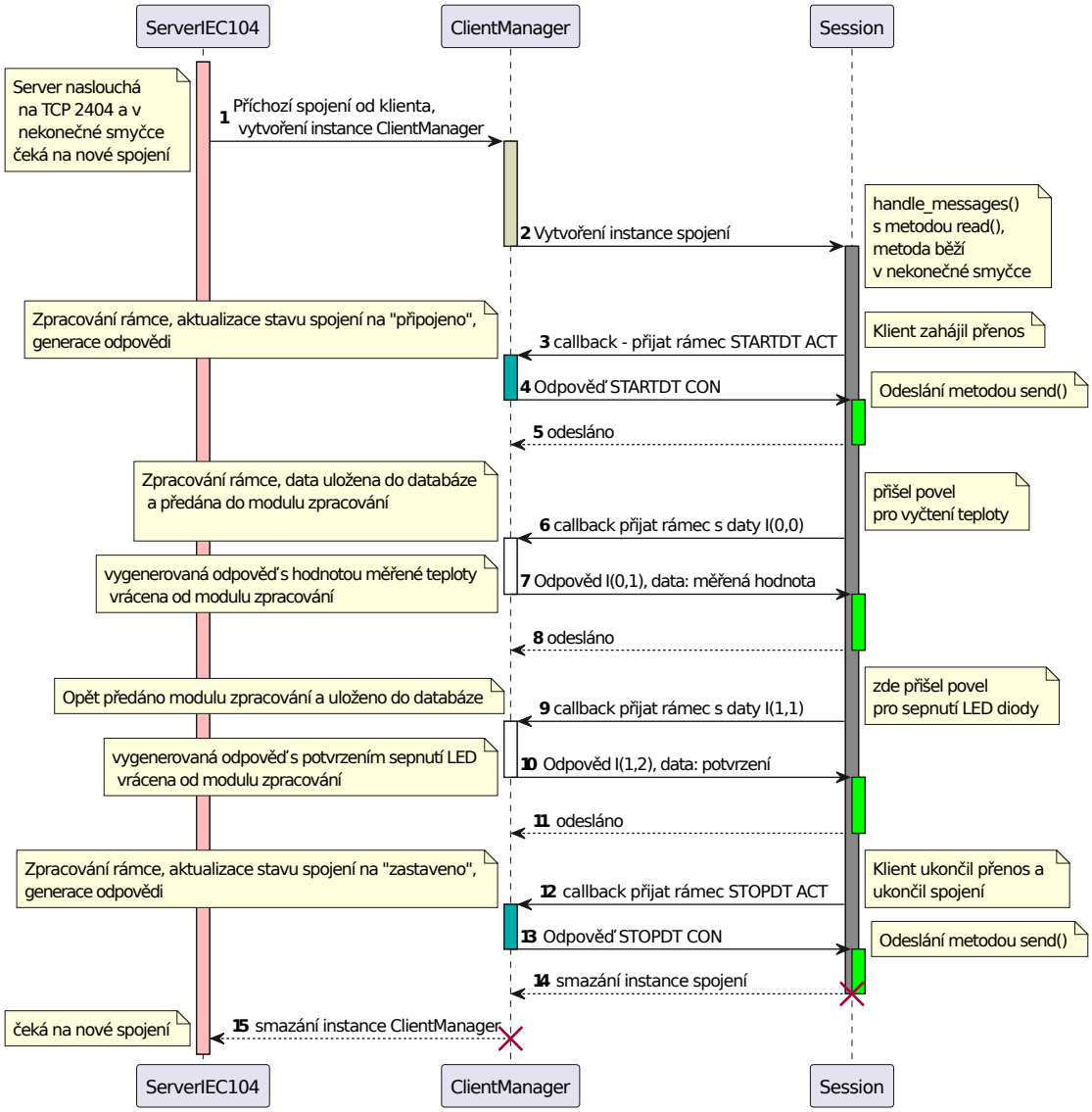
Kód je spouštěn ve třídě *ServerIEC104*. Po jeho spuštění je vytvořen soket, který naslouchá na portu 2404 a čeká na příchozí spojení od klienta. Jakmile se klient připojí, jsou mu vytvořeny instance tříd *ClientManager* a *Session*. Instance *ClientManagera* představuje samotného klienta, kdežto instance třídy *Session* zastupuje spojen. Spojení mezi klientem a serverem může existovat více, avšak instance *ClientManagera* je pro jednoho klienta pouze jedna. Klientů může být připojeno více. Instance *ClientManagera* v sobě udržuje počítadla příchozích rámců a udržuje seznam navázaných spojení. Instance *Session* přijímá rámce od protější strany a předává je k dalšímu zpracování instanci *ClientManagera*. Uvnitř *ClientManagera* poté proběhne aktualizace stavů spojení, případně je vygenerována příslušná odpověď a předána zpět instanci *Session* k odeslání. Na obrázku 4.2 jsou znázorněné vytvořené instance dle předešlého popisu chování aplikace.



Obr. 4.2: Ukázka vytvořených instancí tříd pro více připojených klientů

4.2.3 Sekvenční diagram

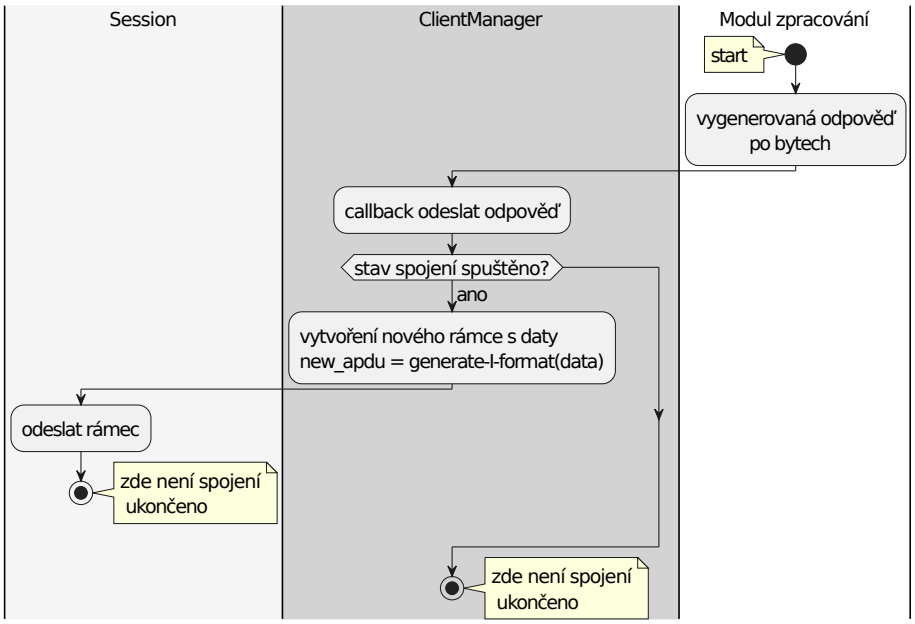
Z pohledu času je běh kódu zmíněných hlavních komponent, a jejich metod, znázorněn v sekvenčního diagramu na obrázku 4.3. V diagramu jsou zobrazeny dvě instance, ve kterých běží smyčky zpracování událostí. Mají podobu oranžového a šedého obdélníčku ve třídách *ServerIEC104* a *Session*. Aby bylo možné provádět smyčky zároveň, je použit asynchronní přístup pomocí knihovny AsyncIO. Detailněji o AsyncIO je psáno v kapitole 4.3.2. V této ukázce je demonstrován scénář s příchozím povelem od klienta pro vyčtení teploty z jednotky a vzápětí povel pro sepnutí LED, scénář viz 3.2. V komentářích v sekvenčním diagramu je snaha zachytit rozhodování a akce s daty v daném čase a místě.



Obr. 4.3: Sekvenční diagram hlavních komponent modulu

4.2.4 Datový tok

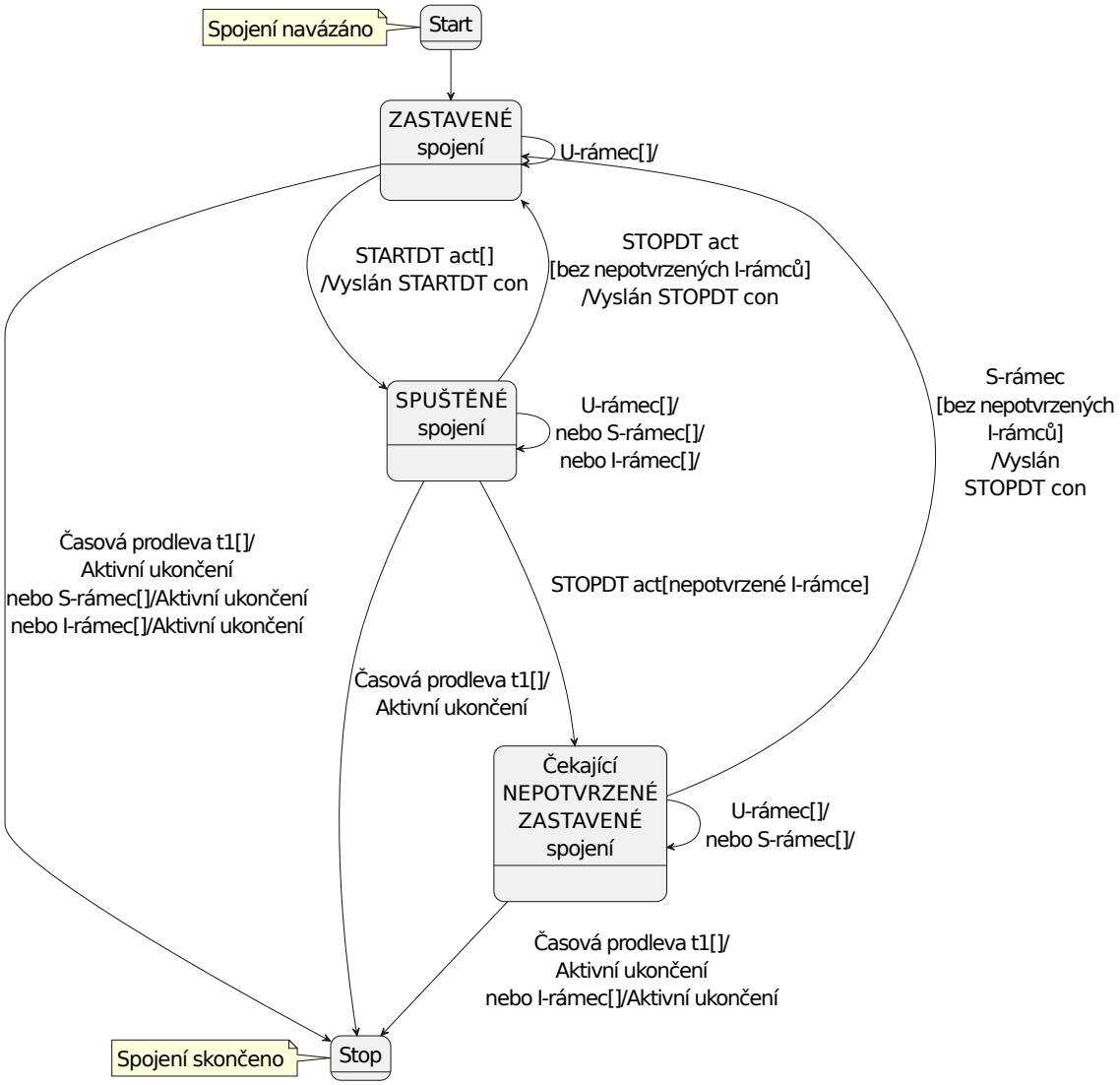
Z pohledu příchozí dat ze spojení ve formě rámců je znázorněn jejich datový tok formou vývojového diagramu. Pro příchozí spojení je diagram výrazně rozsáhlejší. Z tohoto důvodu je vložen do přílohy B.1. Je zde je vidět rozhodování kudy data poteču. V diagramu je znázorněn začátek vykonávání černým puntíkem ve třídě *Session* a hned následuje podmínka, zda je nastaven příznak timeout. Nutno připomenout, že v této metodě běží while cyklus právě s podmínkou, která kontroluje, zda nastal některý z timeoutů. Teprve poté následuje funkce pro čtení dat z příchozího spojení. Třetí sloupec označuje dvě třídy, *MQTTProtocol* a Modul zpracování. Měli by být rozdělené do vlastních sloupců, ale z důvodu čitelnosti textu bylo potřeba zachovat šířku diagramu na velikost stránky A4. Pro rozlišení metod, do které třídy patří je použito barevné označení. Zde na obrázku 4.4 je pak znázorněn vývojový diagram pro zpětný směr, vygenerovanou odpověď vrácenou od modelu zpracování.



Obr. 4.4: Vývojový diagram pro odpověď

4.2.5 Stavový diagram

Popisovat modul jako celek podle stavového diagramu nemá smysl. Modul se nachází stále ve stavu, kdy čeká na spojení. Jakmile přijde nové spojení, je mu vytvořena instance *Session* a ta předána instanci *ClientManagera*. Ovšem samotné spojení (instance *Session*) se již řídí dle stavového diagramu 4.5, který je identický stavovému diagramu řízené stanice v normě [3].



Obr. 4.5: Stavový přechodový diagram spojení - řízená stanice

4.2.6 Modul zpracování

Tato práce je zaměřená na vývoj modulu IEC-104, nikoliv na zpracování přijatých dat a vykonávání funkcí ze souboru IEC-101. Modul IEC-104 by však měl umět nejen řídit stav komunikace na základě přijatých rámců, ale být i schopen vrátit odpověď. Z toho důvodu byl vyvinut i modul zpracování, který staticky implementuje vybrané funkce z tabulky 1.1. Díky tomu je schopen generovat odpovědi pro účely demonstrace popsané v kapitole 3.2. Tyto odpovědi jsou přenos měřené teploty na základě příchozího požadavku na čtení. A druhým je potvrzení, zda byla sepnuta nebo vypnuta LED dioda na výstupu jednotky, opět na základě příchozího požadavku povelu aktivace/deaktivace.

Blokové schéma modulu zpracování je na obrázku 3.4. Moduly klient a server IEC-104 udržují spojení a přenáší mezi sebou data kanálem IEC-104. Modul zpracování v sobě zahrnuje bloky ASDU, Map I/O, Handler ASDU a hlavní blok Main, který je všechny spojuje. Blok ASDU serializuje a deserializuje data (jednotlivé funkce). Map I/O vrací informace o mapování na konkrétní I/O piny. Tyto informace o mapování má uloženy v konfiguračním souboru typu JSON, ukázka konfiguračního souboru je ve výpisu v příloze C.1. Mapuje se podle adresy informačního objektu IOA[2] jeden k jednomu s I/O pinem jednotky. Další hodnotou v konfigurančím souboru je *command_method* sloužící identifikaci rozdílného ovladače pro I/O. Stejným způsobem by mohly být mapovány i vnitřní proměnné systému, což by zvýšilo jeho funkcionalitu. Evok je rozhraní pro ovládání I/O na jednotkách unipi[3]. Komunikace s ním probíhá pomocí dotazů REST API. Blok Handler ASDU implementuje akce spojené s konkrétními funkcemi z IEC-101. Dávalo by smysl tento blok rozdělit na dva. Jeden pro příchozí směr povelů a druhý pro odchozí. Důvodem je rozdílný význam funkcí ve směru ovládání od řídicí stanice a směru sledovací od řízené stanice. V této práci ale postačí pouze jeden takový modul. Pomocí těchto dílčích modulů jsou obslouženy příchozí požadavky funkcí z tabulky 1.1 a je vygenerována příslušná odpověď, která je následně odeslána zpět klientovi.

[2]IOA popsáno v kapitole 1.1.3.

[3]Evok poskytuje přístup k I/O periferiím nejen u řady Patron, ale i Neuron, Gate nebo unipi 1.1.

4.3 Implementační detaily

V této kapitole jsou blíže popsány vybrané funkce modulu s ukázkami jejich implementace. V modulu jsou použity dvě hlavní smyčky pomocí knihovny AsyncIO. Jedna čeká na nové spojení od klienta ve třídě *ServerIEC104* a druhá čeká na příchozí data od připojených klientů v metodě *handle_messages()* ve třídě *Session*. Tyto smyčky lze vidět v sekvenčním diagramu 4.3, vyznačeny jsou dlouhými obdélníčky (oranžový a šedý). Hlavní smyčky by při splnění určitých podmínek mohli běžet nekonečně dlouho, jelikož jejich vnitřní strukturu tvoří cyklus *while*. Podmínkami je myšleno splnění pravidel komunikace IEC-104, tedy stanice si stále vyměňují data a nedojde k časové prodlevě, která by ukončila přenos. Všechny ostatní akce v modulu se dějí pomocí funkcí zpětného volání[4], které tak zajišťují asynchronní chování aplikace. Tyto funkce jsou předávány většinou při vzniku samotné instance v argumentu, tedy v konstruktorech tříd, nebo v argumentu volání konkrétní metody.

4.3.1 Tvorba rámců

Tvorbu rámců zajišťují třídy *Frame* a z ní děděné třídy, které odpovídají jednotlivým formátům rámce, znázorněny jsou v objektovém diagramu na obrázku 4.1. Tyto třídy rámec zapouzdřují a dělají z něj objekt.

Odchozí směr

Tvorba objektu rámce je klasické volání konstruktoru s předáním informací v argumentu. Informacemi jsou pro každý formát jejich odpovídajícími vlastnostmi. Například při tvorbě rámce U-formátu je mu předán typ, o který se jedná, zobrazeno ve výpisu 4.1. Pro formáty rámců I-formát a S-formát je to stejné, akorát jim jsou předávány aktuální stavy čítačů.

Výpis 4.1: Vytvoření objektu rámce START act v jazyce Python.

```
1 def generate_startdt_act(self) -> UFormat:
2     return UFormat(APCI.STARTDT_ACT)
```

O serializaci objektu do podoby strukturovaného rámce, jak je definováno v normě [3], se stará metoda *serialize()* v každé ze tříd všech typů formátů. Na ukázku této metody je použita serializace S-formátu, výpis kódu v 4.2. Takto serializovaný rámec je připraven k odeslání.

[4]Ve výpisech kódu nazváno jako callback.

Výpis 4.2: Serializace S-formátu rámce v jazyce Python.

```
def serialize(self, rsn: int = 0) -> bytes:
    if rsn:
        self._rsn = rsn

    # přizpůsobení čísla do pole po bitech
    third = (self._rsn & 0x7F) << 1
    fourth = (self._rsn >> 7) & 0xFF

    # zabalí jednotlivá pole po bytech
    packed_header = struct.pack(f"{'B' * (self._header_length + 2)}",
            Frame.start_byte(),  # start byte
            self._total_length,  # délka
            1,  # 1. ridici pole
            0,  # 2. ridici pole
            third,  # 3. ridici pole
            fourth,  # 4. ridici pole
            )
    self._structure = packed_header
    return self._structure
```

Příchozí směr

Rámce přijímané v sériové podobě je nutné rozložit na jednotlivé byty a následně z nich znovu sestavit správné struktury. O rozložení a znovu sestavení se stará třída *Parser*, která vrací hotový objekt rámce. Kompletní výpis třídy *Parser* je ve výpisu v příloze D.1.

4.3.2 Důležité metody modulu

Modul využívá asynchronního přístupu pomocí knihovny AsyncIO. V souvislosti s použitím jazyka Python, pro tento projekt, je výhodnější a efektivnější použít asynchronní přístup s AsyncIO narozdíl od paralelního zpracování pomocí Hyper-Threadingu nebo multiprocesingu. Interpret jazyka Python k Hyper-Threadingu a multiprocesingu využívá speciální knihovny, ale jelikož na stroji běží jako jeden proces, je s přepínáním vláken nebo procesů spojena daleko vyšší režie, než by tomu bylo u jazyků nižších úrovní, např. C/C++. Taky vzhledem k neznámé časové posloupnosti komunikace mezi zařízeními je asynchronní přístup pro tuto implementaci

nejvhodnější volbou.

Aplikace neprovádí žádné náročné ani zdlouhavé výpočty s daty, ale naopak většinu svého času tráví čekáním na příchozí zprávy. Výhody použití asynchronního přístupu spočívají v tom, že se jedná o jedno vlákno a jeden proces, který využívá tzv. asynchronní smyčku. Metody, které v této smyčce mohou běžet, se nazývají *coroutines*, česky korutiny. Korutiny na procesoru neběží zároveň, jak vlákna nebo procesy u zmiňovaného Hyper-Threadingu nebo multiprocesingu, ale své vykonávání na procesoru mezi sebou střídají dle potřeby. Narazí-li na nějakou blokující operaci, jsou přesunuty do pozadí, právě do asynchronní smyčky, aby dále neblokovaly procesor ostatním korutinám. Blokujícími operacemi jsou typicky operace s I/O periferiemi nebo komunikace po síti. Jakmile korutina čekající na pozadí obdrží svá data nebo dosáhne výsledku, na který čekala, je vyvolána událost a asynchronní smyčka je schopna korutinu zvonu přesunout k dalšímu vykonání.

Knihovna AsyncIO nabízí různé metody[5], které má uvnitř implementované pomocí *coroutine*, *task* a *future*. Rozdíly mezi nimi jsou, že *coroutine* je obdoba klasické funkce u synchronního programování obsahující nějakou blokující operaci. *Task* je úloha, do které můžeme korutinu zabalit, předat ji asynchronní smyčce do pozadí a počkat na její výsledek v jiném místě kódu. *Future* je to samé jako *task* s rozdílem, že nečekáme na výsledek, který by nám korutina vrátila. *Task* se používá právě pro úlohy, kde je očekáván výsledek v návratové hodnotě. *Future* po obdržení výsledku nevrací žádnou hodnotu, ale typicky provede nějakou zvolenou funkci zpětného volání.

Těchto metod je v implementaci využito na více místech. Nyní bude popsána implementace důležitých metod modulu.

Spuštění serveru a API rozhraní modulu IEC-104

Pro zahájení naslouchání serveru je využita metoda z knihovny AsyncIO *asyncio.start_server()*. Tato metoda otvírá TCP soket na zvolené IP adrese a portu a převádí jej do stavu naslouchání. Metoda je umístěna v metodě *start()* ve třídě *ServerIEC104*. Ve třídě je zároveň spouštěn celý modul vytvořením její instance a následné zavolání metody *start()*. Třída *ServerIEC104* dále nabízí rozhraní API, díky kterému je možno celý modul IEC-104 využít ke komunikaci a výměně dat s klientem. API rozhraním jsou funkce zpětného volání nazvané *on_connect()*, *on_disconnect()*, *on_message()* a *on_send()*. Modul je možné spustit stejně jako v ukázce kódu 4.3.

[5]Dále v této práci jsou použity názvy "metoda", "funkce" nebo "korutina". Ve většině případů se vždy jedná o prostou metodu nějaké třídy. V případě použití korutiny je cílem zdůraznit, že je se jedná o asynchronní metodu uvozenou klíčovým slovem *async* před názvem metody.

Výpis 4.3: Spuštění modulu IEC-104 v jazyce Python.

```
1  import asyncio
2  import ServerIEC104
3
4
5  if __name__ == '__main__':
6      my_server = ServerIEC104()
7      my_server.register_callback_on_connect = on_connect
8      my_server.register_callback_on_message = on_message
9      my_server.register_callback_on_disconnect =
           ↪ on_disconnect
10
11     try:
12         asyncio.run(my_server.start())
13     except KeyboardInterrupt:
14         pass
15     finally:
16         pass
```

V ukázce 4.3 se server spouští vytvořením instance třídy *ServerIEC104* a dále běží asynchronně na pozadí. Registrované metody jsou spouštěny, jakmile je zavolá příslušná událost. To znamená, metoda *on_connect()* při připojení nového spojení od klienta, *on_message()*, při příchozí datech a *on_disconnect()* při ukončení spojení. Pro možnost odeslání odpovědi je v argumentu metody *on_message()* předána i funkce pro zpětné volání. V tomto výpisu nejsou znázorněny argumenty metod. Jsou okomentované přímo v implementaci.

Ukázka struktury třídy *ServerIEC104* je pak ve výpise 4.4. Jako první se provede blok kódu začínající na řádku 17. V něm je vytvořena instance třídy *ServerIEC104* a následně spuštěna metoda *asyncio.run(start())*. Tím je vytvořena asynchronní smyčka. V metodě *start()* se nachází blokující operace uvozena klíčovým slovem *await*. Tato operace je *asyncio.start_server()*, která vytvoří server s IP adresou a portem předanými v argumentu. Dále je v argumentu předána funkce pro zpětné volání *handle_client()*, která je provedena jakmile se připojí klient. Další blokující operací v metodě *main()* je *server.serve_forever()*. Ta vytvořený server spustí a čeká na připojení, dokud není zavoláno ukončení asynchronní smyčky a aplikace ukončena. Aby metoda mohla obsahovat blokující operace uvozené *await* musí být před klíčovým slovem *def* i klíčové slovo *async*. To pak značí, že se jedná o korutinu a můžeme ji také volat s *await*.

Výpis 4.4: Ukázka struktury třídy ServerIEC104 v jazyce Python.

```
import asyncio


class ServerIEC104:
    def __init__(self, name: str = "Server"):
        self.ip = config_loader.config['server']['
            ↪ ip_address']
        self.port = config_loader.config['server']['port'
            ↪ ]

    async def handle_client(self, reader, writer):
        pass

    async def start(self):
        server = await asyncio.start_server(self.
            ↪ handle_client, self.ip, self.port)
        await server.serve_forever()


if __name__ == '__main__':
    my_server = ServerIEC104()
    try:
        asyncio.run(my_server.start())
    except KeyboardInterrupt:
        pass
    finally:
        pass
```

Dále je ve výpisu kódu vidět třída *ServerIEC104* s atributy *self.ip* a *self.port* a již zmíněná korutina *handle_client()*. IP adresa a port jsou načteny z konfiguračního souboru typu JSON. Korutina *handle_client()* má v argumentu předané dva parametry, díky nímž může přijímat a odesílat data klientovi. Blíže je tato metoda popsána v kapitole 4.3.3. Pro ukázku výpis neobsahuje všechny metody ani atributy třídy.

Příjem rámců

Příjem dat od klienta zajišťuje metoda *handle_messages()* ve třídě *Session*. Výpis této metody je vložen do přílohy E.1 z důvodu její velikosti.

Metoda *handle_messages()* je spuštěna jako *future* metodou *start()* viz výpis 4.5. V této ukázce je vidět privátní atribut nazvaný *task*, i když je to *future*. Není to z důvodu, že bychom čekali návratovou hodnotu, ale z důvodu odchytnutí výjímek vzniklých uvnitř *future* a možného ukončení pomocí *task.cancel()*.

Výpis 4.5: Spuštění metody pro příjem rámce pomocí v jazyce Python.

```
1    async def start(self) -> None:
2        self.__task = asyncio.ensure_future(self.
           ↪ handle_messages())
```

Metoda *handle_messages()* je spuštěna jako *future*, protože nevrací žádnou návratovou hodnotu. Není zde klíčové slovo *return*, viz výpis kódu v příloze E.1. Metoda po spuštění běží ve smyčce *while()* dokud není nastaven příznak ukončení asynchronních úloh nebo přijetí příznaku *eof()*. Nastavení příznaku ukončení je provedeno v aktualizaci stavu spojení, jakmile je porušena některá podmínka komunikace, viz stavový diagram 4.5. Funkce *eof()* signalizuje ukončení TCP spojení. Čtení dat ze síťového spojení je zde realizováno asynchronní metodou z knihovny AsyncIO *asyncio.wait_for()* na řádku 6. Ta umožňuje čtení zadaného počtu bytů s omezeným časovým intervalem, po jehož vypršení nastane výjímka. V argumentu je předán interval *t1*, jenž má za následek aktivní ukončení spojení podle kapitoly 1.2.2. V metodě *read()* jsou čteny 2B, protože dopředu není jasné zda přijde rámec s pevnou nebo proměnnou délkou, viz kapitola 1.1.2. Teprve po přečtení 2. bytu, který nese informaci o celkové délce rámce APDU, je možné přečíst zbytek dat z bufferu spojení, na řádku 15. Na řádku 20 následuje získání objektu na příchozí rámec jeho deserializací po jednotlivých bitech v hlavičce APCI, viz kapitola 1.1.2. Dále na řádku 23 je do logu vložena informace o přijetí nového rámce. Na řádcích 25 - 28 proběhnou aktualizace časovačů jejich opětovným spuštěním. Časovače jsou realizovány instancemi třídy *Timer* vytvořených v konstruktoru *Session*. Po vypršení některého z intervalů je volána příslušná funkce pro zpětné volání, ve které je vykonána akce vždy odpovídajícímu časovému intervalu, viz kapitola 4.3.3. Tyto funkce pro zpětné volání jsou umístěny ve třídě *ClientManager*. Další funkcí zpětného volání, ze třídy *ClientManager*, je samotné zpracování přijatého rámce na řádku 31. V argumentu se předává instance daného spojení a samotný rámec s daty. Instance spojení je v argumentu předána, aby bylo možné určit, ze kterého spojení rámec pochází. Na řádcích 34-38 jsou resetovány lokální proměnné, protože metoda *handle_messages()* běží ve smyčce *while*.

4.3.3 Funkce zpětného volání v modulu

V této kapitole jsou popsány metody sloužící ke zpětnému volání označující se jako callback funkce. V kódu se typicky značí prefixem "on_", což ale není pravidlem. Funkce patří mezi základy pro běh asynchronní aplikace.

Přijetí nového spojení od klienta

Pro obsluhu nově příchozího spojení slouží metoda *handle_client()*. Tato metoda se nachází ve třídě *ServerIEC104*, jak je vidět ve výpisu 4.4. Kompletní výpis metody je v příloze F.1. Z TCP spojení je získána IP adresa a port. Podle IP adresy klienta je zjištěno, zda už s ním nějaké spojení existuje. Pokud ne, je vytvořena instance *ClientManager*, v opačném případě jen přiřazena reference již existující instance. Tím je zajištěno, že pro jednoho klienta je vytvořena pouze jedna instance *ClientManager*, jak je znázorněno ve schématu 4.2. Při vytvoření instance *ClientManager* jsou v argumentu předávány funkce zpětného volání pro komunikaci MQTT Brokerem a pro API, viz kapitola 4.3.2. Na řádku 41 je přiřazení funkce zpětného volání pro zpracování přijatých dat. Řádky 42-46 jsou přiřazení funkcí zpětného volání pro zpracování jednotlivých časových prodlev. Dále na řádku 51 je metodou uvnitř třídy *ClientManager* vytvořena instance třídy *Session*. Na řádku 63 pak spuštění metody *handle_messages()* pomocí *create_task()* pro příjem dat. Pokud má klient napojené funkce zpětného volání pro API, zde na řádku 67 je mu volána metoda *on_connect()* s informacemi o spojení v argumentu. V případě vzniklé výjímky v této metodě je volána metoda *on_connect()* s informací, že nastala chyba a nakonec zapsání chyby do logu na řádcích 72-74.

Zpracování přijatých dat

Metoda, která je volána jako funkce zpětného volání pro zpracování dat, po jejich přijetí v metodě *handle_messeges()*, se nazývá *on_message_recv_or_timeout()*. Umístěna je ve třídě *ClientManager*, výpis kódu v 4.6. Ve skutečnosti se ale v této metodě žádná data nezpracovávají. Ve skutečnosti je tato metoda pouze rozcestníkem pro akce, které tuto funkci volají. Těmito akcemi jsou přijetí rámce od klienta nebo uplynutí některé z časových prodlev, viz kapitola 1.2.2.

Metoda pro zpracování dat se nazývá *handle_apdu()*, také ve třídě *ClientManager*. Je ale příliš dlouhé zde vkládat kompletní výpis této metody, proto bude popsána zjednodušenými vývojovými diagramy. Popsáno dále v práci.

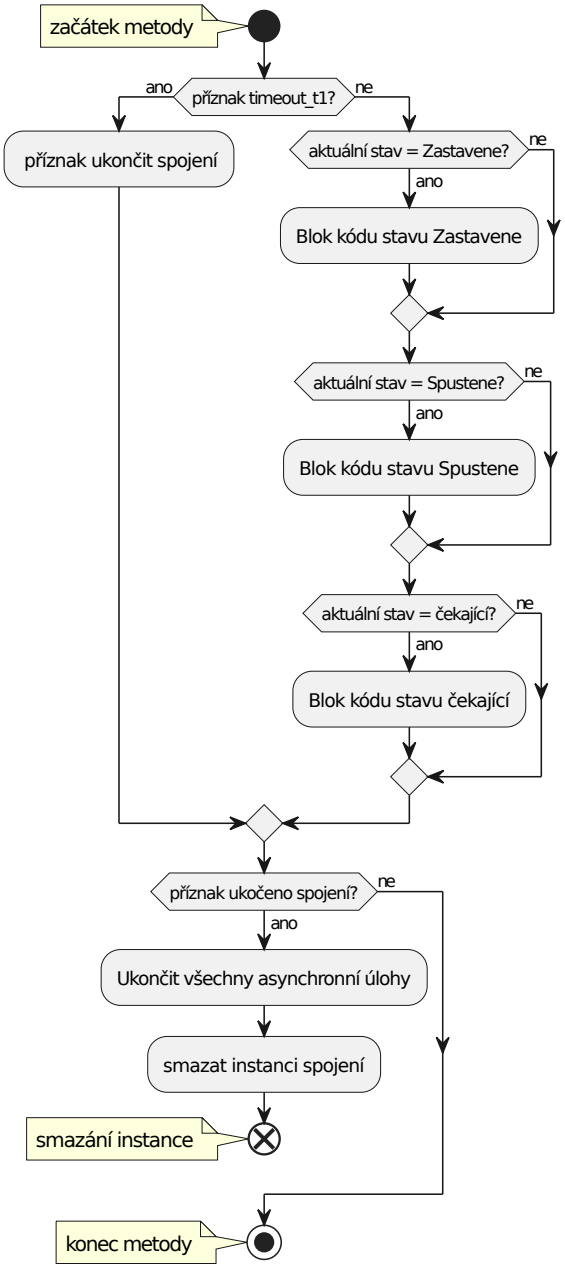
Výpis 4.6: Metoda rozcestníku při příchodu rámce nebo uplynutí časové prodlevy v jazyce Python.

```
async def on_message_recv_or_timeout(self,
    session: Session, apdu: Frame = None) -> None:

    # pokud metodu spustil přijatý ramec, zpracuj ho,
      ↪ pokud ne tak to byl timeout
    if apdu is not None:

        # vložení rámce do vyrovnávací paměti k potvrzen
          ↪ í
        if isinstance(apdu, IFormat):
            self.__recv_buffer.add_frame(apdu.ssn, apdu)
        # jedná-li se o řízenou stanici, jinak řídicí
          ↪ stanice
        if self.__whoami == 'server':
            # zpracování dat
            await self.handle_apdu(session, apdu)
            # aktualizace stavu spojení
            await self.update_state_machine_server(
              ↪ session, apdu)
        else:
            await self.handle_apdu(session, apdu)
            await self.update_state_machine_client(
              ↪ session, apdu)

    else:
        if self.__whoami == 'server':
            await self.update_state_machine_server(
              ↪ session)
        else:
            await self.update_state_machine_client(
              ↪ session)
```

V tomto výpisu 4.6 si lze povšimnout rozdělení mezi kódem pro řízenou stanici (server) a pro řídicí stanici (klient), pomocí proměnné *__whoami*, důvod je popsaný v kapitole 4.4. Metoda pro zpracování rámce *handle_apdu()* je univerzální pro oba typy stanic, avšak pro aktualizaci stavu jejich spojení je nutno je rozdělit. Důvodem je odlišný stavový diagram pro řídicí stanici, viz stavové přechodové

diagramy v [3]. Tyto aktualizace stavu spojení pro konkrétní typ stanice jsou v metodách *update_state_machine_client()* a *update_state_machine_server()*.

Zpracování dat v metodě *handle_apdu()* je závislé na aktuálním stavu spojení, viz stavový diagram na obrázku 4.5. Pro rozhodování, co s přijatým rámcem v každém z možných stavů je zobrazeno ve vývojových diagramech na obrázcích v přílohách G.1, G.2, G.3 a G.4. Diagram znázorňující stav spojení "spuštěné", je z důvodu velikosti rozdělen na dvě části mezi obrázky v přílohách G.2 a G.3. Pro znázornění napojení na sebe jsou vyznačeny body A a B. Všechny tyto stavy jsou pak v sérii za sebou, jak je znázorněno v diagramu 4.6.



Obr. 4.6: Vývojový diagram všech stavů spojení

Zpracování časových prodlev

V této kapitole je zobrazen výpis kódu s metodami pro obsluhu funkcí zpětného volání jednotlivých časových prodlev, ve výpise 4.7. Jednotlivé časové prodlevy a jejich akce jsou popsané v kapitole 1.2.2. Akce pro *t0* je aktivní ukončení spojení. Na řádku 6 je zavolána metoda pro okamžité ukončení spojení a smazání instance *Session*. Časová prodleva *t1* má za následek také ukončení spojení, ovšem zde se jen nastaví příznak, že *timeout_t1* nastal, zapíše se do logu a spustí aktualizaci stavu spojení. Časová prodleva *t2* slouží k potvrzení všech nepotvrzených rámců s daty. Na řádku 18 se vygeneruje nový S-formát rámce a předá se metodě *send_frame()* k odeslání. Na řádku 20 je vytvořená úloha pro odeslání rámce vložena do listu, pro případ, že nastane ukončení. Pro bezpečné ukončení a smazání instancí je nejprve potřeba ukončit všechny úlohy v asynchronní smyčce. Takto se projde list s asynchronními úlohami pomocí cyklu *for* a ukončí se pomocí *task.calcel()*. Časová prodleva *t3* způsobí odeslání testovacího rámce *TESTFR act*. Stejně jako předešlé metody je vytvořen rámec a odeslán.

Výpis 4.7: Metody obsluhy uplynutí časových prodlev t0,t1,t2,t3 v jazyce Python.

```
1  async def handle_timeout_t0(self,
2          session: Session = None) -> None:
3      logging.debug(f"Timer t0 timed_out - {session}")
4      logging.debug(f'Client {session.ip}:{session.port}
         ↪ timed out and disconnected')
5      session.flag_session = 'ACTIVE_TERMINATION'
6      session.delete_self()
7
8  async def handle_timeout_t1(self,
9          session: Session = None) -> None:
10     logging.debug(f"Timer t1 timed_out - {session}")
11     session.flag_timeout_t1 = 1
12     asyncio.ensure_future(self.on_message_recv_or_timeout
         ↪ ( session ))
13
14 async def handle_timeout_t2(self,
15         session: Session = None) -> None:
16     logging.debug(f"Timer t2 timed_out - {session}")
17     if not self.__recv_buffer.is_empty():
18         new_frame = await self.generate_s_frame(session)
19         task = asyncio.ensure_future(self.send_frame(
             ↪ session, new_frame))
```

```
20          self.__tasks.append(task)
21
22  async def handle_timeout_t3(self,
23          session: Session = None) -> None:
24      logging.debug(f"Timer t3 timed_out - {session}")
25      new_frame = self.generate_testdt_act()
26      task = asyncio.ensure_future(self.send_frame(session,
            ↪  new_frame))
27      self.__tasks.append(task)
```

Odeslání dat

V této části je zobrazen výpis 4.8 metody *send_frame()* pro odesílání rámců ve třídě *Session*.

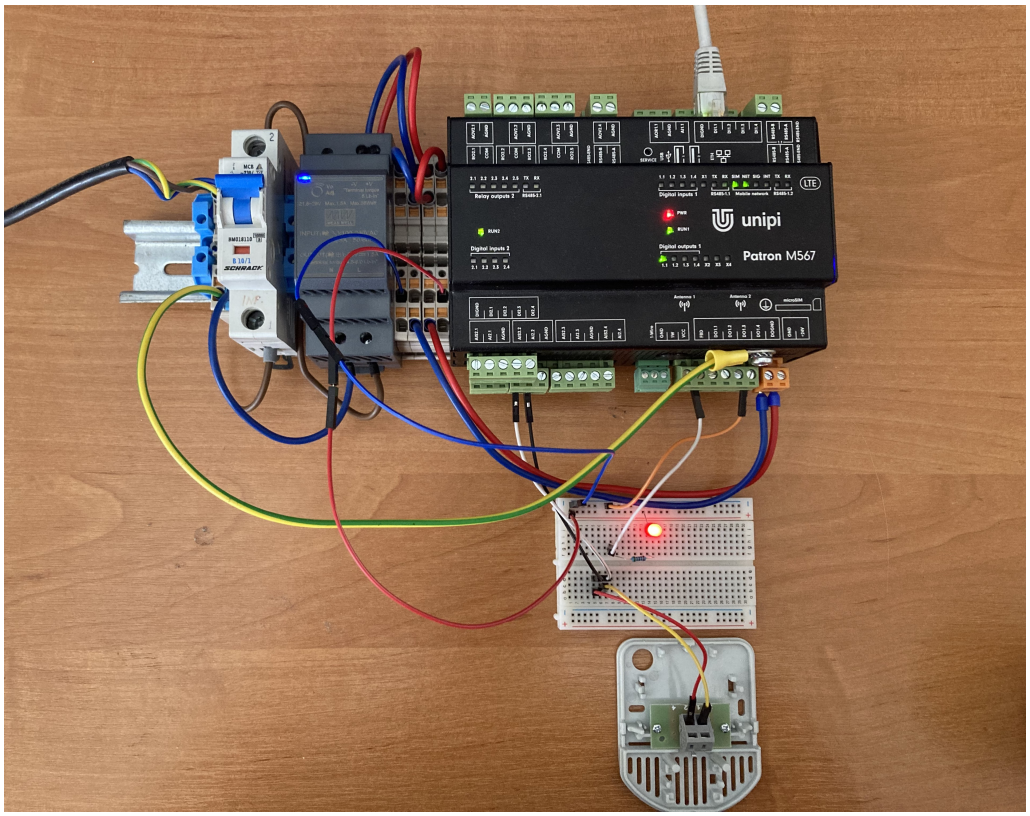
Výpis 4.8: Metoda odeslání rámce v jazyce Python.

```
1  async def send_frame(self, frame: Frame = None) -> None:
2      if not self.__flag_stop_tasks:
3
4          try:
5              if frame is not None:
6                  logging.debug(f"{time.strftime('%X')}-
                       ↪ Send"
7                                f" to {self.ip}:{self.port}"
8                                f" - frame: {frame}")
9                  # serializace rámce pro odeslání po
                       ↪ bytech
10                 self.__writer.write(frame.serialize())
11                 await self.__writer.drain()
12
13             # vyrovnávací paměť pro nepotvrzené rámce
14             if isinstance(frame, IFormat):
15                 self.__send_buffer.add_frame(frame.ssn,
                       ↪ frame)
16
17         except Exception as e:
18             logging.error(f"Exception {e}")
```

Předtím, než jsou data odeslána, je zkontrolován příznak ukončení asynchronních úloh. Je to z důvodu, že v momentě, kdy je spojení již ukončeno, by mohla v asynchronní smyčce zůstat jedna "zombie" úloha pro odeslání dat. Po zápisu informace do logu a odeslání dat jsou data ještě uloženy do vyrovnávací paměti pro následné potvrzení od klienta, že mu přišli data v pořádku.

4.4 Testování a validace

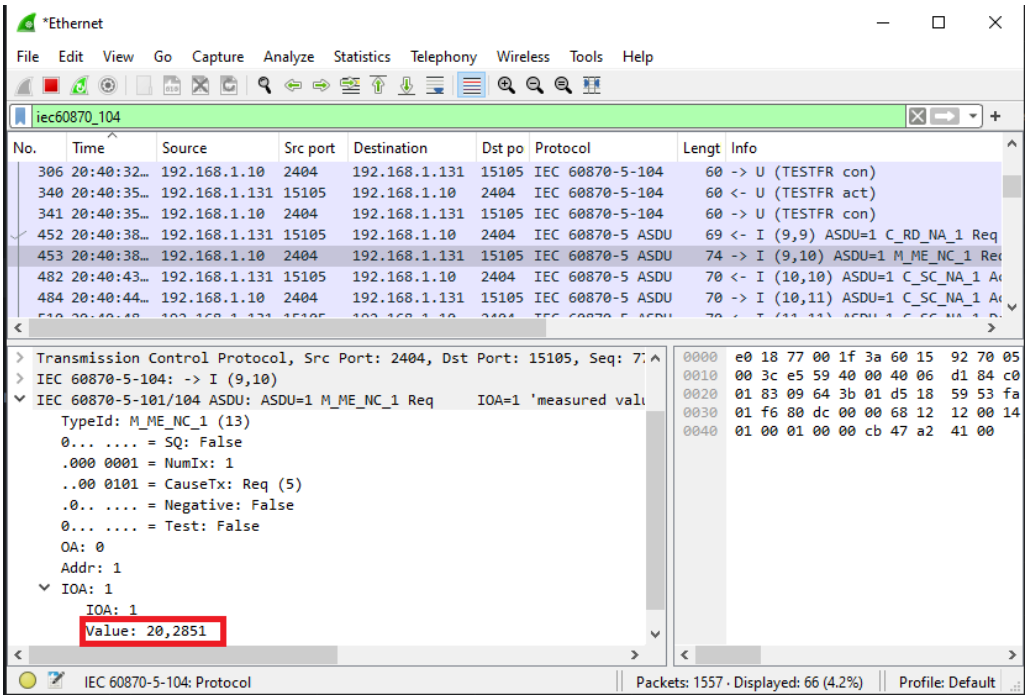
Jelikož během této práce nebyl přístup k reálné aplikaci komunikující protokolem IEC-104, a ani nebyly nelezeny programy pro nekomerční užití, byla zároveň s touto implementací řízené stanice vyvíjena i implementace stanice řídicí. Obě tyto implementace využívají většinu stejných metod. Některé metody bylo však nutné rozlišit pro specifické chování každé stanice. Toho si lze všimnou v některých ukázkách kódu nebo v samotné implementaci v příloze, občas mají metody "*server*" nebo naopak "*client*" ve svém názvu. Pro rozlišení, o kterou stanici se jedná slouží privátní atribut *whoami*, který má v sobě uloženou řetězcovou hodnotu "*server*" nebo naopak "*client*".



Obr. 4.7: Reálné zapojení přípravku

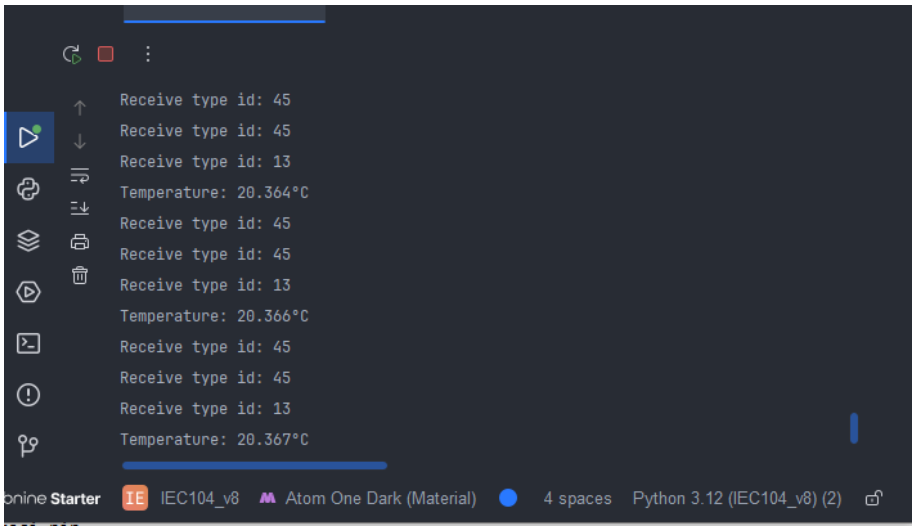
Aplikace byla implementována a ověřována v zapojení podle schématu na obrázku 3.2, kde na jednotce unipi Patron je funkční modul IEC-104, modul pro zpracování, viz obrázek 3.4 a běžící kontejnery v nástroji Docker pro ukládání do databáze, viz obrázek 3.3. Modul klientské stanice byl spouštěn na PC s OS Windows 10. Samotný klient byl dále rozšířen o modul zpracování odpovědí od serveru, aby byl schopen vypsat do konzole změřenou hodnotu teploty z čidla PT1000 připojenému k jednotce. Čidlo PT1000 bylo zvoleno pro její lineární charakteristiku a jednoduchý přepočet měřené hodnoty odporu na teplotu. Reálné zapojení přípravku je zobrazeno na obrázku 4.7. Na obrázku je možné vidět zapojené teplotní čidlo, LED diodu a samotnou jednotku unipi Patron. Teplotní čidlo je přímo připojeno na vstup AI2_02, který je nastaven v režimu měření odporu. LED dioda na výstup DO1_01. Dioda je napájena ze zdroje 24 V a k ní je do série zapojen odpor 5 kΩ.

Klient má staticky naprogramované zasílání dvou testovacích rámců TESTFR act a funkce pro čtení teploty, sepnout LED a vypnout LED. Tato posloupnost se stále dokola opakuje. Na obrázku 4.8 je zobrazena zachycená komunikace mezi PC s IP adresou 192.168.1.131, na které běží klient a jednotka s IP adresou 192.168.1.10, na níž běží server.



Obr. 4.8: Zachycená komunikace IEC104 pomocí Wireshark

Na obrázku 4.9 je zachycen výpis do konzole na klientské stanici, kde je vypisováno jakého typu přišla odpoveď od serveru. Odpověď typu 45 je potvrzení sepnutí nebo vypnutí diody a 13 je měřená hodnoty z teplotního čidla. Tuto hodnotu lze vidět i na obrázku 4.8 zachycenou programem Wireshark.



Obr. 4.9: Výpis konzole na klientské stanici

Pro ukázku fungující databáze je na obrázku 4.10 zachycen graf z nástroje Grafana, na kterém jsou zobrazeny čísla typů příchozích požadavků v čase. Jak bylo zmíněno, klient zasílá periodicky stejnou posloupnost, ve které se střídají pouze typy s číslem 45 a 102. Proto jsou v grafu vidět body pouze na těchto hodnotách.

Obr. 4.10: Závislost příchozích požadavků v čase

Závěr

Cílem této bakalářské práce bylo navrhnout, implementovat a ověřit modul, který umožňuje integraci protokolu IEC-104 do existujícího PLC kontroléru řady UniPi Patron.

Nejprve bylo nutné provést základní analýzu a seznámení s tímto pro mě dosud neznámým protokolem. První část práce se proto zabývá teoretickými poznatky tohoto protokolu. Popsány jsou struktura přenášených rámců a typy dat. Dále jsou zmíněny stavy a pravidla komunikace v protokolu IEC-104, díky nímž zajišťuje bezpečné ovládání vzdálených procesů.

V druhé části práce jsem blíže přiblížil využití protokolu společně se systémy SCADA a odvětvím energetiky, ve kterém má tento protokol největší zastoupení. Dále jsem popsal jednotku Unipi Patron, na které bylo později provedeno ověření funkčnosti implementace a navrhl scénář podobný řízení v elektrické stanici. V mém případě byly pro ověření funkčnosti implementace serveru IEC-104 simulované ovládané prvky, a to spínání LED diody a čtení analogové hodnoty z teplotního čidla.

V třetí části jsem popsal samotnou implementaci serveru IEC-104 v programovacím jazyce Python. Popsal jsem navrženou architekturu celého modulu, včetně komponent a jejich funkcí. Dále jsem popsal implementační detaily, metody a třídy kritické pro běh celé aplikace. Zaměřil jsem se na implementaci serializaci a deserializaci rámců IEC-104, řízení stavů komunikace, zpracování dat a odeslání dat do řídicího systému.

V závěru této práce jsem popsal ověření funkčnosti a způsobu testování implementace serveru IEC-104. Testoval jsem komunikaci serveru se simulovanými ovládanými prvky a ověřil tak správnost zpracování dat a odeslání dat do řídicího systému. Zachycená komunikace prokázala, že implementovaný server IEC-104 splňuje zadáním definované požadavky.

Literatura

[1] Communication Technologies, Inc. *Supervisory Control and Data Acquisition (SCADA) Systems*, Říjen 2004. URL: `https://scadahacker.com/library/Documents/ICS_Basics/SCADA%20Basics%20-%20NCS%20TIB%2004-1.pdf`.

[2] Webstore IEC. 60870-5:2018 ser series. [online], Prosinec 2023. URL: `https://webstore.iec.ch/publication/3755`.

[3] ČSN EN 60870-5-104 ed. 2 (334650). *Systémy a zařízení pro dálkové ovládání – Část 5-104: Přenosové protokoly – Síťový přístup pro IEC 60870-5-101 používající normalizované transportní profily*, 2006.

[4] ČSN EN 60870-5-101 ed. 2 (334650). *Systémy a zařízení pro dálkové ovládání – Část 5-101: Přenosové protokoly – Společná norma pro základní úkoly dálkového ovládání*, 2003.

[5] ELVACa a. s. Elvac a.s. Řešení pro energetiku. [online], 2023. URL: `https://www.rtu.cz/`.

[6] ELVACa a. s. *RTU7MC3-D komunikační jednotka*, Prosinec 2022. Datasheet.

[7] Unipi Technology s.r.o. [online], Prosinec 2023. URL: `https://www.unipi.technology/`.

[8] ed. Postel, J. *Transmission Control Protocol - DARPA Internet Program Protocol Specification*. USC/Information Sciences Institute, Září 1981. RFC 793.

[9] David Košut. Komunikační úlohy v rámci smart grids. dimenzování komunikačních sítí a datového úložiště včetně nákladového modelu. Master's thesis, Vysoké učení technické v Praze. Fakulta elektrotechnická, 2019.

[10] Lukáš Sobotka. Analýza protokolů pro komunikaci v energetických sítích. Master's thesis, Vysoké učení technické v Brně. Fakulta informačních technologií, 2019.

[11] Dominik Pekárek. Popis a testování komunikačních protokolů normy IEC 60870-5-103 a 60870-5-104. Master's thesis, Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2017.

[12] Petr Matoušek. Description and analysis of IEC 104 protocol. Technical report, Faculty of Information Technology Brno University of Technology Brno, Czech Republic, Prosinec 2017.

[13] MZ Automation GmbH. mz-automation/lib60870. [online], Prosinec 2022. URL: https://github.com/mz-automation/lib60870.

[14] MZ Automation GmbH. mz-automation/lib60870.NET. [online], Duben 2019. URL: https://github.com/mz-automation/lib60870.NET.

[15] FreyrSCADA. FreyrSCADA/IEC-60870-5-104. [online], Říjen 2023. URL: https://github.com/FreyrSCADA/IEC-60870-5-104.

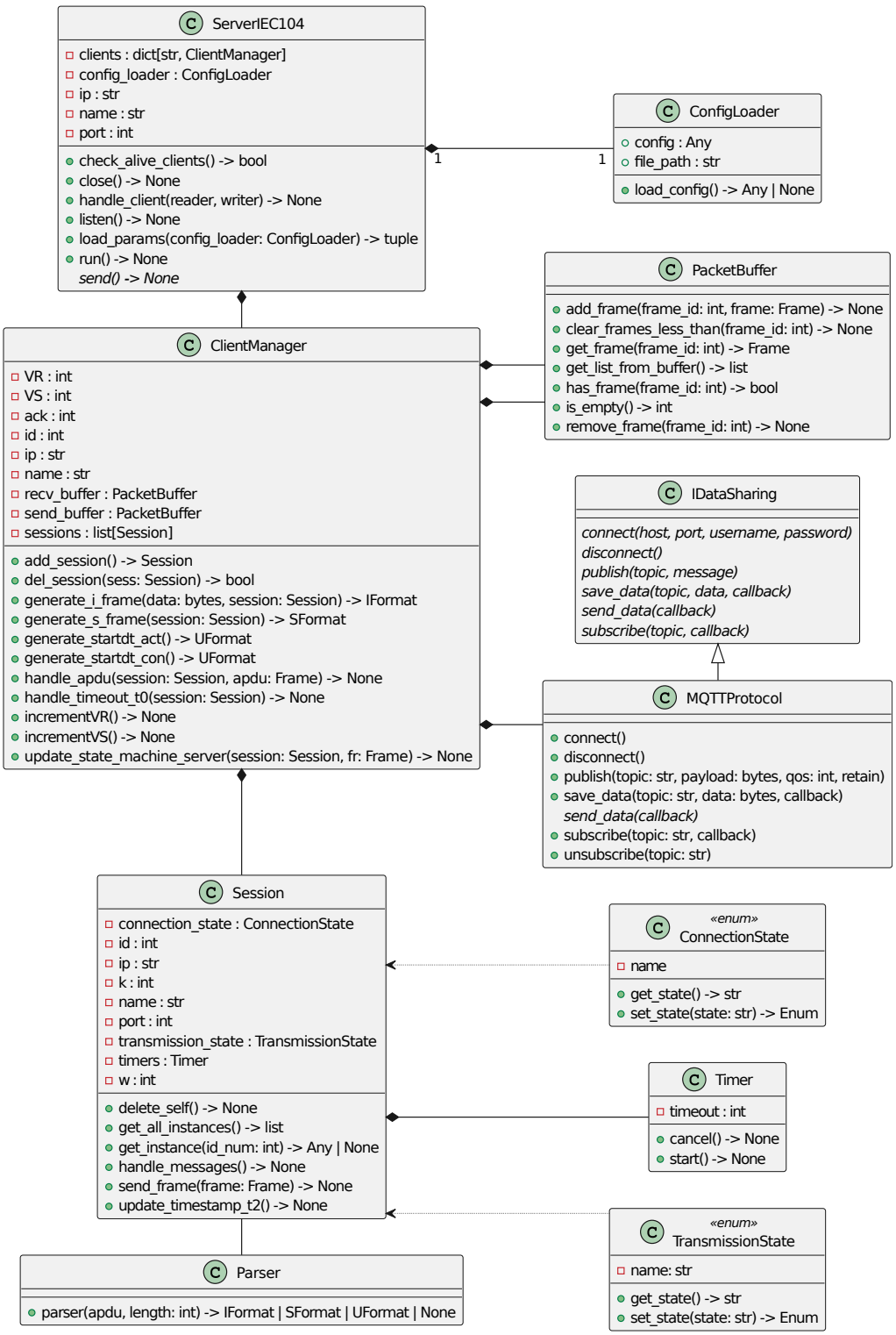
Seznam symbolů a zkratek

AI	Analogový vstup – Analog Input
API	Application Programming Interface – Aplikační programové rozhranní
AO	Analogový výstup – Analog Output
APCI	Application Protocol Control Information – Řídicí informace aplikačního protokolu
APDU	Application Protocol Data Unit – Jednotka dat aplikačního protokolu
ASDU	Application Service Data Unit – Jednotka dat aplikačního služby
ATM	Asynchronous transfer mode – Vysokorychlostní síťová architektura
BMS	Building Management System – Řídicí systém budov
CPU	Central Processing Unit – Centrální procesorová jednotka
ČSN	Česká technická norma
DB	Database System – Databázový systém
DI	Digital Input – Digitální vstup
DLMS	Device Language Message Specification – Standard komunikačního protokolu pro vzdálené odečty a správu energetických měřidel a zařízení
DNP3	Diustributed Network Protocol – Distribuovaný síťový protokol
DO	Digital Output – Digitální výstup
DSL	Domain-Specific Language – Jazyk specifický pro doménu
EN	Evropská norma
eMMC	embedded MultiMediaCard
FR	Frame Relay
GPLv3	General Public Licence version 3
GSM	Global System for Mobile Communications

HMI	Human Machine Interface – Rozhraní člověk-stroj
HW	Hardware – Technické vybavení počítače
IOA	Information Object Address – Adresa informačního objektu
I/O	Input/Output – Fyzické vstupy a výstupy
IEC	International Electrotechnical Commission – Mezinárodní elektrotechnická komise
IEC TS	International Electrotechnical Commission Technical Specification
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things – Internet věcí
IP	Internet Protocol – Internetový protokol
ISDN	Integrated Services Digital Network – Datoví síť integrovaných služeb
IT	Information Technology – Informační technologie
JSON	JavaScript Object Notation
LAN	Local Area Network – Místní síť
LED	Light-Emitting Diode
LSB	Least Significant Bit – bit slova s nejnižší váhou
LTE	Long Term Evolution
MaR	Měření a regulace
MQTT	Message Queuing Telemetry Transport
MSB	Most Significant Bit – bit slova s největší váhou
MTU	Master Terminal Unit – Hlavní koncová jednotka
OPC UA	Open Platform Communications Unified Architecture – Komunikační standard pro průmyslovou automatizaci
OS	Operation System – Operační systém
OSI	Open Systems Interconnection – Propojení otevřených systémů
PC	Personal Computer – Osobní počítač

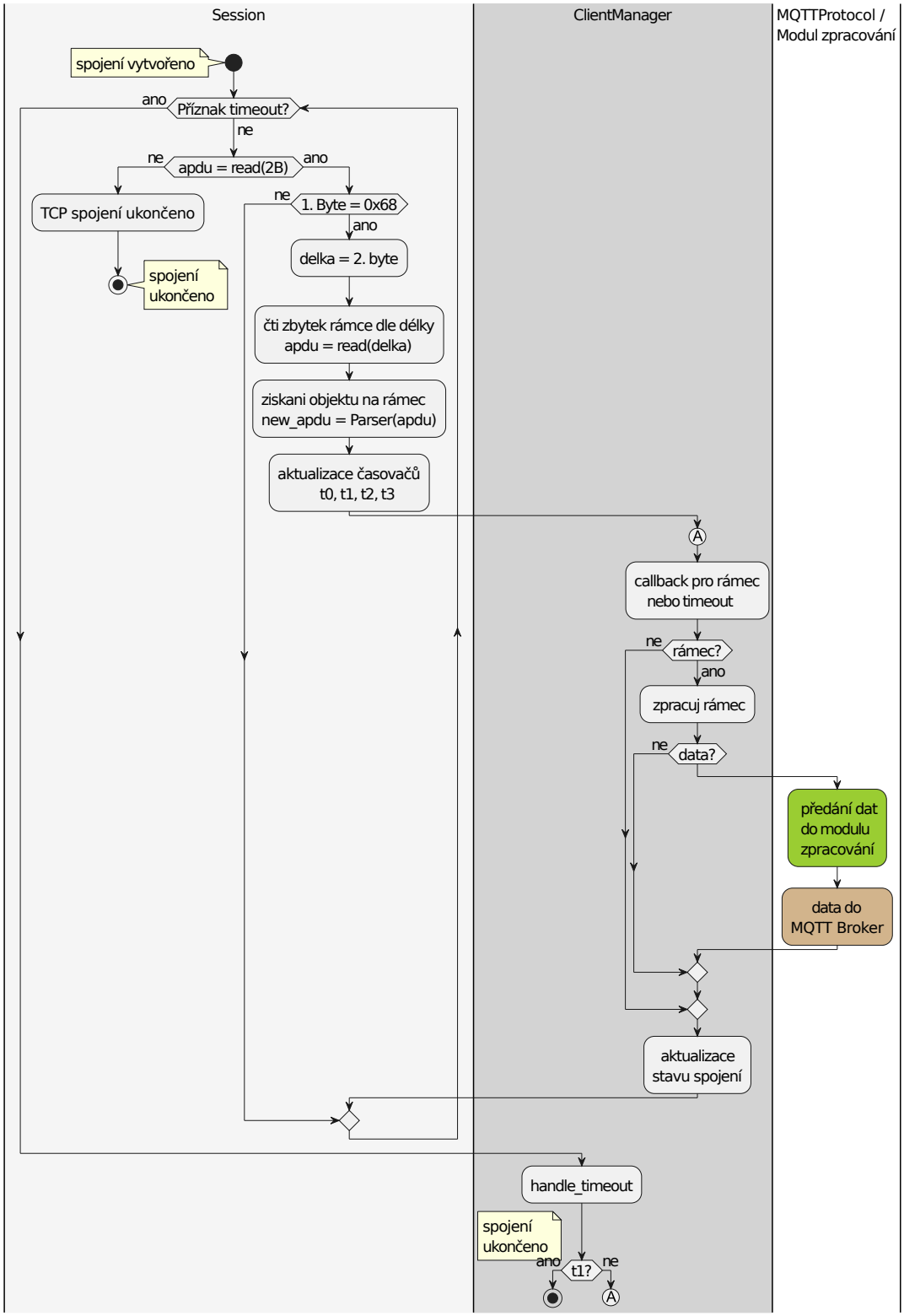
PLC	Programmable Logic Controller – Programovatelná logická jednotka
RAM	Random Access Memory – Operační paměť
REST	Representational State Transfer
RFC	Request For Comments
RO	Relay Output – Reléový výstup
RTU	Remote Terminal Unit – Vzdálená koncová jednotka
SCADA	Supervisory Control And Data Acquisition - Dispečerské řízení a sběr dat
SD	Secure Digital – ve spojení SD karta
SDK	Software Developement Kit – Softwarový vývojový nástroj
SNMP	Simple Network Management Protocol
SQL	Structure Query Language – Strukturovaný dotazovací jazyk
SW	Software – Programové vybavení počítače
TCP	Transmission Control Protocol
USB	Universal Serial Bus – Univerzální sériová sběrnice
VLAN	Virtual LAN – Virtuální místní síť
VN	Vysoké napětí
VPN	Virtual Private Network – Virtuální privátní síť

A Objektový model



Obr. A.1: Objektový model

B Datový tok



Obr. B.1: Vývojový diagram pro cestu dat modulem

C Konfiguračního souboru s mapováním I/O

Výpis C.1: Ukázka konfiguračního souboru s mapováním I/O v jazyku JSON.

```
{ "evok_conf": [
  {
    "host": "192.168.1.10",
    "port": 8080,
    "ASDU_address": 1
  }
],
  "mappings": [
    {
      "ioa": 1,
      "pin": "ai",
      "pin_id": "2_02",
      "command_method": "evok_api",
    },
    {
      "ioa": 2,
      "pin": "do",
      "pin_id": "1_01",
      "command_method": "evok_api",
    },
    {
      "ioa": 3,
      "pin": "ao",
      "pin_id": "1_01",
      "command_method": "gpio",
    },
    {
      "ioa": 4,
      "pin": "di",
      "pin_id": "1_01",
      "command_method": "gpio",
    }
  ]
}
```

D Třída *Parser*

Výpis D.1: Třída Parser v jazyce Python.

```
class Parser:

    @classmethod
    def parser(cls, apdu, length: int) -> IFormat |
        ↪ SFormat | UFormat | None:

        # rozseka data po bytech
        unpacked_apdu = struct.unpack(f"{'B' * length}",
            ↪ apdu)

        # zjisti o který formát se jedná
        frame_format = Parser.what_format(unpacked_apdu)

        # poskládání jednotlivých hodnot po bitech
        if frame_format == "I":
            # Parse I-Format
            ssn = (unpacked_apdu[1] << 7) + (
                ↪ unpacked_apdu[0] >> 1)
            rsn = (unpacked_apdu[3] << 7) + (
                ↪ unpacked_apdu[2] >> 1)
            data = struct.unpack(f"{'B' * length}", apdu)
            asdu_data = data[APCI.ACPI_HEADER_LENGTH:]
            new_instance = IFormat(bytes(asdu_data), ssn,
                ↪  rsn, direction='IN')
            return new_instance

        # poskládání jednotlivých hodnot po bitech
        elif frame_format == "S":
            # Parse S-Format
            rsn = (unpacked_apdu[3] << 7) + (
                ↪ unpacked_apdu[2] >> 1)
            new_instance = SFormat(rsn, direction='IN')
            return new_instance

        # u U-formátu se zjišťuje typ rámce
```

```
30          elif frame_format == "U":
31              # Parse U-Format
32              first_byte = unpacked_apdu[0]
33              new_instance = UFormat(direction='IN')
34
35              # STARTDT ACT
36              if first_byte == APCI.STARTDT_ACT:
37                  new_instance.type = APCI.STARTDT_ACT
38                  return new_instance
39
40              # STOPDT ACT
41              elif first_byte == APCI.STOPDT_ACT:
42                  new_instance.type = APCI.STOPDT_ACT
43                  return new_instance
44
45              # TESTDT ACT
46              elif first_byte == APCI.TESTFR_ACT:
47                  new_instance.type = APCI.TESTFR_ACT
48                  return new_instance
49
50              elif first_byte == APCI.STARTDT_CON:
51                  new_instance.type = APCI.STARTDT_CON
52                  return new_instance
53
54              # STOPDT CON
55              elif first_byte == APCI.STOPDT_CON:
56                  new_instance.type = APCI.STOPDT_CON
57                  return new_instance
58
59              # TESTDT CON
60              elif first_byte == APCI.TESTFR_CON:
61                  new_instance.type = APCI.TESTFR_CON
62                  return new_instance
63
64              else:
65                  # Toto by nemělo nikdy nastat
66                  raise Exception(f"Unexpected␣U-Format␣
                        ↪ received:␣{first_byte}")
67
```

```
68          else:
69              # Neznámý formát - nemělo by nikdy nastat
70              raise Exception("Received unknown APDU format
                    ↪ ")
71
72      # vrátí typ formátu podle informace v hlavičce
73      @classmethod
74      def what_format(cls, first_byte: tuple) -> str | None
            ↪ :
75          first_byte = first_byte[0]
76
77          if not (first_byte & 1):
78              # I-Format
79              return "I"
80          elif (first_byte & 3) == 1:
81              # S-Format
82              return "S"
83          elif (first_byte & 3) == 3:
84              # U-Format
85              return "U"
86          else:
87              # Unknown format
88              return None
```

E Metoda *handle_messages()*

Výpis E.1: Metoda pro příjem dat od klienta v jazyce Python.

```
async def handle_messages(self) -> None:
    # smyčka kontroluje příznak ukončení spojení
    while not self.__flag_stop_tasks:
        try:
            # příjem dat
            header = await asyncio.wait_for(self.__reader
                ↪ .read(2), timeout=self.__timeout_t1)

            # byli přijaty nějaké data
            if header:
                start_byte, frame_length = header

                # data patří k IEC-104
                if start_byte == Frame.start_byte():
                    # načti zbytek rámce
                    apdu = await self.__reader.read(
                        ↪ frame_length)

                    # kontrola délky dle pole délka
                    if len(apdu) == frame_length:
                        # získání deserializace
                        new_apdu = Parser.parser(apdu,
                            ↪ frame_length)

                        # zalogování
                        logging.debug(f"{datetime.now().
                            ↪ strftime("%H:%M:%S:%f")}-
                            ↪ Received␣from␣{self.ip}:{
                            ↪ self.port}␣-␣frame:␣{
                            ↪ new_apdu}")
                        # aktualizace časovačů
                        self.__timer_t0.start()
                        self.__timer_t1.start()
                        self.__timer_t2.start()
                        self.__timer_t3.start()
```

```
29
30                          # spuštění callbacku pro zpracov
                               ↪ ání dat
31                          asyncio.ensure_future(self.
                               ↪ __callback_on_message_recv(
                               ↪ self, new_apdu))
32
33                          # reset lokalních proměnných
34                          header = None
35                          start_byte = None
36                          apdu = None
37                          new_apdu = None
38                          frame_length = None
39
40              # ukončení TCP spojení
41              if self.__reader.at_eof():
42                  self.flag_session = 'ACTIVE_TERMINATION'
43                  asyncio.ensure_future(self.
                       ↪ __callback_on_message_recv(self))
44                  break
45
46          # protistrana neposílá po dobu t1 žádná data,
               ↪ ukončit spojneí
47          except asyncio.TimeoutError:
48                  # příznak ukončení
49                  self.flag_session = 'ACTIVE_TERMINATION'
50                  # aktualizace stavů spojení
51                  asyncio.ensure_future(self.
                       ↪ __callback_on_message_recv(self))
```

F Metoda *handle_client()*

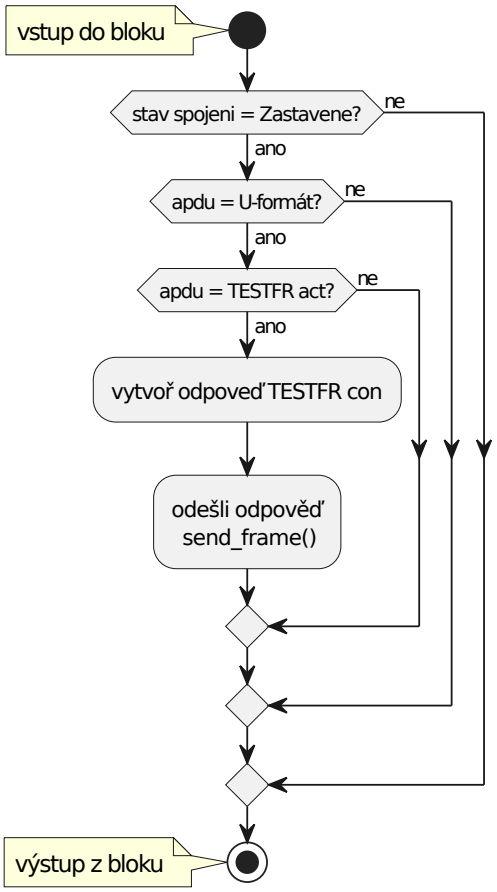
Výpis F.1: Metoda obsluhu nově příchozího spojení od klienta v jazyce Python.

```
1    async def handle_client(self, reader,
2            writer) -> None:
3
4        # získá IP adresu a port z TCP spojení
5        client_addr, client_port = writer.get_extra_info(
           ↪ 'peername')
6
7        # callback pro smazání instance ClientManagera
           ↪ pokud již není aktivní spojení
8        callback_for_delete = self.delete_dead_clients
9
10       # vytvoří instance ClientManager, pokud s
           ↪ klientem již neexistuje spojení
11       if client_addr not in self.clients:
12           client_manager_instance = ClientManager(
13               client_addr,
14               port=self.port,
15               server_name=self.name,
16               mqtt_broker_ip=self.mqtt_broker_ip,
17               mqtt_broker_port=self.mqtt_broker_port,
18               mqtt_topic=self.mqtt_topic,
19               mqtt_version=self.mqtt_version,
20               mqtt_transport=self.mqtt_transport,
21               mqtt_username=self.mqtt_username,
22               mqtt_password=self.mqtt_password,
23               mqtt_qos=self.mqtt_qos,
24               callback_for_delete=callback_for_delete,
25               callback_on_message=self.
                   ↪ __callback_on_message,
26               callback_on_disconnect=self.
                   ↪ __callback_on_disconnect,
27               flag_callbacks=self.__flag_set_callbacks,
28               callback_only_for_client=None,
29               whoami='server')
30
```

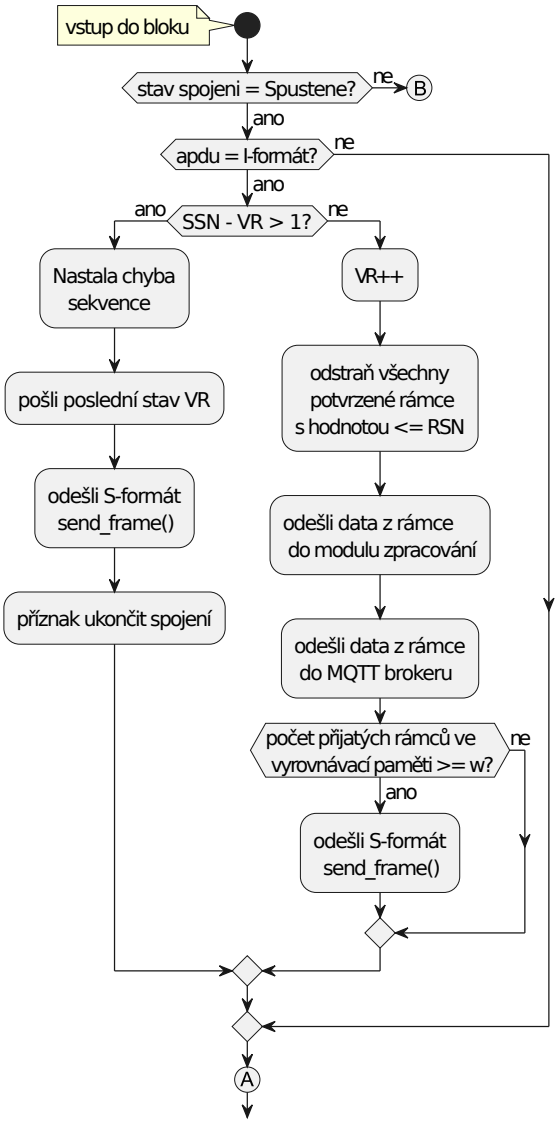
```
31              # přiřazení instance do slovníku, klíč je IP
32              self.clients[client_addr] =
                    ↪ client_manager_instance
33
34              # logování
35              logging.info(f"Established␣connection␣with␣
                    ↪ new␣client:␣{client_addr}")
36
37          # získání reference na instanci
38          client_manager_instance: ClientManager = self.
                ↪ clients[client_addr]
39
40          # callback funkce pro předání v argumentu pro
                ↪ Session
41          callback_on_message_recv =
                ↪ client_manager_instance.
                ↪ on_message_recv_or_timeout
42          callback_timeouts_tuple: tuple = (
43              client_manager_instance.handle_timeout_t0,
44              client_manager_instance.handle_timeout_t1,
45              client_manager_instance.handle_timeout_t2,
46              client_manager_instance.handle_timeout_t3,
47          )
48
49
50          # vytvoření instance Session uvnitř
                ↪ ClientManagera
51          session: Session = client_manager_instance.
                ↪ add_session(
52                  client_addr,
53                  client_port,
54                  reader,
55                  writer,
56                  self.config_parameters,
57                  callback_on_message_recv,
58                  callback_timeouts_tuple,
59                  whoami='server')
60
61          try:
```

```
62              # spuštění asynchronní smyčky v
                    ↪ handle_messages()
63              task = asyncio.create_task(session.start())
64
65              # pokud má klient napojené callback funkce
66              if self.__flag_set_callbacks:
67                  self.__callback_on_connect(client_addr,
                        ↪ client_port, rc=0)
68              # čekání na dokončení (nikdy)
69              await task
70
71          except Exception as e:
72              if self.__flag_set_callbacks:
73                  self.__callback_on_connect("", 0, rc=1)
74              logging.error(f"Exception␣with␣start␣
                    ↪ handle_messages():␣{e}")
```

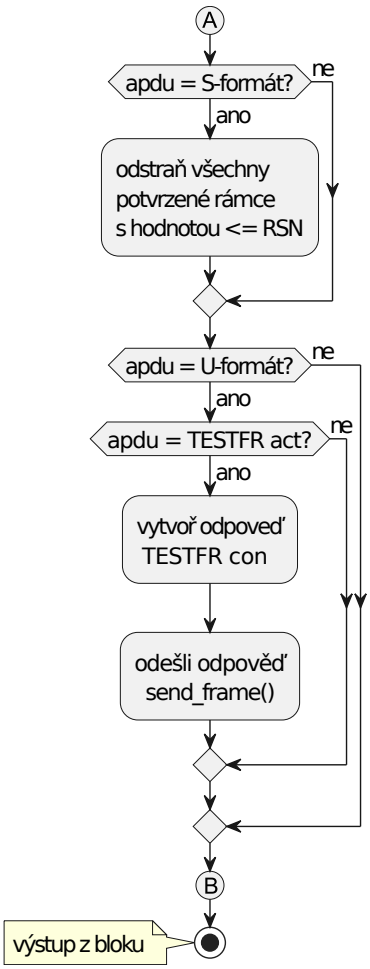
G Metoda *handle_apdu()*



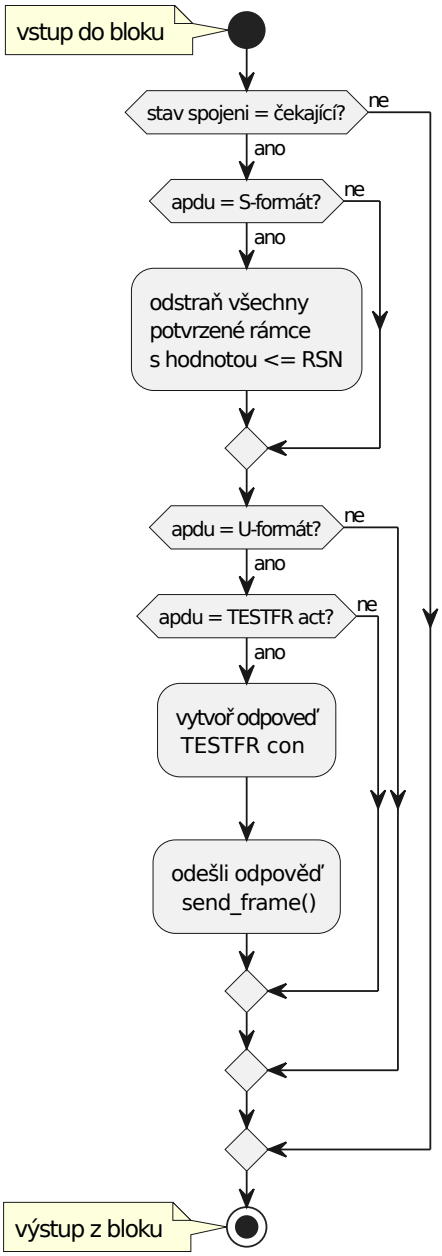
Obr. G.1: Vývojový diagram stav 1 - zastavené spojení



Obr. G.2: Vývojový diagram stav 2 - spuštěné spojení - 1. část



Obr. G.3: Vývojový diagram stav 2 - spuštěné spojení - 2. část



Obr. G.4: Vývojový diagram stav 3 - čekající spojení