
# ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Katedra řídící techniky


# Programové vybavení datového koncentrátoru AMM sítě

Diplomová práce

| | |
|---|---|
| Studijní program: | Otevřená informatika |
| Studijní obor: | Počítačové inženýrství |
| Vedoucí práce: | doc. Ing. Jiří Novák, Ph.D. |


**Bc. Lukáš Krejčí**


**Praha 2014**

# Čestné prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací

V Praze dne 12.5.2014 .....

.....

Podpis autora práce

Chtěl bych poděkovat vedoucímu práce panu doc. Ing. Jiřímu Novákovi, Ph.D. za příležitost pracovat na tématu této práce a za rady a vedení, bez kterých by tato práce nevznikla. Dále děkuji všem, kteří mě na FEL ČVUT vyučovali a umožnili mi tak získání vědomostí potřebných pro realizaci této práce. Také bych chtěl poděkovat svým blízkým za podporu.

# Abstrakt

Tato práce se zabývá návrhem a částečnou implementací softwarového vybavení datového koncentrátoru AMM sítě poskytující datovému koncentrátoru funkčnost programovatelného autonomního chování, stejně jako konfigurovatelnou podporu sítí měřidel a sítí datových ústředen. Bylo třeba zajistit splnění požadavků na využití vestavěné verzi operačního systému Linux, využití interního databázového systému a webového serveru a podporu standardu DLMS/COSEM.

V této práci bylo navrženo hierarchicky strukturované softwarové vybavení a principy interních komunikačních spojení. Dle tohoto návrhu byla implementována komunikační knihovna a součást softwarového vybavení zodpovědná za autonomní chování koncentrátoru.

# Abstract

The Aim of this diploma thesis was to design and partially implement software facility of AMM network data conentrator providing functionality of programmable autonomous behaviour, as well as configurable support of metering and data storage networks to the data concentrator. It was necessary to fullfill requested usage of embedded version of Linux operating system, usage of internal database system and web server and support of DLMS/COSEM standard.

In this work, hierarchically structured software facility and principles of internal communication were designed. According to this design, communication library and part of software facility responsible for autonomous behaviour were implemented.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

# ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Lukáš Krejčí**

Studijní program: Otevřená informatika (magisterský)
Obor: Počítačové inženýrství

Název tématu: **Programové vybavení datového koncentrátoru AMM sítě**

Pokyny pro vypracování:

Navrhněte strukturu programového vybavení koncentrátoru s ohledem na následující požadavky:
- autonomní programovatelné funkce
- podpora zobecněných komunikačních kanálů směrem k měřičům energie
- podpora standardu DLMS/COSEM
- ukládání dat do interní databáze
- vestavěný WWW server pro parametrizaci a prezentaci dat
- robustnost řešení vhodná pro průmyslové prostředí
- operační systém LINUX

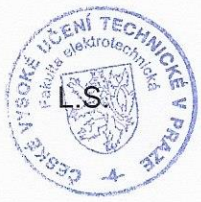
Podrobně specifikujte vnitřní komunikační procedury a implementujte blok autonomního řízení koncentrátoru prostřednictvím stavových automatů popsaných XML soubory.

Seznam odborné literatury:

[1] Hossain, E., Han, Z., Poor, H.V.: Smart Grid Communications and Networking, Cambridge University Press 2012, ISBN 978-1107014138
[2] DLMS/COSEM Architecture and Protocols Specification, DLMS User Association 2004

Vedoucí: doc. Ing. Jiří Novák, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015




prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14. 1. 2014

# Obsah

Seznam obrázků ..... 8

Seznam anglických pojmů a zkratek ..... 9

Seznam příloh ..... 190

1 Úvod ..... 11

2 Analýza ..... 13

2.1 Stávající řešení ..... 13

2.2 Konkurenční řešení ..... 13

2.3 Návrh struktury ..... 14

2.3.1 Obecná struktura ..... 14

2.3.2 Logování ..... 17

2.3.3 Zpracování chyb ..... 17

2.3.4 Správa informací ..... 18

2.3.5 Konfigurace ..... 19

2.4 Komunikace ..... 19

2.4.1 Prostředky operačního systému ..... 19

2.4.2 Provozní přenosy ..... 20

2.4.3 Servisní přenosy ..... 22

2.4.4 Velkokapacitní přenosy ..... 23

2.5 Návrh bloků ..... 25

2.5.1 Obecný návrh ..... 25

2.5.2 Jádro ..... 26

2.5.3 Řízení downlink rozhraní ..... 27

2.5.4 Řízení uplink rozhraní ..... 31

2.5.5 Řízení databáze ..... 32

2.5.6 Řízení chování ..... 34

2.5.7 Konzolové a webové rozhraní ..... 36

2.6 Vývojové prostředky ... 37
3 Syntéza ... 38
3.1 Projektové řízení ... 38
3.2 Knihovna pro meziblokovou komunikaci ... 39
3.2.1 Nižší úroveň a využití prostředků ... 39
3.2.2 Vyšší úroveň a API ... 40
3.3 Blok řízení chování ... 45
3.3.1 Obecné prostředky ... 45
3.3.2 Kontrolér ... 48
3.3.3 Chybový kolektor ... 51
3.3.4 Logger ... 52
3.3.5 Komunikátor ... 52
3.3.6 DLMS klient ... 53
3.3.7 Řadič stavových automatů ... 53
3.3.8 Plánovač ... 62
3.4 Testování a verifikace ... 66
4 Závěr ... 68
Reference ... 70
Příloha A ... 71
Příloha B ... 76
*PQueSys.h* ... 76
*Scheduler.h* ... 77
Příloha C ... 80
*Automata.h* ... 80
*AutomataDriver.h* ... 83

# Seznam obrázků

Obr. 1: Struktura softwarového vybavení koncentrátoru ..... 16
Obr. 2: Příklad průběhu velkokapacitního přenosu mezi dvěma bloky ..... 24
Obr. 3: Návrh stavového automatu bloků ..... 26
Obr. 4: Návrh struktury bloku jádra ..... 26
Obr. 5: Návrh struktury bloku řízení downlink rozhraní ..... 28
Obr. 6: Návrh struktury proxy bloku PLC sítě využívající standard PRIME nebo G3 ..... 30
Obr. 7: Návrh struktury bloku řízení databáze ..... 32
Obr. 8: Návrh struktury bloku řízení chování ..... 34
Obr. 9: Struktura hlavičky meziblokové zprávy ..... 40
Obr. 10: Využití obousměrných a jednosměrných kanálů pro zprávy ..... 42
Obr. 11: Příklad automatu vyčítání nově připojených měřidel ..... 55
Obr. 12: Proces kroku stavových automatů ..... 61
Obr. 13: Proces rozvrhování ..... 65

# Seznam anglických pojmů a zkratek

**AMM** – Advanced Measurement Management; síť pro energetická měření

**PLC** – Power Line Communication; komunikace prostřednictvím napájecí sítě

**DLMS/COSEM** – Device Language Messaging Specification / Companion Specification for Energy Metering; standard využívaný v AMM sítích pro přenos dat z měřidel prostřednictvím objektové výměnny

**OBIS** – Object Identification Systém; systém identifikace COSEM objektů

**SOAP** – Simple Object Access Protocol; rozšířený protokol pro výměnu objektů založený na protokolu HTTP

**DLC** – Downlink Control; blok řízení downlink rozhraní

**DLP** – Downlink Proxy; blok řízení downlink sítě

**ULC** – Uplink Control; blok řízení uplink rozhraní

**ULP** – Uplink proxy; blok řízení uplink sítě

**CGI** – Common Gateway Interface; rozšířená metoda generování dynamických webových stránek

**AWK** - Aho, Weinberger, Kernighan; skriptovací jazyk vhodný ke zpracování textu

**SIGKILL** – signál operačního systému Linux sloužící k zastavení chodu cílového procesu

**PID** – Process ID; unikátní identifikátor procesů v operačních systémech

**Unix timestamp** – časoměrná jednotka Unixových operačních systémů představující počet vteřin od 1. 1. 1970

**Watchdog** – hlídací obvod; jeho činnost může být emulována softwarově (byť s řadou omezení)

**RT signály** – Real-time signály; signály reálného času operačního systému Linux

**POSIX** – Portable Operating System Interface; standard definující rozhraní operačního systému

**SID** – Session ID; ID relace

**PRIME** – Powerline Intelligent Metering Evolution; jeden z používaných standardů PLC komunikace

**G3** – jeden z používaných standardů PLC komunikace

**PRIME-DC** – ovladač PLC modulu pro standard PRIME od společnosti Texas Instruments

**G3-DC** – ovladač PLC modulu pro standard G3 od společnosti Texas Instruments

**CORBA** – Common Object Request Broker Architecture; rozšířený standard pro výměnnu objektů

**BSD, MIT** – rozšířené formáty licencí otevřeného softwaru

**API** – Application Programming Interface; aplikační programovací rozhraní

**SDK** – Software Development Kit; sada nástrojů pro vývoj specifického softwaru

**UPPAAL** – systém pro verifikaci, simulaci a návrh časovaných stavových automatů

**Signal handler** – funkce pro zpracování signálu operačního systému Linux

**Wrapper** – kódový obalovač; část zdrojového kódy obalující určitou funkčnost jiné části zdrojového kódu

**FIFO** – First In First Out; metoda obsluhy fronty postupným odebíráním prvků v pořadí jejich přidání

**FCFS**- First Come First Server; stejný význam jako FIFO

# Seznam příloh

Příloha A – zkrácená verze hlavičkového souboru *libipc.h*……..71
Příloha B – zkrácená verze hlavičkových souborů *Automata.h* a *AutomataDriver.h*……...76
Příloha C – zkrácená verze hlavičkových souborů *PQueSys.h* a *Scheduler.h*……...80

# 1 Úvod

Diplomová práce se zabývá návrhem a implementací části softwarového vybavení datového koncentrátoru AMM (Advanced Measurement Management) sítě. Tyto sítě určené ke vzdálené správě energetických měření jsou hierarchicky strukturované a lze je rozdělit do dvou úrovní. Nižší úroveň AMM sítí je tvořena samotnými měřícími zařízeními. Těmito měřidly mohou být například elektroměry v energetické rozvodné síti, vodoměry či plynoměry a v typickém případě jsou propojena různými druhy sítí, jako je PLC (Power Line Communication), Wireless M-Bus, případně Ethernet. Pro vyčítání a reprezentaci dat z měřidel v těchto sítích je často využíván standard pro objektovou výměnu dat DLMS/COSEM. Tento standard sestává ze dvou částí – DLMS (Device Language Messaging Specification) specifikující komunikační protokol a COSEM (Companion Specification for Energy Metering) specifikující strukturu a datové typy objektů pro výměnu dat (viz. [1]). Vyšší úroveň AMM sítí je tvořena tzv. sítí datových ústředen, jejichž účelem je shromažďovat data z měřidel. Sítě datových ústředen jsou v typickém případě dostupné z celosvětové sítě Internet. Aby bylo možné data z měřidel dostat do datových ústředen, je nezbytné využít mezilehlá zařízení, která by vzájemně obě úrovně, tj. sítě měřidel a sítě datových ústředen, propojila. Takovým zařízením je datový koncentrátor. Jeho účelem však není pouze poskytovat vyšší úrovni rozhraní pro přístup k měřidlům, ale také sítě měřidel řídit. Proto datový koncentrátor bývá centrálním uzlem sítí měřidel využívajících stromovou topologii a master jednotkou v sítích využívajících řízení přístupu na médium založené na principu master-slave. Dále je běžné, že datový koncentrátor autonomně provádí sběr dat z měřidel a provádí aktualizaci jejich firmwaru.

Datový koncentrátor, návrhem a implementací jehož softwarového vybavení se zabývá tato diplomová práce, je vyvíjen jako součást většího projektu společnosti ZPA Smart Energy Trutnov a.s., který cílí na vývoj nové generace každé součásti AMM sítě. V rámci tohoto projektu jsou, kromě datového koncentrátoru, vyvíjeny datové ústředny a nové typy elektroměrů podporující agregaci dat z okolních měřidel. To znamená, že elektroměry budou provádět sběr dat i z okolních měřidel (vodoměrů, plynoměrů a podobně) využívajících například síť Wireless M-Bus a poté budou vyšším úrovním odesílat veškerá nasbíraná data.

Dále je v rámci tohoto projektu prováděn zmíněný vývoj nové generace datového koncentrátoru. Primární cílem tohoto vývoje není pouze použitelnost koncentrátoru v rámci projektem vyvíjené infrastruktury AMM sítě, ale také co největší kompatibilita tohoto koncentrátoru s jinými specifikacemi. Z hlediska softwarového vybavení to znamená především zobecnění komunikačních kanálů k měřidlům a datovým ústřednám (tj. zákaznickým sítím), jejich snadnou konfigurovatelnost a v neposlední řadě také snadnou rozšiřitelnost celého softwarového vybavení o případnou podporu v budoucnu vyvinutých komunikačních sítí. Vzhledem k rozšířenosti standardu DLMS/COSEM je také požadována podpora tohoto standardu vyvíjeným koncentrátorem.

Dalším požadavkem je autonomní chování koncentrátoru, které by mělo být v co největší míře programovatelné bez nutnosti zásahů do softwarového vybavení samotného. To by koncentrátoru umožnilo, mimo jiné, interoperabilitu s libovolnými zákaznickými specifikacemi.

Koncentrovaná data by měla být ve vyvíjeném zařízení ukládána do interní databáze za účelem přehledné a snadné správy těchto dat. Tato interní databáze by měla sloužit zejména pro ukládání dat v okamžicích nedostupnosti datové ústředny.

Vyvíjený datový koncentrátor by dále měl být vybaven interním webovým serverem, který by poskytoval nástroje pro správu měřidel, diagnostiku a sledování chodu jeho softwarového vybavení.

Zadavatelem projektu bylo také určeno, že vyvíjené softwarové vybavení by mělo být založeno na vestavěné verzi otevřeného operačního systému Linux.

Samozřejmostí je pak požadavek na robustnost celého softwarového vybavení vyvíjeného koncentrátoru. Jelikož zařízení tohoto typu je určeno k dlouhodobému běhu bez vnějšího servisního zásahu, je na místě, aby bylo navržené softwarové vybavení odolné proti chybám. V případě výskytu chyby by měl být datový koncentrátor schopný se z této chyby zotavit a případně odeslat hlášení o výskytu dané chyby do datové ústředny.

# 2 Analýza

## 2.1 Stávající řešení

Současná verze datového koncentrátoru společnosti ZPA Smart Energy Trutnov a.s. je označována jako CAM3500. Toto zařízení je založeno na procesoru s architekturou ARM9 a desce TS7260. Zařízení má dispozici až 128 MB RAM a je k němu možné připojit USB disk za účelem rozšíření paměťového prostoru a poskytnutí nevolatilního paměťového úložiště. Softwarové vybavení je založeno na vestavěné distribuci operačního systému Linux. K zajištění bezproblémového chodu je koncentrátor vybaven externím watchdogem a záložní baterií napájející modul hodin reálného času.

Koncentrátor CAM3500 lze vybavit řadou komunikačních modulů. Jako primární rozhraní k datové ústředně (tj. zákaznické síti) je uvažována TCP/IP komunikace, ADSL, Wi-Fi, GPRS nebo PPP. V případě potřeby může být zařízení vybaveno modemem GSM/GPRS, Wi-Fi modulem a externím komunikačním modulem připojeným přes rozhraní RS232, vnitřní nebo vnější USB2.0 port (např. modem PST). Pro připojení k síti měřidel prostřednictvím drátových spojů lze využít PLC moduly, rozhraní RS485 a M-Bus. Bezdrátově se lze k síti připojit pomocí modulů ZigBee 868/2.4 GHz, Wireless M-Bus nebo proprietárního řešení RF Wavenis od fy. Coronis (868 MHz). Z hlediska komunikačních standardů podporuje zařízení CAM3500 standard DLMS/COSEM.

Primární funkcí datového koncentrátoru CAM3500 je funkce směrovače / opakovače / překladače komunikace. To znamená, že zařízení umožňuje realizaci spojení mezi datovou ústřednou na zákaznické síti a koncovým měřidlem. Zařízení také umožňuje provádění hromadných úloh, jako například aktualizace firmwaru na celé skupině měřidel. Tyto úlohy jsou v koncentrátoru napevno dané a lze je konfigurovat pouze v omezené míře. Automaticky vyčtená data dokáže koncentrátor odesílat do datové centrály ve formě komprimovaných archivů. V rámci chodu koncentrátoru také vzniká řada logovacích souborů o různých kritických událostech (například chyba komunikace) a také řada alarmů (například otevření krytu koncentrátoru). Tyto logovací soubory a alarmy lze volitelně odesílat do datové centrály.

Jak bylo uvedeno, cílem této diplomové práce je navrhnout softwarové vybavení nové generace datového koncentrátoru označovaného jako CAM3600. Tento nový koncentrátor by měl oproti přechozí generaci poskytovat zejména větší autonomii, větší konfigurovatelnost, snadnější rozšiřitelnost a v neposlední řadě také vhodnější správu koncentrovaných dat.

## 2.2 Konkurenční řešení

V dnešní době existuje řada datových koncentrátorů pro AMM sítě, nicméně většina z nich sdílí společné nevýhody. Zařízení jsou navrhována dle jedné konkrétní specifikace od jedné konkrétní energetické společnosti a při použití tohoto zařízení v energetickém systému, který tuto specifikaci nesplňuje, je díky téměř nulové modifikovatelnosti zařízení nutné přizpůsobit měřidla této dané specifikaci. Obtížná modifikovatelnost zařízení je dána zejména implementací podpory standardu

DLMS/COSEM a systémem úloh. Zařízení sice standard DLMS/COSEM podporuje, nicméně OBIS kódy jednotlivých COSEM objektů jsou v zařízení napevno zadané, přestože se v závislosti na specifikaci mohou lišit. Ve většině zařízení existuje řada automaticky běžících úloh, jako je automatické vyčítání dat z nově připojených měřidel nebo automatická správa sítě měřidel a pak systém konfigurovatelných úloh. Byť jsou nazývány konfigurovatelnými, u těchto úloh lze nastavit pouze omezené množství parametrů. Nastavit lze dobu startu úlohy, počet opakování, periodicitu, prioritu, seznam měřidel, kterých se úloha týká a nakonec typ úlohy. Například u úlohy vyčítání COSEM objektů z měřidel nelze volit, jaké objekty mají být vyčteny a vždy je vyčítána pevně daná množina objektů. Zařízení obvykle disponují webovým rozhraním, pomocí nějž lze vyčítat data z měřidel ručně, nicméně opět nelze zvolit objekty k vyčtení. Navrhovaný datový koncentrátor by tedy měl nabízet značně větší možnosti v konfiguraci úloh a možnost snadného přizpůsobení prováděných operací dané zákaznické specifikaci. Dalším nedostatkem konkurenčních zařízení může být například omezená podpora zákaznických sítí. Každé zařízení používá proprietární protokoly a často je podporováno pouze ukládání vyčtených dat ve formě XML souboru na FTP server nebo protokol pro výměnu objektů SOAP. Koncentrátor CAM3600 by měl nabídnout širší podporu zákaznických sítí a možnost přizpůsobení rozhraní konkrétním požadavkům. K dalším nevýhodám konkurenčních koncentrátorů lze řadit také častou nestabilitu a neprůhlednost jejich operací. V zařízeních často dochází k chybám, zamrzání řídícího programu a zasekávání úloh, a často nelze dohledat konkrétní příčinu chyby. Navrhované softwarové vybavení by tedy mělo být dostatečně odolné proti chybám a poskytovat uživatelům detailní záznamy o své činnosti.

## 2.3 Návrh struktury

### 2.3.1 Obecná struktura

Při návrhu struktury systému softwarového vybavení koncentrátoru je nezbytné vyjít z činností, které by měly být koncentrátorem prováděny. Tyto činnosti jsou shrnuty v následujících bodech.

- **Řízení rozhraní k zákaznickým sítím** – je žádoucí, aby koncentrátor byl schopen řídit více druhů rozhraní a aby ve srovnání s konkurenčními řešeními bylo snadnější přidat specifická rozhraní na žádost zákazníků.
- **Řízení rozhraní k sítím měřidel** – pro tato rozhraní platí stejná tvrzení, jako u rozhraní k zákaznickým sítím.
- **Podpora standardu DLMS/COSEM** – tento standard poskytuje vhodnou reprezentaci měřidel i dat. Tento standard je podporován jak koncentrátorem CAM3500, tak konkurenčními zařízeními.
- **Autonomní řízení** – jedním z požadavků zadavatele je schopnost autonomní činnosti koncentrátoru. To se týká například automatického vyčítání měřidel, odesílání dat do zákaznických sítí, aktualizace firmwaru měřidel atd. Dále je vyžadováno, aby, na rozdíl od konkurenčních řešení, bylo autonomní chování koncentrátoru snadno konfigurovatelné a modifikovatelné pro konkrétní zákaznickou specifikaci bez nutnosti razantních zásahů do systémů koncentrátoru.

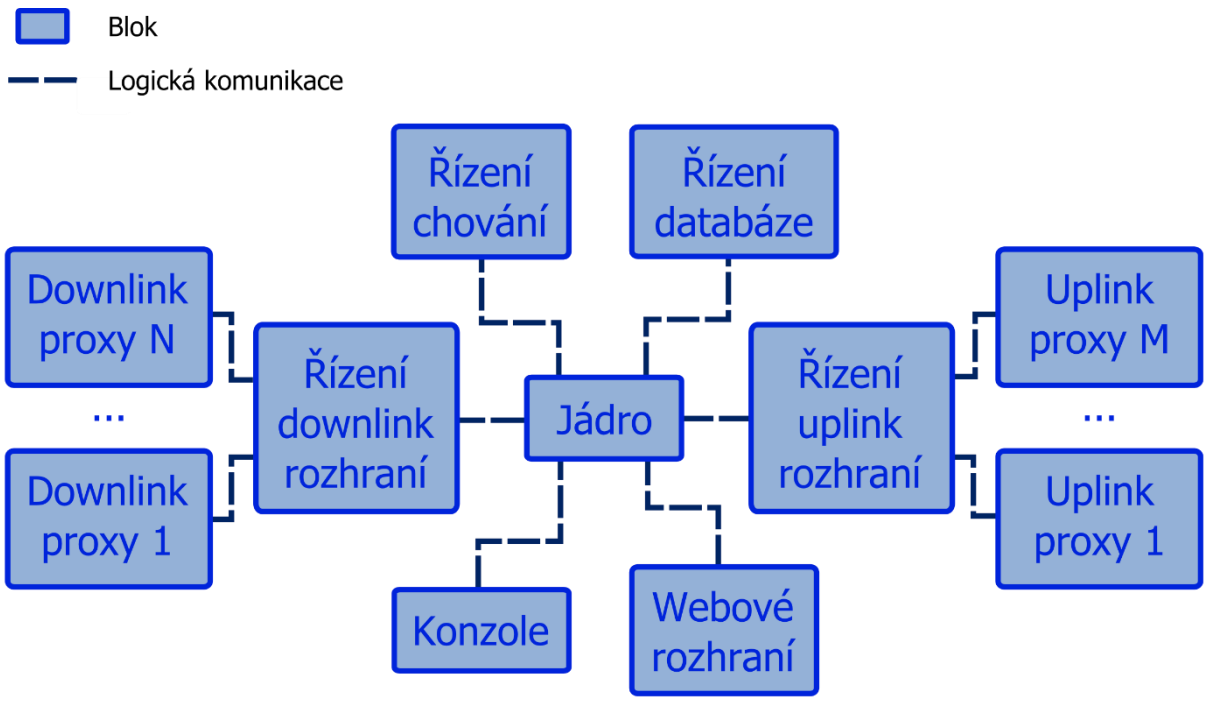
- **Řízení interní databáze** – koncentrovaná data, tj. data z měřidel a logy o událostech nastalých v zařízení, je vhodné ukládat do interní databáze. To oproti prostému ukládání dat do souborů umožňuje snadnou realizaci pozdější správy těchto dat ze zákaznické datové ústředny. Tuto databázi je tedy nutné spravovat a řídit.
- **Správa hardwaru a systémových prostředků** – využívané prostředky operačního systému je třeba vhodným způsobem spravovat. To samé platí o využitém hardwaru, včetně externího watchdogu, který bude použit k resetování koncentrátoru při výskytu závažné chyby.
- **Poskytování webového rozhraní** – webové rozhraní by mělo sloužit pro jednoduché zobrazení stavu zařízení a ke změně základních položek nastavení.
- **Poskytování konzolového rozhraní** – konzolové rozhraní by mělo být používáno pro provádění pokročilých operací a ke změně pokročilých a kritických nastavení koncentrátoru.

Nejjednodušší variantou struktury softwarového vybavení je tzv. monolitická struktura, která spočívá v použití jediné aplikace, která by zajišťovala provádění všech požadovaných činností. Tato aplikace může být, v případě potřeby, rozdělena na více vláken. V takovém případě by mohlo každé vlákno provádět některou z vyžadovaných činností. Výhodou monolitické struktury je zejména jednoduchost spouštění a ukončování softwarového vybavení, neboť je třeba spouštět nebo ukončovat pouze jedinou aplikaci. Nevýhody této struktury však tuto výhodu zdaleka převáží. Jedná se o velkou nepřehlednost zdrojového kódu, složitý vývoj při kooperaci více vývojářů a obtížnou rozšiřitelnost. Jelikož jedním z požadavků na softwarové vybavení je jeho snadná rozšiřitelnost, není toho řešení žádoucí.

Vhodnější je tedy rozdělit softwarové vybavení do několika dílčích aplikací tak, že každá dílčí aplikace (dále označována jako blok) bude provádět konkrétní vyžadovanou činnost. Každá z uvedených činností tedy bude prováděna jedním konkrétním blokem. Činnosti prováděné bloky řízení rozhraní k zákaznickým sítím a sítím měřidel je navíc žádoucí dále rozdělit na bloky řízení jednotlivých instancí různých druhů rozhraní. Je zřejmé, že tyto bloky pro řízení konkrétních instancí konkrétních rozhraní budou hierarchicky spadat pod bloky řízení rozhraní k zákaznickým sítím a sítím měřidel. Tento přístup hierarchického členění je možné zobecnit na všechny bloky, které lze pak rozdělit na bloky nadřazené a podřízené. Každý podřízený blok bude hierarchicky spadat pod některé z nadřazených bloků. Nadřazené bloky, kromě vyžadovaných specifických činností, také budou provádět kontrolu a řízení chodu bloků jím podřízených. U některých bloků však může být takové řízení jejich chodu nežádoucí, zejména proto, že budou spouštěny z aplikací mimo tuto strukturu, a proto budou z takového řízení vyřazeny. Tyto bloky budou dále označovány jako nesledované (a ostatní jako sledované).

Logická komunikace mezi bloky by, kvůli zachování konzistence návrhu, měla v rámci možností respektovat hierarchickou strukturu celého systému. Každý blok tedy bude, pokud to bude možné, komunikovat pouze se sousedícími bloky v hierarchické struktuře, tj. se svými podřízenými bloky a se svým nadřazeným blokem.

Na následujícím obrázku je znázorněna celá znázorněná hierarchická struktura.




**Obr. 1:** Struktura softwarového vybavení koncentrátoru

Softwarové vybavení koncentrátoru tedy bude rozděleno do následujících bloků.

- **Jádro** – bude provádět správu hardwaru a systémových prostředků. Spuštění tohoto bloku vyvolá, prostřednictvím hierarchickému spouštění, rozběh celého softwarového vybavení datového koncentrátoru.
- **Řízení downlink rozhraní (DLC)** – bude provádět činnosti spojené se správou rozhraní k sítím měřidel, a tedy spravovat seznam měřidel, monitorovat jejich stav a zajišťovat přenos dat mezi měřidly a koncentrátorem.
- **Downlink proxy (DLP)** – bude řídit komunikaci je jednom určitém druhu rozhraní k síti měřidel. Pokud bude existovat více instancí jednoho druhu rozhraní, pak pro každou instanci bude spuštěn jeden konkrétní DLP blok daného typu rozhraní. Tento blok také bude převádět data z měřidel z jejich interní reprezentace do formy specifikované standardem DLMS/COSEM, pokud to bude potřeba.
- **Řízení uplink rozhraní (ULC)** – bude provádět činnosti spojené se správou rozhraní k zákaznickým sítím, jako je zajištění přenosu dat mezi zákaznickými sítěmi a koncentrátorem.
- **Uplink proxy (ULP)** – bude řídit komunikaci na jednom určitém druhu rozhraní k zákaznickým sítím. Stejně jako v případě DLP bloků bude možné používat více instancí ULP bloku pro konkrétní typ rozhraní, pokud to bude potřeba.
- **Řízení databáze** – bude provádět činnosti spojené s řízením interního databázového systému. Bude zajišťovat zápis dat do databáze, jejich vyčítání a také údržbu databázového systému.
- **Řízení chování** – bude zajišťovat autonomní řízení koncentrátoru. Tento konkrétní blok bude důvodem hlavní odlišnosti od konkurenčních řešení – snadno konfigurovatelného chování. Tento blok také bude zajišťovat podporu standardu DLMS/COSEM.

- **Konzole** – bude poskytovat uživatelské konzolové rozhraní. Tento blok bude nesledovaný.
- **Bloky webového rozhraní** – jedná se o programy generující webové rozhraní prostřednictvím CGI webového serveru. Tyto bloky také nebudou sledované.

Detailní popisy jednotlivých bloků budou následovat v kapitole 2.5 Návrh bloků.

Spouštění takto strukturovaného softwarového vybavení bude respektovat jeho hierarchickou strukturu. Po spuštění bloku na nejvyšší úrovni hierarchie (bloku jádra) tento blok spustí, ve formě démonů, své sledované podřízené bloky. Každý z podřízených bloků pak po svém spuštění spustí, formou démonů, své sledované podřízené bloky. Tímto způsobem budou postupně spuštěny všechny sledované bloky.

Podpora standardu DLMS/COSEM je zajištěna především blokem řízení chování, který bude obsahovat DLMS/COSEM klientskou aplikaci. Veškerá data z měřidel tedy budou zasílána do tohoto bloku, který data bude zpracovávat. Pokud by síť, ke které je dané měřidlo připojeno, nepodporovala DLSM/COSEM standard, bude úkolem DLP bloku tohoto rozhraní překládat data z měřidel připojených k této síti do formátu definovaného standardem DLMS/COSEM.

Je zřejmé, že takto navržená struktura nejen že splňuje požadavek na snadnou rozšiřitelnost, ale také usnadňuje vývoj, správu zdrojového kódu a kooperaci více vývojářů.

### 2.3.2 Logování

V systému bude docházet k různým provozním událostem, které bude potřeba nějakým způsobem logovat. Každý blok bude k logování primárně používat logovací databázi, do které budou logy zapisovat prostřednictvím bloku řízení databáze. Pokud z jakéhokoliv důvodu nebude možné požadavek na zápis logu odeslat (například pokud bude logovanou situací výskyt chyby při komunikaci), bude třeba jej zaznamenat jiným způsobem. Nejjednodušším způsobem je přímý zápis do logovacích souborů, nicméně při nečekaném ukončení aplikace v průběhu zápisu logu není zaručeno, že daný log bude do souboru skutečně zapsán. Vhodnějším způsobem je využití logovacího systému syslog, který je dostupný v použité distribuci operačního systému Linux (viz. [2]). Vzhledem k tomu, že tento systém je sdílen vícero běžícími aplikacemi, je nezbytné při vyčítání logů informace z ostatních aplikací odfiltrovat. To lze například pomocí jednoduchého AWK skriptu.

Vzhledem k jednoduchosti řešení zápisu logů do vlastních logovacích souborů, kdy stačí pouze přesměrovat standardní proudy při spuštění bloku, bude tato metoda využívána zejména v průběhu vývoje a v pozdější fázi bude nahrazena využitím logovacího systému syslog.

### 2.3.3 Zpracování chyb

V každém bloku může dojít k řadě chybových situací. Je však třeba odlišovat chybové situace očekávané a neočekávané.

Očekávaná chybová situace je taková, u níž předpokládáme výskyt za běžného provozu koncentrátoru a u které nepředpokládáme narušení základního běhu bloků. Takovou situací může být například případ, kdy se blok řízení chování pokusí zapsat data do databáze v okamžiku, kdy byla překročena maximální velikost databázového souboru, nebo případ, kdy se DLC blok pokusí odeslat data prostřednictvím DLP

bloku v okamžiku, kdy je odesílací fronta tohoto DLP blok plná. Takové chyby nesmí v žádném případě narušit normální chod bloku a každý blok by tedy měl takovou situaci korektně vyřešit a případně její výskyt zapsat do logu.

Neočekávaná chybová situace je taková, jejíž výskyt je nepravděpodobný, značí možnost selhání systému či hardwaru a negativně ovlivňuje základní běh bloku. Příkladem takové situace může být případ, kdy ovladač PLC modemu přestane odpovídat DLP bloku PLC sítě, nebo případ, kdy databázový systém bude při pokusu o manipulaci s databází hlásit nestandardní chybu. Za účelem sledování výskytu takových chyb bude každý sledovaný blok spravovat vlastní chybový čítač a při výskytu neočekávané chyby by se měl pokusit korektně chybu vyřešit, inkrementovat tento čítač a případně výskyt chyby zapsat do logu. Pravidelně bude tento čítač kontrolovat a v případě, že počet chyb za periodu s konfigurovatelnou délkou (tzv. chybové okno) překročí konfigurovatelný práh (tzv. maximální chybovou frekvenci), přejde do chybového stavu. Protože každý nadřazený blok bude mít přehled o stavech svých podřízených bloků, dozví se, že blok přešel do chybového stavu (detaily o tomto mechanismu jsou v následující kapitole). Poté se může nadřazený blok vynucením změny stavu pokusit daný podřízený blok restartovat. Zároveň však nadřazený blok musí inkrementovat svůj chybový čítač. Takto bude opakovaný výskyt chyb propagován až na nejvyšší úroveň hierarchie, do bloku jádra. Ten, jako blok na nejvyšší úrovni hierarchie, při překročení prahu jeho chybového čítače nepřejde do chybového stavu, ale tuto skutečnost řádně zapíše do logu a poté přestane budit externí watchdog, což vyvolá hardwarový restart celého systému.

Nabízí se také možnost, že nadřazené bloky budou spravovat dodatečný chybový čítač pro každý jejich podřízený blok. Při výskytu chybového stavu u podřízeného bloku a následnou změnou jeho stavu by nadřazený blok inkrementoval příslušný chybový čítač. Po jeho přetečení by pak nadřazený blok ukončil běh příslušného podřízeného bloku zasláním signálu SIGKILL a poté by podřízený blok znovu spustil. Tento mechanismus je však pravděpodobně zbytečný a jeho případné použití bude ponecháno na implementátorech jednotlivých nadřazených bloků.

### 2.3.4 Správa informací

Každý blok bude spravovat informace týkající se jeho specifických činností, nicméně všechny nadřazené bloky budou spravovat následující informace.

- **Data sledování podřízených bloků** – bude sloužit ke sledování stavu podřízených bloků. Jedná se zejména o informace o aktuálním stavu daného podřízeného bloku, jeho PID, typ, časovou značku posledního okamžiku odeslání žádosti o stav a časovou značku posledního okamžiku přijetí odpovědi na tento požadavek.
- **Tabulka distribuce událostí** – bude sloužit k řízení distribuce událostí z daného nadřazeného bloku do jeho podřízených bloků. Bude se skládat především ze seznamu událostí a bloků, které si odběr upozornění o výskytu dané události zaregistrovali.

Jak je vidět, v řadě případů bude potřeba spravovat v blocích různé časové značky. V operačním systému Linux je vhodné je reprezentovat pomocí tzv. Unix timestamp, což je standardní jednotka k měření absolutního času v Unixových operačních systémech. Jedná se o počet sekund „od začátku Unixové epochy“ neboli počet vteřin od 1. 1. 1970.

### 2.3.5 Konfigurace

Konfigurace každého z uvedených bloků bude určena obsahem jeho konfiguračního souboru. Konfigurační soubor každého bloku bude ve formátu XML a bude obsahovat následující položky.

- **Časový limit blokujících operací** – bude použit při provádění blokujících operací v rámci meziprocesní komunikace.
- **Nastavení chybového čítače** – bude sestávat z povolení chybového čítače, nastavení periody kontroly čítače (tzv. délky chybového okna bloku) a limitu (tzv. maximální chybové frekvence), po jehož překročení vstoupí blok do chybového stavu.

Každý podřízený blok bude mít v konfiguračním souboru navíc následující položku.

- **Časový limit registrace** – bude určovat čas, jaký bude blok vyčkávat na odpověď od nadřazeného bloku na žádost o registraci.

Následující položky budou přítomny pouze v konfiguračních souborech nadřazených bloků.

- **Seznam podřízených bloků** – bude určovat, jaké podřízené bloky má daný nadřazený blok řídit, cesty k jejich spustitelným souborům a cestu ke konfiguračnímu souboru daného bloku, se kterými má být daný podřízený blok spuštěn, která bude do bloku předávána jako parametr při spouštění.
- **Perioda kontroly stavu podřízených bloků** – časový údaj ve vteřinách, který bude určovat četnost kontroly stavu podřízených bloků prostřednictvím mechanismu softwarového watchdogu.
- **Časový limit softwarového watchdogu** – bude určovat časový okamžik od okamžiku odeslání žádosti o stav do určitého podřízeného bloku, po jehož uplynutí se nadřazený blok pokusí tento daný podřízený blok znovu spustit (více detailů v kapitole 2.4.3.3 Sledování podřízených bloků).

Konfigurační soubory jednotlivých bloků dále budou obsahovat položky specifické pro daný blok. Obsah konfiguračních souborů jednotlivých bloků jsou uveden v kapitole 2.5 Návrh bloků.

## 2.4 Komunikace

### 2.4.1 Prostředky operačního systému

Vzhledem k rozmanitým požadavkům na spojení mezi jednotlivými bloky byly komunikační přenosy v systému rozděleny do tří typů. Jedná se o přenosy provozních dat, servisních dat a velkokapacitní přenosy. Na všechny tyto typy přenosů je v operačním systému Linux vhodné vyžít zejména fronty zpráv, pojmenované roury, semafory a signály reálného času (viz. [4]).

Fronty zpráv přenáší vždy definovaný počet bajtů z paměti, a tudíž jsou určeny k blokovému (paketovému) přenosu dat. Minimální přenositelná jednotka odpovídá zprávě velikosti jedné proměnné datového typu *long int* (velikost 4 bajty na použité architektuře). Hodnota této položky je pak využita při vyčítání zpráv z fronty, kdy je do výstupního bufferu uložen daný počet bajtů z nejstarší zprávy ve frontě s danou hodnotou této položky. Toto se dá využít například k odlišení typů, adresátů, odesílatelů, nebo priorit zasílaných zpráv, což činí z front zpráv ideální hlavní prostředek pro meziblokovou komunikaci. Vzhledem k tomu, že k jediné frontě zpráv může přistupovat několik procesů najednou, bylo by dokonce možné mít jedinou frontu zpráv, případně mít například jednu frontu zpráv na každou úroveň hierarchie. Tato řešení však sdílí společný nedostatek, a tím je omezená velikost fronty. Při sdílení fronty

mezi více procesy se s každým zúčastněným procesem zvyšuje riziko zaplnění fronty. Protipólem tohoto způsobu komunikace je pak řešení, ve kterém má každý blok vlastní vstupní a výstupní frontu. Ani toto řešení ovšem není ideální, neboť fronty zpráv by takto zabraly zbytečně veliké množství paměti. Vhodné je tedy použití kompromisu, kdy každý blok při spuštění vytvoří jedinou frontu zpráv, kterou bude používat pro přijímání zpráv. Zároveň se připojí na vstupní frontu zpráv svého nadřízeného bloku, kterou bude používat pro odesílání zpráv. Tento způsob vyžaduje využití pouze tolika front, kolik bude v systému bloků, což je poloviční počet oproti druhému řešení a současně představuje dostatečně malé riziko přeplnění front.

Pojmenované roury jsou oproti tomu určeny k proudovému přenosu dat. Při vytvoření pojmenované roury je v zadané lokaci v souborovém systému vytvořen tzv. virtuální soubor (tedy uzel souborového systému, který neodkazuje na žádná data v paměťovém úložišti), který je pak použit k přístupu do odpovídající pojmenované roury. Pojmenované roury jsou vhodné zejména k přenosu větších objemů dat, kdy by použití front zpráv nebylo, vzhledem k velikosti přenášených dat, výhodné, protože zpráva by musela být opakovaně dělena.

Semafory v operačním systému Linux jsou stejné, jako v jakémkoliv jiném operačním systému POSIXového typu. V komunikaci je lze využít například pro synchronizaci přenosů.

Signály reálného času (dále jen RT signály), se od klasických signálu známých ze všech operačních systému POSIXového typu liší zejména tím, že při přijetí jsou před samotným zpracováním vkládány do signálových front adresovaných procesů. Tato serializace signálů zaručuje, že na rozdíl od standardních signálů je každý přijatý RT signál zpracován adresovaným procesem. Ke každému RT signálu je navíc možné přiřadit jeden argument velikosti *long int*, nebo jeden ukazatel (tedy velikosti 4 bajtů na použité architektuře), což z RT signálů činí využitelný prostředek například pro kritickou komunikaci.

### 2.4.2 Provozní přenosy

Přenosy provozních dat budou probíhat prostřednictvím front zpráv. Kvůli zachování konzistence návrhu je nutné, aby tyto přenosy respektovaly logickou stromovou topologii komunikace. Každý zpráva je tedy přeposlána z daného bloku buď do odpovídajícího podřízeného bloku (pokud daný blok nějaký má), nebo do jemu nadřazeného bloku. Ten obdobně buď přepošle tuto zprávu odpovídajícímu adresátovi, pokud je mezi jeho podřízenými bloky, nebo přepošle zprávu do svého nadřazeného bloku. Toto se může opakovat, dokud zpráva nedosáhne bloku jádra, ve kterém je přesměrována do příslušného podřízeného bloku, který ji případně přesměruje do svého podřízeného bloku, a tak podobně, dokud zpráva nedosáhne svého adresáta.

Existují dvě možnosti, jak realizovat směrování zpráv v nadřazených blocích, a to adresace uzlů a adresace zpráv. Při adresaci uzlů měl každý blok svou unikátní adresu, která by byla hierarchická. Každá zpráva by pak obsahovala adresy cíle a odesílatele. Nadřazené bloky by pak přeposílaly zprávy na základě dané hodnoty adresy na dané úrovni hierarchie (obdobně fungují například sítě založené na protokolu IP). Výhodou adresace zpráv je, že každá zpráva má pevně daného příjemce a navíc ve stromové topologii existuje pro každou zprávu jediná cesta. Komunikace je tak velmi předvídatelná a už z adresy ve zprávě lze vyčíst cestu, kterou bude zpráva směrována. Nevýhodou je fakt, že každý blok musí o existenci

ostatních bloků vědět a musí také vědět, jaké typy zpráv lze do ostatních bloků zasílat. Při adresaci zpráv by byl ke každé zprávě přiřazen unikátní typ. Nadřazené bloky se pak tímto typem při směrování řídí (neboli nadřazený blok pozná, kam má danou zprávu přeposlat). Nevýhodou je například absence informace o odesílateli zprávy, která je nezbytná pro zasílání odpovědí. Výhodou je fakt, že bloky nemusí o existenci bloků, se kterými jsou spojeny, vůbec vědět. Jelikož celý systém by měl být snadno rozšiřitelný, bude směrování provozních přenosů primárně založeno na adresaci zpráv.

Přenos provozních dat bude založen na principu požadavek – odpověď. Pokud blok obdrží požadavek, musí na něj odpovědět (pozitivně nebo negativně). Vzhledem k tomu, že směrování bude založeno na adresaci zpráv, bude tedy nutné do zprávy doplnit informaci o odesílateli. Blok, který bude odpovídat na požadavek, tuto položku zkopíruje do odpovědi a nebude se zabývat jejím významem. Nadřazený blok směrující zprávu s odpovědí však tuto informaci využije ke správnému nasměrování zprávy s odpovědí do svého podřízeného, nebo nadřazeného bloku. Tato informace se bude skládat ze dvou položek, a to PID odesílajícího bloku a identifikačního čísla relace. Identifikační číslo relace (Session ID, dále jen SID) bude sloužit zejména pro blok odesílatele k odlišení jednotlivých požadavků od sebe. Obě tyto hodnoty pak budou využity nadřazenými bloky ke směrování. To je možné realizovat například tak, že při přijetí požadavku, který by měl být přesměrován do jiného bloku si nadřazený blok, vytvoří záznam v lokální směrovací tabulce skládající se z těchto dvou položek a PID podřízeného bloku, ze kterého požadavek přišel. Pokud obdrží nadřazený blok odpověď na nějaký požadavek, který sám nevyslal, nalezne záznam v této směrovací tabulce se stejnými hodnotami těchto položek a přesměruje zprávu do bloku s daným PID.

Kvůli usnadnění diagnostiky bude každý nadřazený blok využívat následující zprávy provozní komunikace.

- **Žádost o stav podřízených bloků** – bude použita nadřízeným blokem k získání přehledu o podřízených blocích svých podřízených bloků. Tato zpráva bude využívána kvůli diagnostice softwarového vybavení koncentrátoru.
- **Stav podřízených bloků** – zasílána nadřízenému bloku podřízeným blokem jako odpověď na požadavek na stav podřízených bloků. Bude obsahovat seznam podřízených bloků, jejich PID, typ, aktuální stav a časovou značku okamžiku vstupu do tohoto stavu.

#### *2.4.2.1 Distribuce událostí*

Jak již bylo řečeno, v systému bude vznikat řada událostí, na jejichž výskyt bude třeba upozornit bloky, které budou mít o takové upozornění zájem. Stejně jako v případě běžných provozních přenosů je nutné, aby celý tento mechanismus respektoval logickou stromovou topologii komunikace. Mechanismus distribuce událostí tedy bude fungovat následovně. Při vzniku události zkontroluje blok, ve kterém událost vznikla, jestli některý z jeho podřízených bloků (pokud nějaké má) si odběr notifikace o této události zaregistroval. Pokud ano, tak notifikaci o vzniku této události odešle tomuto bloku. Tuto notifikaci poté vždy odešle svému nadřazenému bloku. Při přijetí notifikace o události ji blok, v případě, že sám si odběr této notifikace zaregistroval, zpracuje, a poté přepošle do těch svých podřízených bloků, které si odběr této

notifikace zaregistrovali. Na rozdíl od ostatní provozní komunikace nebude notifikační zpráva potvrzována odpovědí. Použit bude jediný následující typ zpráv.

- **Notifikace o události** – zasílána blokem, ve kterém událost vznikla. Tato zpráva nevyžaduje žádnou odpověď.

### 2.4.3 Servisní přenosy

Servisní data budou použita nadřazenými bloky k řízení chodu jejich podřízených bloků, a tedy jejich přenos by měl být spolehlivý a měl by mít vyšší prioritu, než ostatní typy přenosů. K těmto přenosům tedy budou využity RT signály a fronty zpráv při využití výše zmíněného rozlišení priorit zpráv prostřednictvím první položky ve zprávě.

RT signály budou využity pouze pro nouzovou změnu stavu bloku. Pokud nadřazený blok detekuje chybový stav bloku, může změnit jeho stav a například tak vynutit znovunačtení jeho konfigurace. K tomuto účelu bude využit jediný RT signál, jehož zasílaným argumentem bude nový stav adresovaného bloku.

K přenosu ostatních servisních zpráv budou využity fronty zpráv. Aby byly odlišeny servisní zprávy od provozních, budou mít servisní zprávy odlišnou hodnotu první povinné položky ve zprávě. Jako servisní budou přenášeny přenosy registrace, distribuce událostí a sledování bloků.

#### *2.4.3.1 Registrace*

K procesu registrace dojde v blocích úrovně 1 a více vždy po spuštění bloku. Jeho účelem je dát nadřazenému bloku k dispozici informace nezbytné k řízení podřízeného bloku, a tedy klíč ke vstupní frontě zpráv a PID daného bloku. V rámci procesu registrace budou využity následující typy kritických zpráv.

- **Žádost o registraci** – zasílána podřízeným blokem nadřazenému bloku při spouštění bloku. Tato zpráva bude obsahovat výše zmíněné potřebné informace.
- **Potvrzení / zamítnutí registrace** – zasílána nadřazeným blokem podřízenému bloku jako odpověď na žádost o registraci. Pokud blok obdrží zprávu o zamítnutí registrace nebo pokud vyprší časový limit, uvolní veškeré zabrané prostředky a ukončí svůj běh.
- **Zrušení registrace** – zasílána podřízeným blokem nadřazenému bloku při ukončování běhu. Po obdržení této zprávy dojde k vyřazení odesílajícího bloku ze sledování adresovaným nadřazeným blokem. Tato zpráva nevyžaduje žádnou odpověď.

#### *2.4.3.2 Řízení distribuce událostí*

Proces řízení odběru notifikací o událostech bude fungovat obdobně, jako mechanismus distribuce událostí samotný. Pokud blok odešle registraci k odběru notifikací svému nadřazenému bloku, ten si tuto informaci uloží a poté, v případě, že daná událost nevzniká v tomto nadřazeném bloku, ani v žádném z jeho podřízených bloků, odešle stejnou registraci do svého nadřazeného bloku (pokud nějaký má). V rámci distribuce událostí budou využity následující typy kritických zpráv.

- **Registrace odběru notifikací o událostech** – zasílána podřízeným blokem nadřazenému bloku, zpravidla při spouštění bloku. Obsahuje seznam událostí, jejichž notifikace chce odesílající podřízený blok odebírat. Tato zpráva nevyžaduje žádnou odpověď.
- **Zrušení odběru notifikací o událostech** – zasílána podřízeným blokem nadřazenému bloku, zpravidla při ukončování bloku. Tato zpráva nevyžaduje žádnou odpověď.

#### *2.4.3.3 Sledování podřízených bloků*

Každý nadřazený blok by měl zajistit, že jemu podřízené bloky jsou spuštěné a připravené přijímat požadavky. Za tímto účelem bude každý nadřazený blok periodicky odesílat žádost o aktuální stav do všech svým podřízených bloků. Pokud daný podřízený blok neodpoví do daného časového limitu, nebo pokud získaný stav podřízeného bloku bude chybový, nadřazený blok se tento blok pokusí restartovat prostřednictvím RT signálu s žádostí o změnu stavu na stav studeného či horkého startu, případně vynucení ukončení a opětovného spuštění bloku. Tímto způsobem budou sledovány všechny podřízené bloky, kromě tzv. nesledovaných bloků. Tento mechanismus bude dále označován jako softwarový watchdog a bude využívat následující kritické zprávy.

- **Žádost o stav** – zasílána periodicky nadřazeným blokem každému jeho podřízenému bloku.
- **Stav bloku** – zasílána podřízeným blokem nadřazenému bloku, jako odpověď na žádost o stav. Obsahuje aktuální stav, ve kterém se odesílající blok nachází.

### 2.4.4 Velkokapacitní přenosy

Jedinou zprávou lze pomocí front zpráv přenést pouze omezené množství dat. V případě, že je třeba přenést větší objem dat, je nutné využít velkokapacitních přenosů. Ty je možné realizovat několika způsoby.

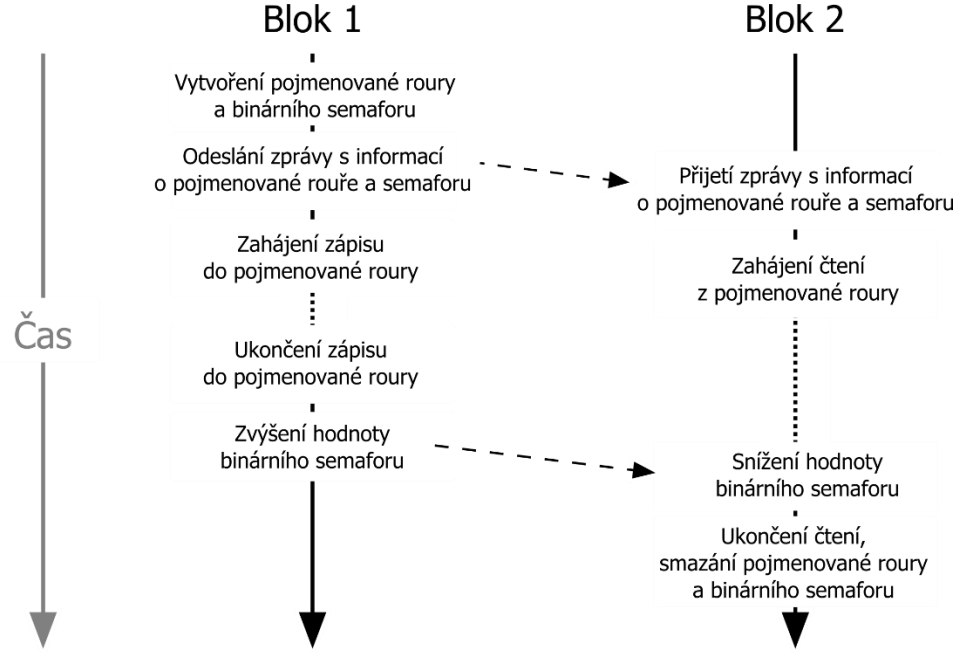
Jedno z možných řešení je fragmentace zpráv. Ta spočívá v dělení příliš velkých zpráv na dostatečně malé dílčí zprávy (fragmenty). Po přijetí by byly tyto fragmenty původní zprávy složeny do jediné zprávy. Výhoda tohoto řešení spočívá zejména v tom, že nevyužívá žádné další prostředky operační systému. Nevýhodou je pak zejména riziko zvýšeného zpoždění přenosu fragmentovaných zpráv způsobené nutností přenášet více zpráv.

Dalším řešením je například využití souborů k přenosu velkých objemů dat. Místo toho, aby blok odesílal objemná data přímo, uloží je do souboru. Cestu k tomuto souboru pak odešle adresátovi ve formě standardní provozní zprávy. Toto řešení by bylo výhodné zejména díky snadné implementaci a díky tomu, že nepředstavuje žádnou zátěž systému front zpráv navíc. Nevýhodou jsou pak zejména dlouhé časy přístupů k souborům na paměťových úložištích. Tato nevýhoda by se sice dala odstranit využitím souborového systému v operační paměti, nicméně to by natrvalo snížilo maximální využitelnou velikost operační paměti.

Vhodným kompromisem mezi oběma přístupy je využití pojmenovaných rour. Blok odesílající objemná data vytvoří pojmenovanou rouru, do které data zapíše stejným způsobem, jako kdyby je zapisoval do souboru. Adresovanému bloku poté odešle standardní zprávu obsahující cestu k této pojmenované rouře. Adresovaná blok pak data z pojmenované roury vyčte stejným způsobem, jako by je

vyčítal ze souboru. Výhodou tohoto řešení je snadná implementace řešení. Navíc, jelikož pojmenované roury nejsou skutečné soubory a data do nich zapsaná jsou uložena pouze v operační paměti, je jejich použití výhodnější, než použití standardních souborů. Toto řešení je tedy pro velkokapacitní přenosy nejvhodnější a bude použito.

Je nezbytné, aby odesílající blok mohl adresovanému bloku signalizovat konec zápisu a adresovaný blok tak mohl vyčíst zbylá data v pojmenované rouře a přenos ukončit. To je možné realizovat například zápisem specifické ukončovací sekvence dat do pojmenované roury. Nicméně nelze vyloučit, že se podobná sekvence vyskytne i mezi přenášenými daty a proto je tento způsob synchronizace nevhodný. Vhodnější způsob spočívá ve využití standardní zprávy s informací o ukončení přenosu. Tato zpráva by byla distribuována jako událost a adresovaný proces by po jejím přijetí vyčetl poslední zbylá data z pojmenované roury a přenos by byl dokončen. Výhodou je fakt, že tato metoda synchronizace nevyužívá žádné další prostředky operačního systému. Nevýhodou je možné riziko zvýšení odezvy ukončení přenosu. Dalším možným způsobem synchronizace je využití synchronizačního semaforu, který by byl vytvářen odesílajícím blokem. Tento semafor by byl inicializován na hodnotu 0 a při vyčítání dat by se adresovaný blok pokoušel snížit jeho hodnotu. Jakmile by odesílající blok zapsal poslední část dat do pojmenované roury, tak by zvýšil hodnotu tohoto semaforu. Adresovaný proces by uspěl ve snížení hodnoty semaforu, vyčetl by zbylá data z pojmenované roury a přenos by mohl být ukončen. Tento způsob synchronizace je nesrovnatelně spolehlivější, než zápis ukončovací sekvence do pojmenované roury a současně je rychlejší, než použití standardní zprávy. Nevýhodou může být použití prostředku operačního systému navíc, nicméně se jedná pouze o jeden semafor na jeden velkokapacitní přenos, což je snesitelné. Tento způsob tedy bude použit k synchronizaci velkokapacitních přenosů. Na následujícím obrázku je sekvenční diagram typického velkokapacitního přenosu.




**Obr. 2:** Příklad průběhu velkokapacitního přenosu mezi dvěma bloky

## 2.5 Návrh bloků

### 2.5.1 Obecný návrh

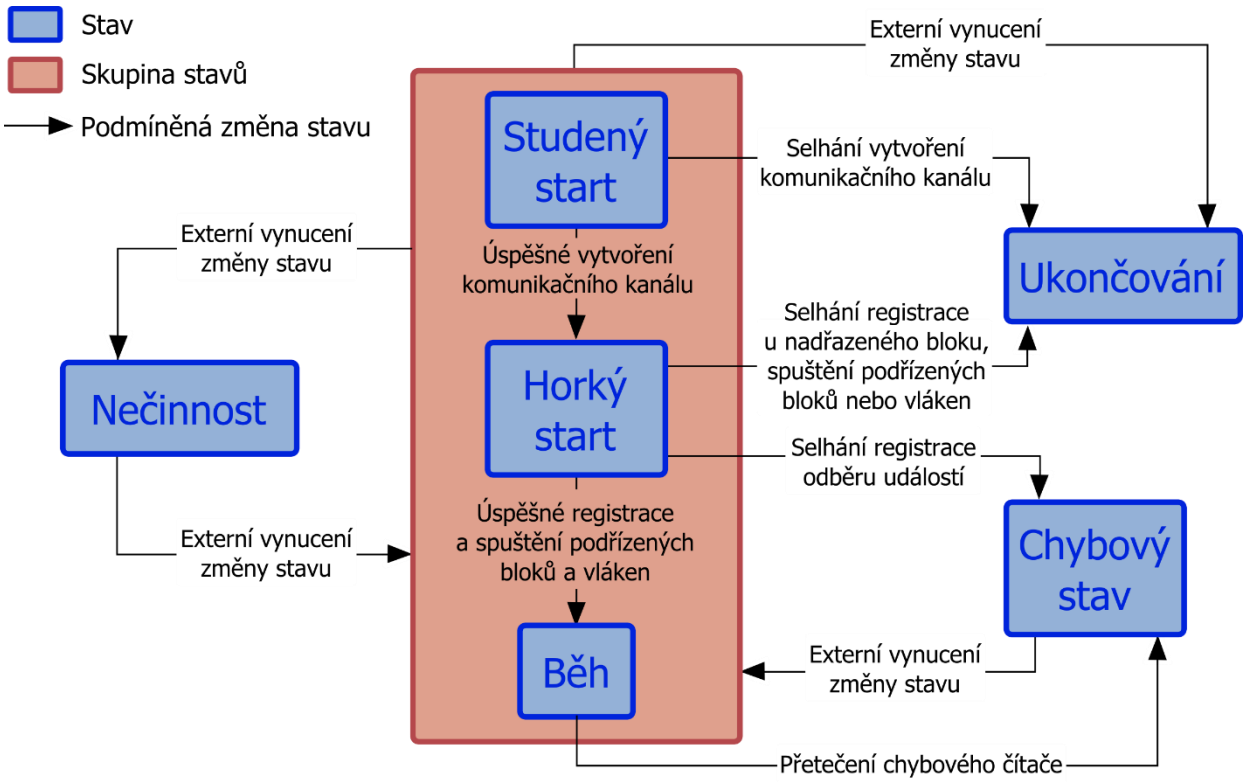
Vzhledem k dosažení vyšší efektivity operací prováděných jednotlivými bloky je vhodné bloky rozdělit do více vláken. Hlavní vlákno každého bloku bude řídit činnost celého bloku vykonáváním stavového automatu bloku a bude zajišťovat nejdůležitější z operací prováděných blokem. V případě, že daný blok je nadřazeným blokem, je vhodné činnost řízení podřízených bloků a případně i směrování komunikace mezi nimi také delegovat do samostatného vlákna. Další vlákno, případně vícero vláken, může být delegováno pro vykonávání různých operací specifických pro daný blok.

Je samozřejmě možné činnost jednoho nebo více vláken sloučit a v extrémním případě mít v rámci bloku i jediné vlákno. Takový návrh ale vyžaduje použití absolutně neblokujících operací (neboli blokujících operací s nulovým časovým limitem), aby bylo zajištěno pravidelné provádění stavového automatu bloku a odesílání stavu na výzvu softwarového watchdogu nadřazeného bloku. V takovém případě by tedy proces bloku neustále zabíral mnoho procesorového času a bylo by tedy nutné proces pravidelně uspávat na daný časový interval. Stále by však bylo potřeba zaručit, že specifické operace prováděné blokem budou natolik rychlé, že blok bude stíhat reagovat na výzvy softwarového watchdogu nadřazeného bloku. Nicméně, sloučení činnosti určitých vláken určitého bloku může mít ve specifických případech své výhody a proto takové rozhodnutí bude záležet především na implementátorovi daného bloku.

Aby bylo možné sledovat stavy jednotlivých bloků, bude se každý běžící blok vždy nacházet v jednom z následujících stavů.

- **Studený start** – v tomto stavu se bude blok nacházet typicky po svém spuštění a budou v něm prováděny základní inicializace bloku. V tomto stavu dojde k vytvoření komunikačního spojení s nadřazeným blokem.
- **Horký start** – v tomto stavu načte každý blok svou konfiguraci, alokuje prostředky potřené pro jeho chod a poté vykoná procesy registrace u nadřazeného bloku. Každý nadřazený blok v tomto stavu provede spuštění svých podřízených bloků.
- **Běh** – v tomto stavu bude blok vykonávat veškeré běžné operace. Bude vyřizovat kritickou komunikaci s nadřazeným blokem a bude řídit své podřízené bloky.
- **Nečinnost** – v tomto stavu nebude blok dělat vůbec nic. Přechod do tohoto stavu bude možný pouze externím zásahem.
- **Ukončování** – v tomto stavu blok uvolní zabrané prostředky, ukončí případné podřízené bloky a provede zrušení registrací. Poté blok ukončí svou činnost.
- **Chybový stav** – v tomto stavu se bude blok nacházet v případě opakovaného výskytu chybových situací. Blok by však měl, pokud to bude možné, nadále vykonávat běžné provozní operace.

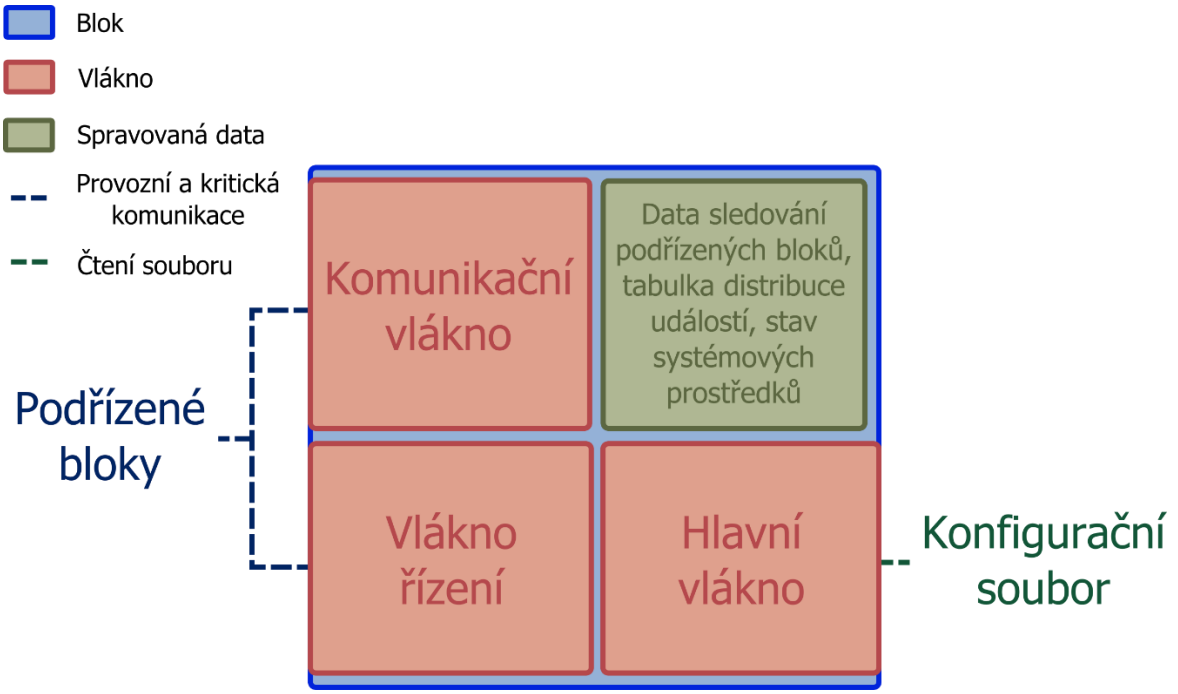
Návrh stavového automatu bloku využívajícího těchto stavů se nachází na následujícím obrázku.




**Obr. 3:** Návrh stavového automatu bloků

### 2.5.2 Jádro

Hlavním úkolem prováděným blokem jádra bude řízení jemu podřízených bloků, zajišťování spojení a směrování zpráv mezi podřízenými bloky a buzení externího watchdogu. K méně důležitým úkolům tohoto bloku pak bude patřit monitorování systémových zdrojů. Na následujícím obrázku je vidět návrh jednotlivých vláken tohoto bloku.




**Obr. 4:** Návrh struktury bloku jádra

Hlavní vlákno bude řídit činnost celého bloku vykonáváním stavového automatu bloku, bude se starat o buzení externího watchdogu a bude monitorovat využití procesoru, pamětí a datových úložišť.

Řídicí vlákno tohoto bloku bude řídit činnost všech bloků podřízených bloku jádra. To znamená, že toto vlákno bude zajišťovat spouštění a případné ukončování všech těchto bloků a bude monitorovat jejich chod.

Komunikační vlákno tohoto bloku bude zajišťovat spojení a směrování zpráv mezi ostatními bloky. Tento proces bude fungovat na základě návrhu popsaného v kapitole 2.4 Komunikace.

Blok jádra bude spravovat, navíc k položkám společným pro všechny nadřazené bloky, následující položku.

- **Stav systémových prostředků** – bude obsahovat využití procesoru, paměti a datových úložišť v okamžiku poslední kontroly a časovou značku (formát Unix timestamp) poslední kontroly prostředků.

Konfigurační soubor bloku jádra bude obsahovat, navíc k položkám společným pro všechny nadřazené bloky, následující položky.

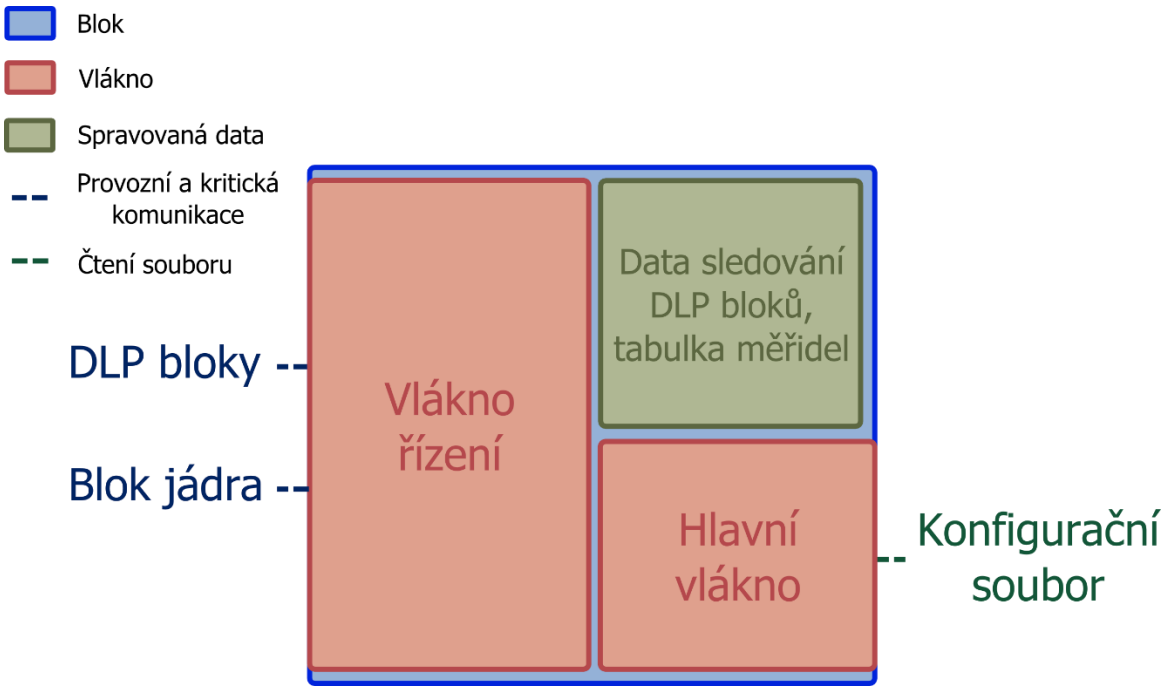
- **Perioda buzení externího watchdogu** – časový údaj ve vteřinách, který bude určovat periodu, se kterou bude jádro budit externí watchdog.
- **Perioda kontroly systémových prostředků** – časový údaj v sekundách, který bude určovat periodu, se kterou bude jádro kontrolovat stav procesoru, paměti a datových úložišť.
- **Konfigurace webového serveru** – jelikož webový server je aplikace třetí strany, není podřízený žádnému z bloků, a tedy jeho konfiguraci bude řídit blok jádra. Bude se jednat především o nastavení maximálního počtu současně připojených klientů.

Blok jádra bude nejen přesměrovávat zprávy odeslané z podřízených bloků do příslušných adresovaných bloků, ale také bude sám využívat následující zprávy provozní komunikace (navíc ke zprávám společným pro nadřazené bloky).

- **Žádost o stav systémových prostředků** – bude využívána zejména kvůli diagnostickým účelům.
- **Stav systémových prostředků** – zasílána jako odpověď na žádost o stav systémových prostředků. Bude obsahovat stejné položky, jako spravovaná položka se stavem systémových prostředků.

### 2.5.3 Řízení downlink rozhraní

Účelem bloku řízení downlink rozhraní (dále jen DLC bloku) bude řízení komunikace na sítích měřidel, jejich správa a zajištění prvotního zpracování přijatých dat a událostí z těchto síti. Vzhledem k nutnosti podpory více než jednoho typu sítě měřidel je nezbytná existence tzv. proxy bloků, které budou překládat data a události z daných sítí měřidel do interní reprezentace. Tyto proxy bloky budou podřízené bloku DLC. Na následujícím obrázku je vidět návrh rozdělení tohoto bloku na jednotlivá vlákna.




**Obr. 5:** Návrh struktury bloku řízení downlink rozhraní

Hlavní vlákno bude řídit činnost celého bloku vykonáváním stavového automatu bloku a bude obstarávat kritickou komunikace s nadřízeným blokem (tj. blokem jádra).

Vlákno řízení proxy bloků bude řídit činnost podřízených proxy bloků jednotlivých rozhraní a bude zajišťovat směrování komunikace a řízení distribuce událostí. Toto vlákno se také bude starat o správu měřidel v sítích. To bude prováděno zejména pomocí sledování aktivity daného měřidla. To využívá předpokladu, že v případě, kdy od daného měřidla často přichází nějaká data, je toto měřidlo pravděpodobně připojeno k síti a je aktivní. Pokud delší dobu od určitého měřidla žádná data nedorazila, pak je možné, že toto měřidlo již není k síti připojeno. Tento mechanismus bude realizován tak, že v okamžiku přijetí dat od určitého měřidla bude resetován indikátor aktivity tohoto měřidla a poté bude tento indikátor periodicky snižován až do stavu, kdy bude oprávněné předpokládat, že dané měřidlo již není připojené. V tom okamžiku bude dané měřidlo vyřazeno ze sledování.

Blok DLC bude, navíc k položkám spravovaným všemi nadřazenými bloky, obsahovat následující položky.

- **Tabulka měřidel** – bude sloužit ke správě měřidel a každý záznam se bude skládat z následujících položek.
    - **Interní ID měřidla** – kladné celé číslo v rozsahu 1 až 2048 (číslo 0 je vyhrazeno pro koncentrátor samotný).
    - **Interní ID nadřazeného měřidla** – bude využito v případě, že síť, na které se dané měřidlo nachází, používá stromovou topologii. Pak tato položka umožňuje rekonstrukci takové topologie. Pokud je nadřazeným měřidlem daného měřidla přímo koncentrátor, je toto číslo rovné nule. To je případ všech měřidel v sítích bez stromové topologie.

- **Nativní ID** – položka velikosti čtyř bajtů, která v typickém případě bude představovat adresu daného měřidla v příslušné síti. Bude využita při zasílání požadavků na konkrétní operace s daným měřidlem do odpovídajícího proxy bloku.
- **Příslušnost k proxy bloku** – bude označovat proxy blok spravující síť, ve které se dané měřidlo nachází.
- **Sériové číslo** – patnáctiznakový textový řetězec jednoznačně identifikující dané měřidlo.
- **Aktuální stav aktivity a časová značka poslední změny** – bude představovat hodnotu indikátoru aktivity daného měřidla a časovou značku poslední změny úrovně aktivity ve formátu Unix timestamp. V případě, že měřidlo začne být považováno za neaktivní, bude vyřazeno z tabulky měřidel.

Konfigurační soubor bloku DLC bude obsahovat, navíc k položkám společným pro všechny podřízené a nadřazené bloky, následující položky.

- **Maximální počty měřidel** – bude mít formu jedné hodnoty pro každý spravovaný proxy blok a hodnoty celkového maximálního počtu měřidel.
- **Nastavení sledování aktivity měřidel** – bude se jednat o nastavení periody změny indikátoru aktivity měřidla, po jejímž uplynutí dojde k automatickému snížení stavu aktivity daného měřidla.
- **Chování při dosažení maximálního počtu měřidel** – bude určovat, zda při připojení nového měřidla do sítě v okamžiku, kdy je buď dosažen maximální počet měřidel na dané síti, nebo je dosažen celkový maximální počet měřidel, bude toto připojené měřidlo odregistrováno ze sítě nebo zda v takové situaci dojde k odregistrování nejméně aktivního měřidla.

Blok DLC bude využívat následující zprávy provozní komunikace.

- **Požadavek k odeslání dat** – bude zasílán jinými bloky do DLC bloku a také DLC blokem samotným do proxy bloků. Bude obsahovat data k odeslání a ID adresovaného měřidla. Nativní ID v případě, že požadavek byl DLC blokem odeslán a interní ID v případě, že požadavek byl DLC blokem přijat. V případě, že je DLC blokem tato zpráva přijata, je vytvořena stejná zpráva, interní ID měřidla je nahrazeno nativním ID a zpráva je přeposlána do příslušného proxy bloku.
- **Potvrzení odeslání dat** – bude zasíláno DLC blokem do bloků, ze kterých byl přijat požadavek k odeslání dat poté, co odeslání dat bude potvrzeno příslušným proxy blokem.
- **Požadavek na tabulku měřidel** – bude přijímán DLC blokem a používán zejména k účelům diagnostiky a sledování stavu sítí.
- **Tabulka měřidel** – bude odesílán jako odpověď na požadavek na tabulku měřidel. V závislosti na počtu připojených měřidel tato zpráva může být žádajícímu bloku odeslána formou velkokapacitního přenosu.

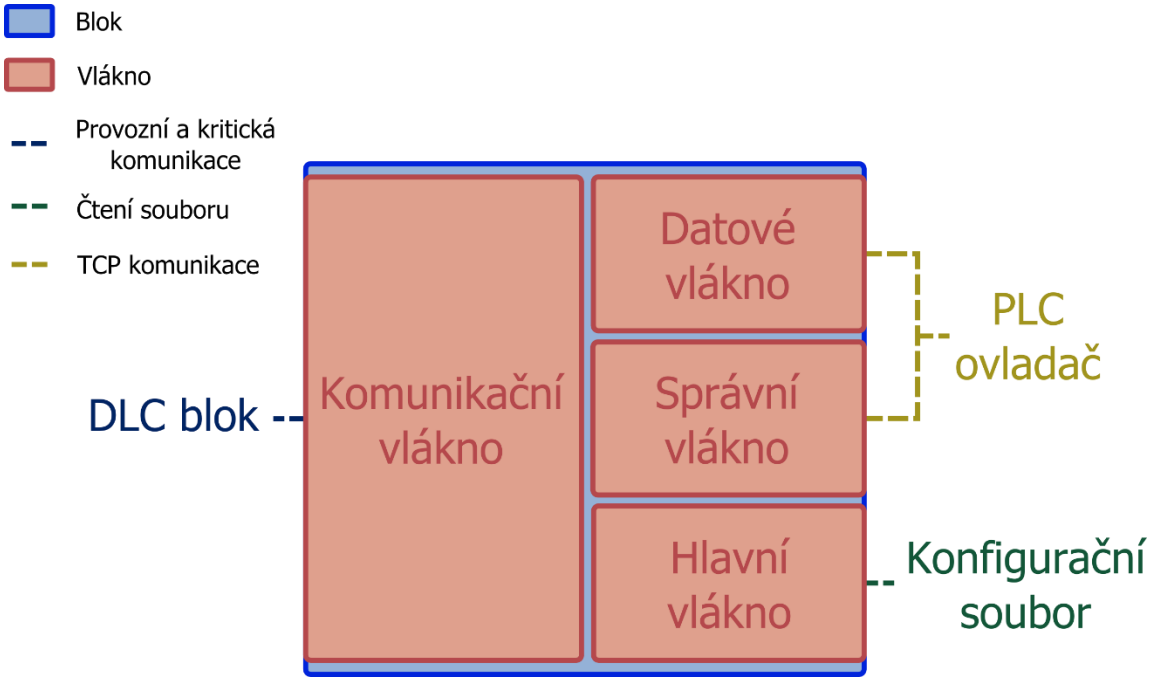
Blok DLC bude zdrojem následujících událostí.

- **Dosažení maximálního počtu měřidel** – notifikace o této události bude odesílána v případě, že bude překročen některý z nastavených maximálních počtů připojených měřidel.
- **Příjem dat** – notifikace o této události bude odesílána v okamžiku přijetí nových dat z nějakého proxy bloku.

- **Změna tabulky měřidel** – notifikace o této události bude DLC blokem odeslána v okamžiku, kdy některé z měřidel v tabulce změní svůj stav na aktivní nebo neaktivní, nebo pokud do tabulky přibude nové měřidlo.

#### *2.5.3.1 Downlink proxy*

Úkolem downlink proxy bloku (dále jen DLP bloku) bude poskytovat DLC bloku rozhraní k řízení a komunikaci v rámci jednoho typu sítě měřidel. Existuje řada variant sítí měřidel, jako jsou například sítě PLC nebo Wireless M-Bus, a struktura DLP bloku bude u každého typu sítě různá. Pokud daná síť nebude podporovat standard DLMS/COSEM, bude úkolem DLP bloku této sítě překládat data z měřidel na této síti do tohoto standardu. Nicméně zatím je ze strany zadavatele požadavek pouze na podporu sítě PLC využívající standard PRIME (viz. [6]) nebo G3, která standard DLMS/COSEM podporuje. Na následujícím obrázku je návrh DLP bloku PLC sítě (dále jen PLC-DLP).




**Obr. 6:** Návrh struktury proxy bloku PLC sítě využívající standard PRIME nebo G3

K ovládání PLC modemu bude využit ovladač PRIME-DC (v případě standardu PRIME) nebo G3-DC (v případě standardu G3) poskytovaný společností Texas Instruments. Tento ovladač je prostá aplikace běžící v uživatelském režimu operačního systému a s PLC modemem na vývojové desce komunikuje prostřednictvím protokolu UART. Jako rozhraní pro aplikace je u tohoto ovladače použit TCP server komunikující na třech portech. První port slouží pro datovou komunikaci, druhý pro řídicí komunikaci a třetí port poskytuje testovací rozhraní. Blok PLC-DLP tedy bude obsahovat jedno vlákno pro první a jedno pro druhý port a bude překládat blokem přijaté požadavky do formy vyžadované tímto ovladačem.

Datové vlákno bude provádět příjem a prvotní zpracování dat z PLC sítě. Tato data nebude nijak interpretovat a bude je dále pomocí komunikačního vlákna zasílat ve formě notifikace o události do nadřazeného bloku (tj. DLC bloku).

Správní vlákno bude sloužit k řízení PLC sítě a příjmu v ní se vyskytujících událostí. Tyto události budou blokem vyhodnocovány, a buď budou na místě zpracovány, nebo upozornění na jejich výskyt budou odesílány do nadřazeného DLC bloku.

Hlavní vlákno bude řídit činnost celého bloku prostřednictvím stavového automatu a bude zajišťovat kritickou komunikaci s nadřazeným blokem (tj. DLC blokem).

Komunikační vlákno bude vyřizovat veškerou provozní komunikaci a na základě požadavků směrovat data do správného vlákna (tj. datového či správního).

Blok PLC-DLP nebude spravovat žádné informace navíc a jeho konfigurační soubor bude obsahovat pouze standardní položky.

Blok PLC-DLP bude využívat následující zprávy provozní komunikace.

- **Požadavek k odeslání dat** – bude zasílán DLC blokem a bude obsahovat data k odeslání a nativní ID adresovaného měřidla. Nativní ID měřidel se v případě PLC sítě založené na standardu PRIME či G3 liší podle použité konvergenční podvrstvy. Nejpoužívanější konvergenční podvrstva standardů PRIME je založena na protokolu IEC61334-4-32 a tudíž nativní ID v případě PLC-DLP bloku bude adresa uzlu tohoto protokolu.
- **Potvrzení odeslání dat** – bude zasíláno jako odpověď na požadavek k odeslání dat v okamžiku, kdy odeslání dat potvrdí, nebo nahlásí jeho selhání, příslušný ovladač. Tato zpráva bude obsahovat informaci o úspěchu či příčině neúspěchu odeslání dat.
- **Požadavek provedení operace správy** – bude zasílán DLC blokem a bude specifikovat operace, které je nad sítí třeba provést. Například se může jednak o požadavek k odregistrování měřidla ze sítě.
- **Potvrzení provedení operace správy** – bude zasíláno jako odpověď na požadavek k provedení operace správy. Tato zpráva bude obsahovat informaci o úspěchu či příčině selhání operace správy.

Tento blok bude zdrojem následujících událostí.

- **Příjem dat** – notifikace o této události bude do nadřazeného DLC bloku odesílána v okamžiku, kdy ovladač ohlásí přijetí nových dat.
- **Připojení nového měřidla** – notifikace o této události bude zasílána do nadřazeného DLC bloku v okamžiku, kdy ovladač ohlásí úspěšnou registraci nového měřidla do sítě.

### 2.5.4 Řízení uplink rozhraní

Účelem bloku řízení uplink rozhraní (dále jen ULC bloku) bude vykonávání obdobných úkonů, jaké bude provádět DLC blok, s tím rozdílem, že ULC blok bude tyto úkony provádět nad zákaznickými sítěmi. Stejně jako v případě řízení downlink rozhraní budou jednotlivé sítě spravovány různými instancemi ULP bloků. Ty budou určeny k řízení pouze jediného konkrétního typu sítě.

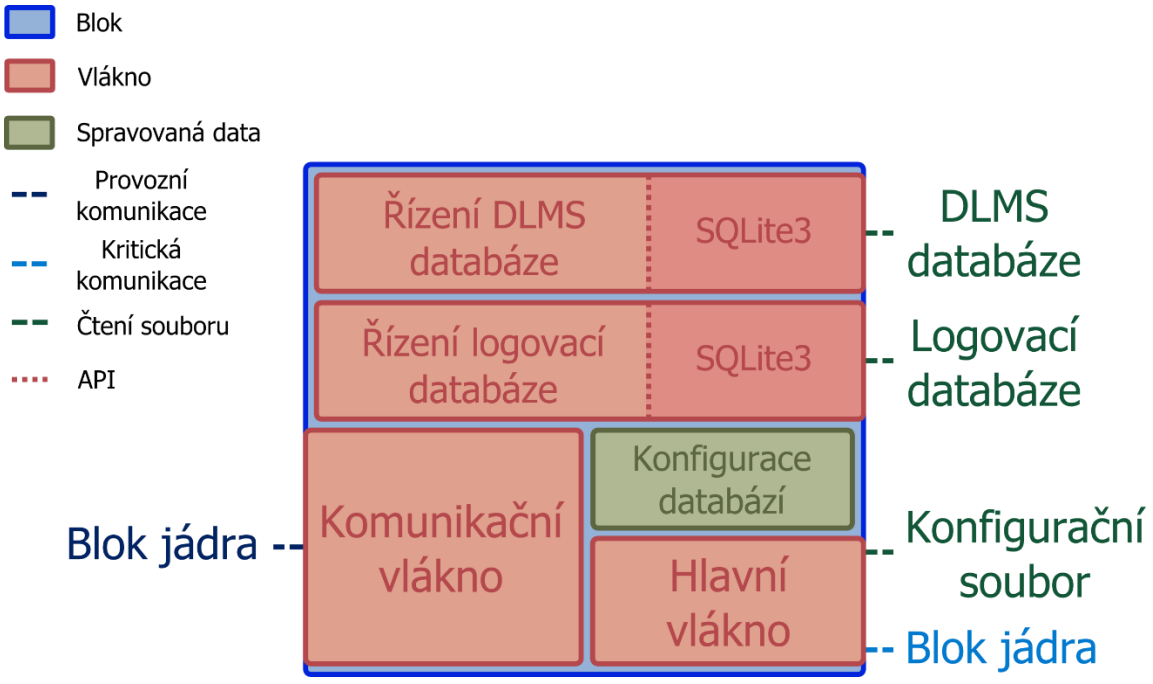
Existuje mnoho rozmanitých variant realizací spojení se zákaznickými sítěmi. V drtivé většině případů bude zákaznická síť založena na protokolovém zásobníku TCP/IP a bude záležet pouze na konkrétní použité aplikační vrstvě. Tou může být například protokol pro výměnu objektů SOAP, případně CORBA,

nebo se může jednat o komunikaci formou nahrávání souborů s daty na FTP či TFTP server, případně lze prostřednictvím UDP přenosu zasílat do zákaznických sítí přímo DLMS/COSEM data. V době vzniku této diplomové práce nebyly požadavky na uplink rozhraní známé a proto nebyla navržena struktura ULC bloku, ani žádného z ULP bloků.

### 2.5.5 Řízení databáze

Úkolem bloku řízení databáze bude správa databázového systému koncentrátoru a poskytování rozhraní pro přístup do něj ostatním blokům. Klíčovou otázkou u tohoto bloku je volba vhodného databázového systému. Vzhledem k relativní jednoduchosti a rozšířenosti bylo určeno, že databázový systém bude založen na relačních databázích, bude využívat dotazovací a deklarační jazyk SQL a bude zdarma dostupný ve formě otevřeného software, ideálně s licencí typu BSD, MIT, či jejich ekvivalentem. Databázových systémů, které tyto požadavky splňují je řada. Mezi známější z nich patří například databázové systémy MySQL a PostgreSQL. Tyto databázové systémy jsou často využívané ve velkých serverech, a tedy jsou schopné zvládat naráz velká množství dotazů. Pro účely datového koncentrátoru se však jedná o až příliš komplexní řešení, které by zabíralo zbytečně mnoho systémových prostředků. Vhodnější volbou by tedy byl databázový systém určený pro vestavěná zařízení. Příkladem takového databázového systému je SQLite3. Ten, při volbě vhodného návrhu databáze, dokáže bez problému a větších latencí zpracovávat dotazy nad databázemi, jejichž velikost je řádově v gigabajtech až dokonce v desítkách gigabajtů. Na základě těchto faktů bude tedy databázový systém SQLite3 použit jako interní databázový systém koncentrátoru.

Na rozdíl od systémů MySQL a PostgeSQL není realizován jako samostatně běžící aplikace, ale knihovna s definovaným API pro přístup do databází. Z tohoto důvodu bude dalším úkolem bloku řízení databáze příchozí dotazy serializovat. Na následujícím obrázku je návrh jeho struktury.




**Obr. 7:** Návrh struktury bloku řízení databáze

V databázovém systému budou využívány dvě databáze. První databáze bude sloužit pro ukládání logovacích dat z ostatních bloků (dále jen logovací databáze). Druhá databáze bude použita pro ukládání dat z měřidel a její struktura bude respektovat strukturu COSEM objektů (dále jen DLMS databáze). Jak je vidět z obrázku návrhu, blok řízení databáze bude obsahovat jedno vlákno pro každou databázi. Každé takové vlákno řízení databáze bude vyřizovat dotazy příchozí do dané databáze. Aby nedocházelo k hromadění dat v příslušné databázi, bude vlákno řízení databáze pravidelně odstraňovat staré záznamy z databáze. Také, z bezpečnostních důvodů, bude monitorovat velikost databázového souboru, aby nedošlo k jeho přílišnému nárůstu.

Hlavní vlákno bude řídit činnost celého bloku vykonáváním stavového automatu bloku a bude obstarávat kritickou komunikace s nadřízeným blokem (tj. blokem jádra).

Komunikační vlákno tohoto bloku bude zajišťovat překlad příchozích zpráv provozní komunikace do interní reprezentace a jejich přesměrování do příslušných databázových vláken.

Je vidět, že tato struktura je snadno rozšiřitelná, neboť v případě, že v budoucnu vznikne požadavek na další databázi (například databázi uživatelů), stačí přidat jediné vlákno řídící tuto novou databázi a příslušně upravit komunikační vlákno.

Blok řízení databáze bude spravovat, navíc k položkám společným pro všechny podřízené bloky, následující položku.

- **Tabulka databází** – u každé databáze bude specifikována její aktivní konfigurace a navíc aktuální velikost databázového souboru.

Konfigurační soubor bloku řízení databáze bude obsahovat, navíc k položkám společným pro všechny podřízené bloky, následující položku.

- **Nastavení databází** – u každé databáze bude specifikovat její typ, cestu k databázovému souboru, velikost časového okna databáze, které bude určovat, jak staré záznamy mohou být v databázi uloženy a maximální velikost databázového souboru.

Aby tento blok poskytoval rozhraní k přístupu do databází pro ostatní bloky, budou využity následující zprávy provozní komunikace.

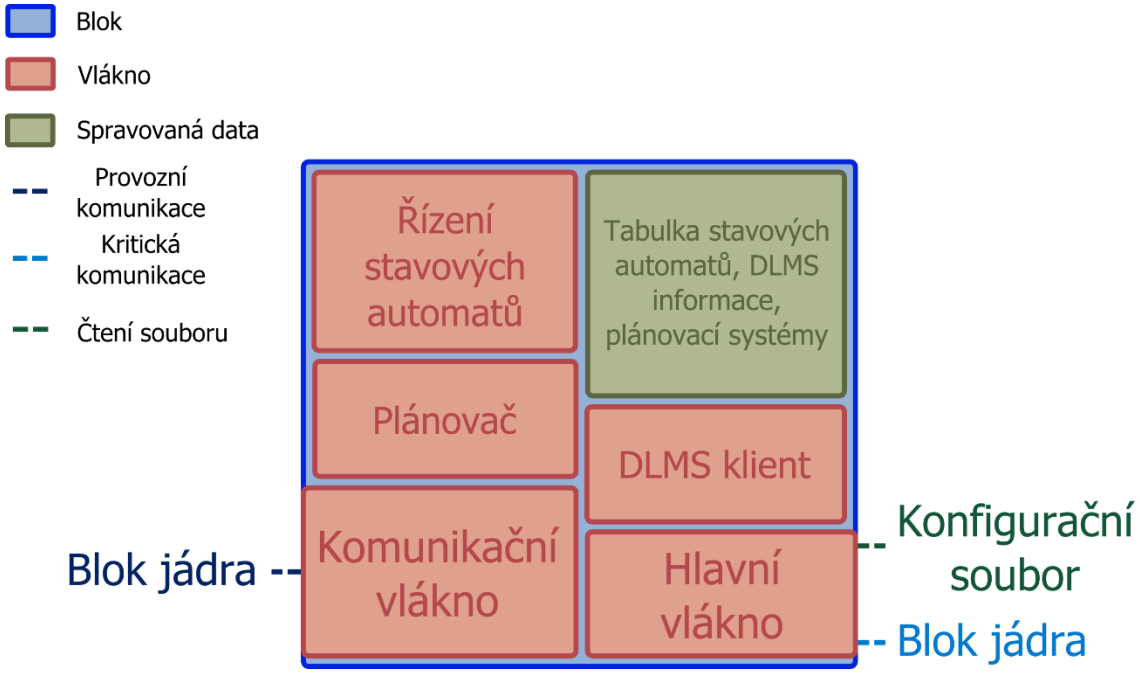
- **Žádost o vykonání SQL dotazu** – bude obsahovat SQL dotaz k vykonání se specifikací databáze, nad kterou má být vykonán.
- **Žádost o zápis logu** – bude obsahovat informaci identifikující zdrojový blok logu, závažnost logu, titulek a text logu.
- **Výsledek operace** – bude zasílána jako odpověď na předchozí uvedené žádosti. Bude obsahovat informaci o úspěchu operace a data z databáze, nebo chybový kód identifikující příčinu jejího neúspěchu.
- **Žádost o tabulku databází** – bude využívána zejména k diagnostickým účelům.
- **Tabulka databází** – bude odesílána jako odpověď na žádost o tabulku databází. Bude obsahovat stejné položky, jako spravovaná položka tabulka databází.

Blok řízení databáze bude zdrojem následující události.

- **Dosažení maximální velikosti databázového souboru** – notifikace o této události bude zasílána v okamžiku, kdy některé z vláken řízení databáze detekuje dosažení maximální velikosti příslušného databázového souboru.

### 2.5.6 Řízení chování

Blok řízení chování bude zdrojem veškerých autonomních operací vykonávaných koncentrátorem. Jeho účelem bude sledování veškerých událostí vznikajících uvnitř koncentrátoru, jejich vyhodnocování, příjem dat z měřidel a zjištění jejich významu. Návrh jeho struktury je na následujícím obrázku.




**Obr. 8:** Návrh struktury bloku řízení chování

Je žádoucí, aby na rozdíl od konkurenčních řešení byla autonomie koncentrátoru snadno konfigurovatelná. Bude tedy zajištěna prostřednictvím systému konfigurovatelných stavových automatů, které na základě vzniklých událostí budou provádět dané akce. Tyto automaty budou popsány prostřednictvím XML souboru vhodného formátu. Je žádoucí, aby tyto automaty bylo snadné verifikovat. Proto se jako vhodný kandidát na popis automatů jeví jazyk systému UPPAAL (viz. [5]), což je systém pro návrh, simulaci a verifikaci časovaných stavových automatů. Detaily o podpoře tohoto jazyku jsou uvedeny v kapitole 3.2.7 Řadič stavových automatů. Chod těchto stavových automatů bude řídit vlákno řízení běhu automatů. V případě, že některý automat bude vyžadovat provedení určité akce, bude odpovídající požadavek předán do vlákna plánovače.

Vlákno DLMS klienta bude zajišťovat podporu standardu DLMS/COSEM. Bude přebírat veškerá data přijatá z měřidel, tato data bude analyzovat a na jejich základě bude provádět zápis dat do databáze, aktualizaci DLMS informací a případné generování notifikací o událostech pro vlákno řízení běhu automatů. Vlákno bude využívat klientskou aplikaci standardu DLMS/COSEM dodanou zadavatelem.

Jak již bylo řečeno, do vlákna plánovače budou vláknem řízení automatů předávány požadavky na vykonání určitých operací. Tyto akce budou využívat různé zdroje (tj. rozhraní k sítím elektroměrů, konzolové rozhraní a podobné) a budou mít různou prioritu. Úkolem vlákna plánovače bude zajišťovat,

aby na některém ze zdrojů nedošlo ke konfliktu a aby byly vyžádané akce prováděny s ohledem na jejich různou prioritu. Důležitá je volba vhodného rozvrhovacího algoritmu. Akce, jejíž provádění bude žádáno nejčastěji, bude vyčtení dat z měřidel. Tato akce je na většině běžně používaných sítí měřidel velmi pomalá a vyčtení jednoho měřidla může trvat i několik minut. Z tohoto důvodu je dostačující použití jednoduchého algoritmu, ale je nezbytné zohlednit existencí více zdrojů (tj. více sítí měřidel v případě vyčítání měřidel) a priorit žádaných akcí. Z tohoto důvodu bude vlákno plánovače využívat systém rozvrhování, kde každý zdroj bude mít přiřazený jeden systém prioritních front. Ten bude obsahovat jednu frontu pro každou prioritu a tyto fronty budou vzájemně propojené dle posloupnosti priorit. Jednotlivé fronty budou obsluhovány systémem FCFS (tj. tyto fronty budou FIFO buffery). Pro zlepšení efektivity rozvrhování je možné využít tzv. zrání akcí, kde je procesům, které jsou v rozvrhovacím systému již delší dobu, s postupem času zvyšována priorita.

Konfigurační soubor bloku řízení chování bude, navíc k položkám běžným pro všechny bloky, obsahovat následující položky.

- **Cesta k definičnímu souboru stavových automatů** – bude určovat, z jakého souboru bude blok načítat definice stavových automatů.
- **Parametry plánovače** – tato položka se může skládat z více položek v závislosti na použitém rozvrhovacím algoritmu.
- **Konfigurace DLMS klienta** – tato položka se může skládat z více položek v závislosti na použitém DLMS klientovi. V typickém případě se bude jednat o seznam podporovaných COSEM objektů s jejich OBIS kódy. Právě změnou takové položky v konfiguraci by šlo snadno upravit podporu standardu DLMS/COSEM podle zadané specifikace bez nutnosti změn ve zdrojovém kódu.

Blok řízení chování bude spravovat, navíc k položkám společným pro všechny bloky, následující položky.

- **Tabulka stavových automatů** – přehled jednotlivých automatů a informací o nich. Pro každý stavový automat bude obsahovat následující informace.
    - **Identifikátor** – textový řetězec jednoznačné identifikující daný stavový automat.
    - **Aktuální stav** – bude označovat aktuální stav, ve kterém se daný stavový automat nachází. Tato položka v sobě zahrnuje i časovou značku (tj. Unix timestamp) vstupu do něj.
- **DLMS informace** – data použitá DLMS klientem pro interpretaci přijatých dat. Obsah této položky bude záviset na použitém DLMS klientovi, nicméně lze předpokládat, že u každého DLMS klienta bude potřeba udržovat seznam podporovaných COSEM objektů s jejich OBIS kódy a seznam měřidel, ve kterém budou pro každé měřidlo udržovány následující položky.
    - **Interní ID** – položka bude totožná s interním ID v tabulce měřidel DLC bloku.
    - **Proxy** – tato položka bude určovat, ke které síti (tj. ke kterému proxy bloku) dané měřidlo náleží.
    - **Stav měřidla** – bude určovat, kdy naposledy a jestli úspěšně bylo měřidlo vyčteno.

- **Rozvrhovací systémy** – bude se jednat o výše popsané systémy prioritních FCFS front, které bude vlákno plánovače využívat k rozvrhování vyžádaných akci.

Blok řízení chování bude využívat následující zprávy provozní komunikace.

- **Žádost o tabulku stavových automatů** – bude zasílána jinými bloky (typicky bloky webového a konzolového rozhraní) a bude sloužit diagnostickým účelům.
- **Tabulka stavových automatů** – bude zasílána blokem řízení chování jako odpověď na žádost o tabulku elektroměrů. Bude obsahovat cestu k definičnímu souboru stavových automatů a následující položku pro každý stavový automat.
    - **Aktuální stav** – tato položka bude odpovídat stejnojmenné položce z tabulky stavových automatů.
- **Požadavek na operaci se stavovým automatem** – tato zpráva bude zasílána ostatními bloky a bude sloužit zejména k diagnostickým účelům a bude sloužit k vynucení změny stavu specifického automatu, k ukončení a zahájení jeho chodu nebo vynucení opětovného načtení stavových automatů z definičního souboru.
- **Výsledek operace se stavovým automatem** – tato zpráva bude zasílána blokem řízení chování jako odpověď na požadavek na operaci se stavovým automatem a bude obsahovat informaci o úspěchu, případně příčiny neúspěchu dané operace. Příčinou neúspěchu žádané operace může být například nepodporovaná operace, nenalezení specifikovaného stavového automatu, neexistence vynuceného stavu či neúspěch načtení automatů z definičního souboru.

Blok řízení chování bude také přijímat notifikace o celé řadě událostí, které mohou vznikat v jiných blocích, nebo dokonce v bloku řízení chování samotném a bude samozřejmě využívat různé zprávy ostatních bloků za účelem jejich řízení v rámci prováděných akcí.

### 2.5.7 Konzolové a webové rozhraní

Konzolové a webové rozhraní datového koncentrátoru budou poskytovat nástroje ke konfiguraci, údržbě, diagnostice a sledování jeho chodu. Webové rozhraní bude poskytovat základní verze těchto nástrojů, a pomocí něj tedy bude možné monitorovat stav jednotlivých bloků, zobrazovat sítě měřidel, sledovat stav databází, monitorovat autonomní řízení a prohlížet logovaná data. Také bude umožňovat provádění změn v nastavení základních parametrů. Konzolové rozhraní pak poskytne, v textové formě, plnohodnotné verze těchto nástrojů.

Z hlediska webového rozhraní je důležitou otázkou volba vhodného webového serveru. Webové rozhraní koncentrátoru bude současně využívat pouze několik uživatelů, a proto není vhodné použití plnohodnotného serveru, jako je například Apache. Vhodnější je použití webového serveru pro vestavěná zařízení, nicméně je nutné, aby daný webový server podporoval zabezpečená připojení, zvládal současné připojení více klientů najednou a disponoval CGI rozhraním pro generování dynamických webových stránek. Použit tedy bude webový server lighttpd, který tyto požadavky splňuje.

Z hlediska blokové struktury jsou si bloky konzolového a webového rozhraní velmi podobné. Oba bloky budou podřízené jádru, nicméně nebudou jím sledované. Bloky webového rozhraní budou spouštěny

prostřednictvím CGI webového serveru lighttpd a budou sloužit ke generování dynamických webových stránek. Blok konzole bude spouštěn buď přímo uživatelem, nebo prostřednictvím bloku řízení chování. Za účelem funkčnosti nástrojů poskytovaných oběma bloky budou oba využívat veškeré zprávy provozní komunikace, které využívají ostatní bloky.

## 2.6 Vývojové prostředky

Cílem této práce je implementovat nástroje pro meziblokovou komunikaci a blok řízení chování prostřednictvím stavových automatů. Jelikož nástroje pro meziblokovou komunikaci budou používány všemi bloky, je žádoucí, aby byly implementovány v jazyce C. Blok řízení chování je možné implementovat v jazyce C++ s využitím standardu C++11.

Jak bylo řečeno v úvodu, k vývoji byla zvolena vývojová deska Smart Data Conectrator Evaluation Module od společnosti Texas Instruments. Tato deska je vybavena procesorem Sitara AM3359 a je k ní dostupné SDK. Nejdůležitějšími součástmi tohoto SDK jsou Linuxová distribuce Arago ve verzi 2013, dále kompilátory *arm-linux-gnueabihf-gcc* a *arm-linux-gnueabihf-g++* od společnosti Linaro, varianty běžně používaných knihoven pro architekturu ARMv7 a nezbytné hlavičkové soubory. Toto SDK bude využito pro vývoj celého softwarového vybavení koncentrátoru.

Jelikož blok řízení chování i nástroje pro meziblokovou komunikaci budou využívat standardní prostředky dostupné v každé distribuci operačního systému Linux a nejsou závislé na konkrétním hardwaru vývojové desky, lze k jeho vývoji využít i běžný stolní počítač. K vývoji těchto součástí tedy bude využita také Linuxová distribuce Kubuntu 14.04 „Trusty Tahr“ v 64bitové verzi a kompilátory *clang* a *clang++*. Jako IDE bylo zvoleno prostředí NetBeans 8.0.

# 3 Syntéza

## 3.1 Projektové řízení

V rámci projektu vývoje datového koncentrátoru CAM3600 jsem byl pověřen vedením týmu vývoje softwarového vybavení a proto bylo nutné vybrat vhodné prostředky týmové kooperace a zajistit, že každý člen týmu bude mít vždy k dispozici aktuální verzi návrhu softwarového vybavení koncentrátoru. To vyžaduje zejména volbu vhodného systému pro správu verzí.

Jako systém pro správu verzí byl zvolen distribuovaný systém git. Celkem byly vytvořeny dva repositáře tohoto systému. Jeden pro zdrojové kódy a dokumentaci softwarového vybavení a druhý pro zdrojové kódy a dokumentaci podpůrných, zejména testovacích, aplikací.

Každý člen týmu dostal za úkol implementovat některé z bloků uvedené v kapitole 2.5 Návrh bloků. Jelikož bude každý blok, s výjimkou bloků konzole a webového rozhraní, spouštěn jako démon, bude jeho název obsahovat písmeno „d“ na konci názvu. Blok jádra se tedy nazývá *cored*, blok řízení chování je označován jako *controld* a podobně. Aby bloky využívající služby ostatních bloků mohly s těmito bloky komunikovat, bylo určeno, že každý implementátor vytvoří k implementovanému bloku tzv. API soubor. Bude se jednat o hlavičkový soubor s názvem ve tvaru *<Název bloku>_api.h* (tedy například *cored_api.h*), který by měl obsahovat tvary veškerých zpráv provozní komunikace, které je daný blok schopen přijímat a zpracovávat. Ostatní bloky pak budou tento API soubor používat k sestavení zpráv určených pro daný blok. Také byl vytvořen soubor *common_api.h*, který obsahuje obecné zprávy používané všemi bloky. Jedná se tedy především o zprávy kritické komunikace, jako jsou například registrační zprávy, notifikace o událostech a zprávy softwarového watchdogu. Dále jsou v něm definovány možné stavy bloku uvedené v kapitole 2.3.1 Obecná struktura formou výčtového datového typu *T_COMM_BLK_STATUS*.

Nakonec byla dohodnuta následující adresářová struktura repositáře softwarového vybavení.

- ***dev_support*** – adresář obsahující podpůrné dokumenty k vývoji. Obsahuje například obraz souborového systému použité Linuxové distribuce, příklady zdrojových kódu a podobně.
- ***doc*** – adresář obsahující dokumentaci softwarového vybavení datového koncentrátoru CAM3600. V současné době obsahuje pouze dokument s jeho návrhem.
- ***src*** – adresář obsahující zdrojové kódy softwarového vybavení. Obsahuje následující podadresáře.
    - ***_share*** – obsahuje soubory využívané více bloky. Obsahuje následující podadresáře.
        - ***api*** – adresář pro API soubory bloků.
        - ***lib*** – adresář pro sdílené knihovny.
    - ***Adresáře jednotlivých bloků*** – adresáře s názvem ve tvaru *<Název bloku>* (například tedy *cored* v případě adresáře bloku jádra).

Každý implementátor by měl zasahovat pouze do souborů v adresářích implementovaných bloků a do API souborů těchto bloků. V případě nutnosti zásahu do zdrojových kódů či API souboru jiného bloku by mělo na prvním místě dojít ke kontaktování implementátora daného bloku. Tato opatření zajistila bezproblémovou kooperaci týmu při vývoji.

## 3.2 Knihovna pro meziblokovou komunikaci

Aby jednotlivé bloky přistupovaly k meziblokové komunikaci jednotně, bylo třeba zajistit vhodné prostředky, které by jednotný přístup ke komunikaci umožňovaly všem blokům. Z tohoto důvodu byla v rámci této diplomové práce implementována knihovna pro meziblokovou komunikaci nazvaná *libipc*.

Důležitá byla volba vhodného typu knihovny. V operačním systému Linux existují tři typy knihoven (viz. [3]). Statické knihovny jsou napevno linkovány do programů při kompilaci a při změně v knihovně je tak nutná rekompilace celého programu. Jejich výhoda spočívá zejména v rychlejší kompilaci, jelikož sdílené části programu nejsou opakovaně kompilovány v každém programu využívajícím knihovnu. Dalším typem jsou pak sdílené knihovny, tzv. shared objects. Při kompilaci programu využívajícího tento typ knihovny se knihovna nestává trvalou součástí programu a místo toho je knihovna načítána při každém spuštění programu. To umožňuje snadnou výměnu knihovny, neboť při její změně a současném zachování API stačí rekompilovat pouze knihovnu samotnou. Posledním typem knihoven jsou tzv. dynamické knihovny. Ty jsou podobné sdíleným, ale jsou v programech načítány explicitně prostřednictvím dodaných funkcí. Jelikož komunikační nástroje budou v blocích často využívané a díky žádoucím výhodám sdílených a dynamických knihoven, bude knihovna *libipc* implementována jako sdílená.

### 3.2.1 Nižší úroveň a využití prostředků

Nižší úroveň knihovny *libipc* se skládá z tzv. nízko-úrovňových wrapperů. Tyto wrappery pouze zajišťují rozhraní pro korektní použití prostředků meziprocesní komunikace operačního systému Linux a pro každý takový prostředek využívaný knihovnou *libipc* v jejím rámci existuje jeden wrapper. Tyto wrappery usnadňují, oproti použití přímých funkcí z knihoven operačního systému (viz. [4]), manipulaci s daným prostředkem meziprocesní komunikace a poskytují také kontrolu správnosti vstupních parametrů (tzv. sanity check) prostřednictvím tzv. assertion testů. Knihovna tedy obsahuje následující soubory s wrappery.

- ***msgq.c, msgq.h*** – nízko-úrovňový wrapper pro komunikaci prostřednictvím front zpráv. Obsahuje funkce pro získání deskriptoru fronty z klíče, odesílání a příjem dat z fronty a její smazání.
- ***sems.c, sems.h*** – nízko-úrovňový wrapper pro práci s tzv. sadami semaforů. V operačním systému Linux lze totiž semafory sdružovat do tzv. sad a navíc lze provádět více operací s jedním semaforem najednou. Tento wrapper tedy obsahuje funkce pro získání deskriptoru sady z klíče, vykonání posloupnosti specifických operací nad semafory v sadě a zrušení sady.
- ***rtsig.c, rtsig.h*** – nízko-úrovňový wrapper pro práci s RT signály. Obsahuje funkci pro navázání signálu na signal handler a funkci pro odeslání RT signálu s parametrem specifickému procesu.
- ***npipe.c, npipe.h*** – nízko-úrovňový wrapper pro práci s pojmenovanými rourami. Obsahuje funkce na vytvoření pojmenované roury s daným umístěním virtuálního souboru a současného získání deskriptoru tohoto virtuálního souboru, smazání pojmenované roury a jejího virtuálního souboru, otevření a uzavření tohoto souboru a čtení a zápis dat z pojmenované roury.

S frontami zpráv, sadami semaforů a pojmenovanými rourami se v operačním systému Linux manipuluje prostřednictvím tzv. deskriptorů. Deskriptor je číslo (datový typ *int*), které jednoznačně

identifikuje daný objekt v systému. V případě pojmenované roury se jedná o deskriptor jeho virtuálního souboru, kdežto v případě front zpráv a sad semaforů se jedná o deskriptor fronty nebo semaforu. K získání deskriptoru fronty zpráv nebo sady semaforů je nutné použít tzv. klíč (datový typ *key_t*). Tento klíč lze při vytváření front například generovat prostřednictvím funkce *ftok(...)* (viz. [2]) nebo lze použít konstantní hodnotu. Pokud dva běžící programy zažádají například o deskriptor fronty zpráv pomocí stejného klíče, oba programy dostanou deskriptor odpovídající jedné a té samé frontě.

Tyto klíče je třeba znát před samotným přístupem k danému objektu fronty zpráv či sady semaforů, a proto přístup, kdy jsou třeba klíče k získání komunikačních prostředků, bude zachován i na vyšší úrovni a API knihovny *libipc* (detaily jsou uvedeny v následující kapitole).

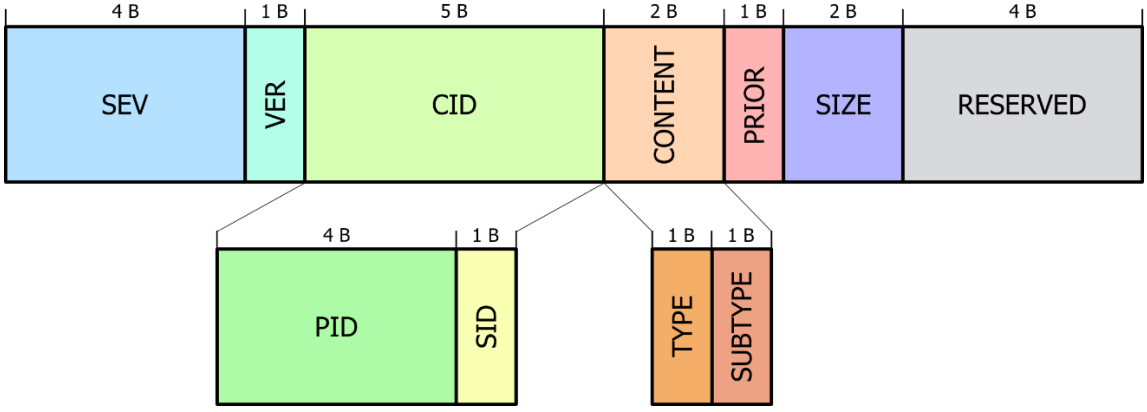
### 3.2.2 Vyšší úroveň a API

Vyšší úroveň knihovny je tvořena souborem *libipc.c* obsahujícím implementaci knihovny a hlavičkovým souborem *libipc.h*, který definuje API celé knihovny. Celá knihovna je rozdělena na čtyři části dle poskytovaných prostředků meziblokové komunikace – kanály zpráv, sady semaforů, RT signály a velkokapacitní kanály.

#### *3.2.2.1 Kanály zpráv*

Kanály zpráv jsou nejdůležitějším komunikačním mechanismem poskytovaným knihovnou *libipc* a jejich prostředky definované touto knihovnou lze rozdělit do dvou částí – na zprávy samotné a komunikační kanály určené k jejich přenášení.

Hlavičkový soubor *libipc.h* definuje zprávu (datový typ *T_IPC_MSGQ_MSG*) jako sbalenou strukturu skládající se z hlavičky a těla. Jelikož knihovna nepřiřazuje přenášeným datům význam, tělo zprávy je definováno pouze jako pole bajtů (datový typ *uint8_t*). Velikost těla zprávy je určena hodnotou preprocesorové konstanty IPC_MSGQ_MSG_DATA_MAX_SIZE a v současné době je nastavena na 256 bajtů. Konkrétní struktury těl jednotlivých zpráv jsou pak definovány v API souborech bloků, které jsou schopny dané zprávy přijímat. Hlavička zprávy je pak definována datovým typem *T_IPC_MSGQ_MSG_HEADER*. Na následujícím obrázku je vidět struktura tohoto typu.




**Obr. 9:** Struktura hlavičky meziblokové zprávy

Jak je vidět, hlavička zprávy je definována jako sbalená struktura skládající se z následujících položek.

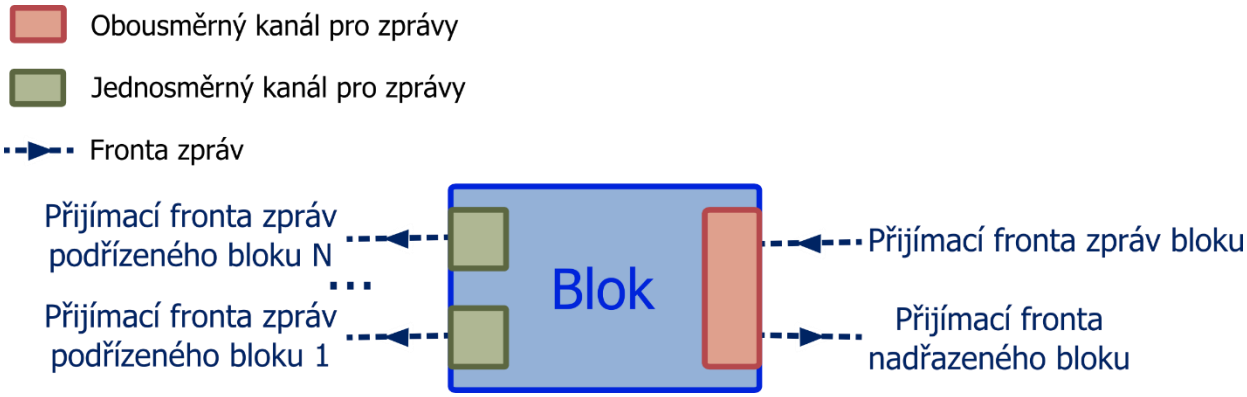
- ***sev*** – položka typu *long int* určující závažnost zprávy, tedy zda se jedná o kritickou či provozní zprávu.

- ***ver*** – položka typu *uint8_t* určující verzi protokolu. V aktuální implementaci je tato hodnota vždy rovna jedné.
- ***cid*** – položka jednoznačně identifikující komunikační relaci, jejíž hodnota je využívána nadřazenými bloky ke směrování odpovědí na požadavky. Jedná se o strukturu složenou z následujících dílčích položek.
    - ***pid*** – položka typu *pid_t* určující PID bloku iniciátora dané komunikační relace.
    - ***sid*** – položka typu *uint8_t* identifikující jednu komunikační relaci v bloku iniciátora.
- ***content*** – položka identifikující obsah zprávy, která je použita nadřazenými bloky ke směrování požadavků do svých podřízených bloků. Jedná se o strukturu skládající se z následujících položek.
    - ***type*** – položka typu *uint8_t* jednoznačně identifikující typ zprávy. Tato položka také typicky určuje blok první hierarchické úrovně (tedy blok přímo napojený na blok jádra), do kterého má být zpráva přesměrována.
    - ***subtype*** – položka typu *uint8_t* jednoznačně identifikující podtyp zprávy. Tato položka je typicky použita nadřazeným blokem první hierarchické úrovně (například DLC blokem) ke směrování zprávy do příslušného podřízeného bloku nebo adresovaným blokem ke zjištění účelu zprávy.
- ***prior*** – položka typu *uint8_t* určující prioritu zprávy. Tato položka není v současné době nijak využita, nicméně počítá se, že v budoucnu bude určovat, s jakou prioritou by měl cílový blok danou zprávu vyřizovat.
- ***size*** – položka typu *uint16_t* určující počet platných bajtů v těle zprávy.
- ***reserved*** – položka velikosti čtyřech bajtů, která není v současné době nijak využita.

V hlavičkovém souboru *libipc.h* jsou dále definovány výčtové datové typy určující možné závažnosti zprávy a také jejich typy a podtypy na základě návrhu uvedeného v předchozích kapitolách 2.4 Komunikace a 2.5 Návrh bloků. Detaily o definovaných typech lze nalézt v hlavičkovém souboru *libipc.h* (jeho zkrácená verze je dostupná v příloze A). Za zvláštní zmínku však stojí typ zprávy *IPC_MSGQ_N_T_RES*. Tato hodnota je použita v každé odpovědi na požadavek (v položce *content.type*), aby nadřazený blok směrující tuto odpověď poznal, že má směrování provádět na základě hodnoty položky *cid* v hlavičce zprávy. Hodnota položky *content.subtype* pak určuje, jaká data odpověď nese. Například v případě bloku řízení databáze a požadavku na zápis logu do logovací databáze bude ve zprávě s odpovědí na tento požadavek nastavena položka *content.type* na výše zmíněnou hodnotu *IPC_MSGQ_N_T_RES* a položka *content.subtype* bude nastavena na hodnotu *IPC_MSGQ_N_T_RES_S_DBDRES*, která určuje, že se jedná o odpověď na požadavek na provedení operace s databází.

Komunikační kanály pro zprávy definované knihovnou *libipc* využívají fronty zpráv a dají se rozdělit do dvou typů definovaných výčtovým datovým typem *T_IPC_MSGQ_CHANNEL_TYPE*. Prvním typem kanálu je tzv. obousměrný kanál pro zprávy (*IPC_MSGQ_CHANNEL_TW*) využívající jednu vysílací a jednu přijímací frontu zpráv. Při zápisu do obousměrného kanálu je zpráva vložena do odesílací fronty a při příjmu je zpráva vyčítána z přijímací fronty. Jak bylo řečeno v kapitole 2.4 Komunikace, každý blok bude disponovat jedinou přijímací frontou. Každá podřízený blok tedy bude využívat jeden obousměrný

kanál, jehož přijímací fronta bude přijímací frontou daného bloku a vysílací fronta bude přijímací frontou jeho nadřazeného bloku. Pokud je blok také nadřazeným blokem, bude prostřednictvím tohoto kanálu (a tedy své přijímací fronty) přijímat také zprávy zaslané jeho nadřazenými bloky. Dalším definovaným typem kanálu je tzv. jednosměrný kanál pro zprávy (*IPC_MSGQ_CHANNEL_OW*), který využívá pouze jedinou frontu zpráv, pomocí níž lze zprávy odesílat i přijímat. Kanály tohoto typu budou využívány nadřazenými bloky pro odesílání zpráv do jejich podřízených bloků. Každá nadřazený blok tedy bude využívat jeden jednosměrný kanál pro každý svůj podřízený blok tak, že fronta zpráv využívaná každým kanálem bude přijímací frontou daného podřízeného bloku. Využití obousměrných a jednosměrných kanálů z pohledu jednoho bloku je znázorněno na následujícím obrázku.




**Obr. 10:** Využití obousměrných a jednosměrných kanálů pro zprávy

Komunikační kanály pro zprávy jsou v hlavičkovém souboru *libipc.h* definovány datovým typem *T_IPC_MSGQ_CHANNEL*, což je struktura obsahující jednak typ kanálu pro zprávy (datový typ *T_IPC_MSGQ_CHANNEL_TYPE*) a pak paměťovou unii obsahující struktury určené k přístupu k frontám obousměrného a jednosměrného kanálu. Struktura obousměrného kanálu obsahuje deskriptory přijímací a vysílací fronty, kdežto struktura jednosměrného kanálu obsahuje pouze deskriptor jediné fronty kanálu a druhou, nevyužitou položku (tzv. pad). Pro provádění operací nad tímto datovým typem pak knihovna poskytuje následující funkce.

- ***ipc_msgq_create_tw(…)*** – tato funkce slouží k vytvoření obousměrného kanálu pro zprávy ze zadaných klíčů odesílací a přijímací fronty zpráv. Její návratová hodnota určuje, zda bylo vytvoření kanálu úspěšné, či nikoliv.
- ***ipc_msgq_create_ow(…)*** – tato funkce slouží k vytvoření jednosměrného kanálu pro zprávy ze zadaného klíče fronty zpráv. Její návratová hodnota určuje, zda bylo vytvoření úspěšné, či nikoliv.
- ***ipc_msgq_destroy(…)*** – tato funkce ničí specifický kanál pro zprávy. V případě, že tento kanál je obousměrný, je zničena pouze odesílací fronta zpráv.
- ***ipc_msgq_snd(…)*** – tato funkce slouží k odeslání specifické zprávy pomocí specifického kanálu pro zprávy. Pomocí parametrů je možné specifikovat, zda má tato funkce blokovat a případně jaký má být časový limit blokování. Pokud je daný kanál pro zprávy obousměrný, je k odeslání zprávy použita odesílací fronta zpráv tohoto kanálu. Návratová hodnota této funkce určuje úspěch či neúspěch odeslání.

- ***ipc_msgq_rcv(…)*** – tato funkce slouží k přijetí zprávy specifické závažnosti ze specifického kanálu pro zprávy. Pomocí parametru je možné specifikovat, zda má tato funkce blokovat. Pokud však tato funkce blokuje volající vlákno, může ho blokovat nekonečnou dobu. Pokud je daný kanál pro zprávy obousměrný, je zpráva vždy přijata z jeho přijímací fronty zpráv. Tato funkce vrací počet přijatých bajtů zprávy zmenšený o velikost první povinné položky zprávy (určující závažnost) v případě úspěchu, v opačném případě vrací záporné číslo.

Všechny tyto funkce využívají v rámci svých operací funkce poskytované nízko-úrovňovým wrapperem pro práci s frontami zpráv. Detaily o parametrech a návratových hodnotách těchto funkcí lze nalézt v hlavičkovém souboru *libipc.h* (jeho zkrácená verze je k dispozici v příloze A).

#### *3.2.2.2 Sady semaforů*

Pro práci se sadami semaforů definuje hlavičkový soubor knihovny datový typ sady semaforů (*T_IPC_SEMS_SET*). Ten je definován jako struktura skládající se z deskriptoru sady a její velikosti, tj. počtu semaforů v sadě. Nad tímto datovým typem pak následující funkce provádí své operace.

- ***ipc_sems_create(…)*** – tato funkce vytváří sadu semaforů zadané velikosti ze zadaného klíče. Její návratová hodnota určuje úspěch či neúspěch operace.
- ***ipc_sems_destroy(...)*** – tato funkce ruší specifickou sadu semaforů.
- ***ipc_sems_get_val(...)*** – tato funkce slouží k získání hodnoty specifického semaforu ve specifické sadě. Její návratová hodnota určuje hodnotu semaforu či neúspěch operace.
- ***ipc_sems_set_val(...)*** – tato funkce slouží k nastavení specifické hodnoty na specifickém semaforu ve specifické sadě. Může být použita například k inicializaci semaforů v sadě na určitou hodnotu. Její návratová hodnota určuje úspěch či neúspěch operace.
- ***ipc_sems_take(…)*** – tato funkce provádí pokus o odebrání specifické hodnoty ze specifického semaforu ve specifické sadě. Tato funkce může být blokující a může být volána s časovým limitem. Její návratová hodnota určuje úspěch či neúspěch operace.
- ***ipc_sems_give(…)*** – tato funkce provádí pokus o přidání specifické hodnoty do specifického semaforu ve specifické sadě. Tato funkce může být blokující a může být volána s časovým limitem. Její návratová hodnota určuje úspěch či neúspěch operace.

Veškeré uvedené funkce využívají funkce poskytované nízko-úrovňovým wrapperem pro práci se sadami semaforů. Více detailů o těchto funkcích, jako informace o jejich parametrech a návratových hodnotách, je k dispozici v hlavičkovém souboru *libipc.h* (zkrácená verze k dispozici v příloze A).

#### *3.2.2.3 Signály reálného času*

Hlavičkový soubor *libipc.h* definuje pro práci s RT signály podporované typy signálů (výčtový datový typ *T_IPC_RTSIG_SIGNAL*), formát parametru předávaného se signálem (datový typ *T_IPC_RTSIG_SIG_INFO*) a formát funkce signal handleru (datový typ *T_IPC_RTSIG_HANDLER*). Podporován je pouze jediný typ RT signálu, a to signál určený k vynucení změny stavu bloku (*IPC_RTSIG_SIG_FORCE_STATE*), jehož parametrem je stav, do kterého by měl adresovaný blok přejít.

Pro operace s RT signály poskytuje knihovna pouze dvě funkce. Jednu ke spárování signálu s funkcí ke zpracování signálu (funkce *ipc_rtsig_set_handler(...)*) a funkci k odeslání RT signálu s parametrem cílovému procesu (funkce *ipc_rtsig_snd(...)*). Obě tyto funkce pouze volají odpovídající funkce z nízkoúrovňového wrapperu pro práci s RT signály bez dalších dodatečných operací.

#### *3.2.2.4 Velkokapacitní kanály*

Knihovna poskytuje nástroje pro realizaci velkokapacitních přenosů využívající mechanismy popsané v kapitole 2.4.4 Velkokapacitní přenosy. V hlavičkovém souboru *libipc.h* je definován datový typ velkokapacitního kanálu (*T_IPC_NPIPE_CHANNEL*). Tento datový typ je struktura skládající se ze jména pojmenované roury, což je cesta k virtuálnímu souboru, deskriptoru virtuálního souboru pojmenované roury, režimu kanálu, blokujícímu příznaku a synchronizační sadou semaforů. Tato sada semaforů (datový typ *T_IPC_SEMS*_SET) je použita k synchronizaci mezi vysílacím a přijímacím blokem velkokapacitního přenosu. Režim kanálu je definován datovým typem *T_IPC_NPIPES_MODE* a určuje, zda je daný kanál určen ke čtení či k zápisu. Pokud je nastaven blokující příznak, tak veškeré operace prováděné nad daným kanálem mohou blokovat volající vlákno. Knihovna poskytuje pro práci s kanály následující funkce.

- ***ipc_npipe_create(…)*** – tato funkce slouží k vytvoření velkokapacitního kanálu se zadanými parametry. Synchronizační sada semaforů je vytvořena s jediným semaforem, který je inicializován na hodnotu 0. Návratová hodnota této funkce určuje úspěch či neúspěch vytvoření.
- ***ipc_npipe_destroy(…)*** – tato funkce ruší specifický velkokapacitní kanál smazáním virtuálního souboru odpovídající pojmenované roury.
- ***ipc_npipe_open(…)*** – tato funkce otevře specifický velkokapacitní kanál. Pokud je režim daného kanálu nastaven na čtení a pokud je blokující příznak daného kanálu nastaven, tato funkce může blokovat volající vlákno do té doby, než je otevřen kanál se stejnou pojmenovanou rourou s nastaveným režimem čtení. Návratová hodnota této funkce určuje úspěch či neúspěch operace.
- ***ipc_npipe_close(…)*** – tato funkce zavře specifický velkokapacitní kanál.
- ***ipc_npipe_wait(…)*** – tato funkce slouží k čekání na synchronizační signál specifického velkokapacitního kanálu a v typickém případě je volána přijímacím blokem ke zjištění, zda odesílající blok ukončil zápis dat do kanálu. Pokud je blokující příznak daného kanálu nastaven, může tato funkce blokovat se zadaným časovým limitem. Tato funkce využívá dříve popsané funkce pro práci se sadami semaforů z knihovny *libipc*. Návratová hodnota této funkce určuje, zda bylo čekání úspěšné či nikoliv.
- ***ipc_npipe_notify(…)*** – tato funkce slouží k odeslání synchronizačního signálu na specifickém velkokapacitním kanálu. Stejně jako funkce *ipc_npipe_wait* může blokovat volající vlákno s daným časovým limitem a využívá dříve popsané funkce pro práci se sadami semaforů. Návratová hodnota této funkce určuje, zda bylo odeslání úspěšné, či nikoliv.
- ***ipc_npipe_snd(…)*** – tato funkce zapisuje specifické množství dat ze specifického bufferu do specifického velkokapacitního kanálu. Pokud je blokovací příznak daného kanálu nastaven,

tato funkce může blokovat volající vlákno nekonečnou dobu. Vrací počet bajtů zapsaných do kanálu v případě úspěchu a záporné číslo v případě neúspěchu.

- ***ipc_npipe_rcv(…)*** – tato funkce vyčítá specifické množství dat ze specifického velkokapacitního kanálu a ukládá je do specifického bufferu. Pokud je blokovací příznak daného kanálu nastaven, tato funkce může blokovat volající vlákno nekonečnou dobu. Vrací počet bajtů vyčtených z kanálu v případě úspěchu a záporné číslo v případě neúspěchu.

Všechny uvedené funkce, s výjimkou funkcí pro synchronizaci velkokapacitního kanálu (*ipc_npipe_wait(...)* a *ipc_npipe_notify(...)*), využívají funkce nízko-úrovňového wrapperu pro práci s pojmenovanými rourami. Detaily o jejich parametrech a návratových hodnotách jsou k dispozici v hlavičkovém souboru *libipc.h* (zkrácená verze je k dispozici v příloze A).

## 3.3 Blok řízení chování

V rámci této diplomové práce byla implementována část bloku řízení chování v souladu s návrhem uvedeným v kapitole 2.5.6 Řízení chování. Jako programovací jazyk implementace byl zvolen jazyk C++ definovaný dle standardu C++11. Jelikož se jedná o jazyk objektově orientovaný, bylo vhodné navrhnout celý blok objektově. Z tohoto důvodu byl blok rozdělen na následující objekty.

- **Kontrolér** – tento objekt provádí své činnosti v rámci hlavního vlákna aplikace bloku řízení chování a stará se o řízení celého bloku a o kritickou komunikaci s blokem jádra.
- **Chybový kolektor** – k tomuto objektu přistupují všechny ostatní objekty bloku a slouží ke sběru chybových hlášení. Tento objekt je implementací mechanismu chybového čítače popsaného v kapitole 2.3 Návrh struktury.
- **Logger** – tento objekt nevyužívá vlastní vlákno a pouze poskytuje ostatním blokům logovací rozhraní.
- **Komunikátor** – tento objekt zaštiťuje činnosti, které by měly být dle návrhu prováděny komunikačním vláknem bloku, což je řízení provozní komunikace s blokem jádra. Nicméně, na rozdíl od návrhu, tento blok využívá dvě separátní vlákna.
- **DLMS klient** – tento objekt provádí své činnosti v rámci separátního vlákna a odpovídá vláknu DLMS klienta z návrhu.
- **Řadič stavových automatů** – tento objekt provádí činnosti spojené s řízením stavových automatů a tedy odpovídá vláknu řízení stavových automatů z návrhu. Veškeré operace provádí na separátním vlákně.
- **Plánovač** – tento objekt provádí operace spojené s rozvrhováním akcí vyžádaných objektem řadiče stavových automatů na jednotlivých zdrojích. Tyto operace provádí v kontextu separátního vlákna.

Detaily o jednotlivých objektech jsou uvedeny v následujících kapitolách.

### 3.3.1 Obecné prostředky

Většina objektů v bloku řízení chování vykonává své operace v rámci svého vlastního separátního vlákna. Pokud tedy potřebují dva objekty komunikovat navzájem, je nutné využít různých prostředků

mezivláknové komunikace. K implementaci vláken a operací s nimi je využita knihovna *pthreads* dostupná v použité Linuxové distribuci (konkrétně se jedná o implementaci knihovny NPTL s tzv. vlákny na úrovni jádra) a tedy jsou využívány i prostředky pro mezivláknovou komunikaci dostupné v této knihovně. V případě, že nějaký objekt potřebuje získat přístup k datům jiného objektu, musí vyčtení těchto dat provést přes metody dostupné u tohoto objektu. V rámci těchto metod pak musí cílový objekt zajistit ochranu datových závislostí mezi prováděnými operacemi nad těmito daty, například využitím mezivláknového mutexu (datový typ *pthread_mutex_t*). Jsou ale případy, kdy by určitý objekt mohl vyžadovat, aby jiný cílový objekt vykonal určitou operaci v rámci svého vlákna. V těchto situacích je tedy vhodné zaslat cílovému objektu požadavek na vykonání určité operace. K vyrovnání rychlostí jednotlivých objektů je žádoucí u každého objektu použít nějakou formu vyrovnávací paměti. Za tímto účelem byla implementována šablonová abstraktní třída *Container* (popsána v hlavičkovém souboru *Container.h*), která poskytuje základní rozhraní a metody, které by měl každý mezivláknový kontejner poskytovat. K zajištění vzájemného vyloučení vláken přistupujících ke kontejneru je využit mezivláknový mutex (datový typ *pthread_mutex_t*). K synchronizaci zápisových a čtecích operací jsou využity dvě podmínkové proměnné (datový typ *pthread_cond_t*). Třídy obsahuje veřejné abstraktní metody *Read(...)* a *Write(...)* pro čtení a zápis do kontejneru. Třída *Container* obsahuje následující chráněné metody.

- ***Lock(…)*** – tato metoda zajišťuje vzájemné vyloučení pomocí uzamčení ochranného mezivláknového mutexu kontejneru a měla by být volána jako první metoda v rámci přetížených metod *Read(...)* a *Write(...)* ve třídách potomků třídy *Container*. Tato metoda může v závislosti na parametrech blokovat volající vlákno s daným časovým limitem a její návratová hodnota určuje úspěch či neúspěch operace.
- ***Unlock(...)*** – tato metoda je protikladem metody *Lock(...)* a v jejím rámci je odemčen ochranný mezivláknový mutex kontejneru. Měla by být volána jako poslední v rámci přetížených metod *Read(...)* a *Write(...)* ve třídách potomků třídy *Container*. Návratová hodnota této funkce určuje úspěch či neúspěch.
- ***WaitSync(…)*** – tato metoda provádí čekání na specifický synchronizační signál kontejneru a může volající vlákno blokovat se specifickým časovým limitem. Synchronizační signály existují v rámci třídy *Container* dva – signál *SyncWrite* a signál *SyncRead* a jsou implementovány pomocí podmínkových proměnných. Tyto signály představují informaci o tom, zda je již možné do kontejneru provádět zápis či zda z něj již je možné číst. Tyto signály jsou využity v případech, kdy je kontejner prázdný a nějaké vlákno z něj chce číst, nebo kdy je kontejner plný a nějaké vlákno do něj chce zapisovat. V takovýchto případech metoda blokuje do doby obdržení synchronizačního signálu nebo vypršení časového limitu a vrací informaci o úspěchu či neúspěchu operace. V opačných případech funkce vždy vrací informaci o úspěchu operace a nikdy neblokuje. Účel metody by se dal shrnout následovně. Pokud tato metoda vrací hodnotu *true*, lze specifikovanou operaci s kontejnerem (čtení či zápis do něj) provést. Pokud vrací hodnotu *false*, tato operace nyní nemůže být provedena a funkce automaticky odemkne ochranný mutex. Tato

metoda by měla být volána v rámci metod *Read(...)* a *Write(...)* ve třídách potomků třídy *Container* ihned po zavolání metody *Lock(...)*.

- ***SignalSync(…)*** – tato metoda je protikladem metody *WaitSync(...)* a jejím prostřednictvím je odesílán specifický synchronizační signál kontejneru. Volána by měla být v rámci metod *Read(...)* a *Write(...)* ve třídách potomků třídy *Container* ihned po provedení dané operace s kontejnerem. V závislosti na stavu kontejneru před provedením operace může být v rámci této metody odeslán příslušný synchronizační signál s využitím příslušné podmínkové proměnné.

Všechny uvedené metody mohou v případě fatálních selhání způsobit vznik. Více detailů o třídě *Container* je k dispozici v hlavičkovém souboru *Container.h* (dostupný ve fomě elektronické přílohy).

Dále byla implementována šablonová třída *Buffer*, která je potomkem třídy *Container* a implementuje výše zmíněné mechanismy nad dvou koncovou frontou (třída *std::deque*), díky čemuž představuje vhodný prostředek poskytující výše zmíněnou meziobjektovou vyrovnávací paměť. Více detailů o této třídě je k dispozici v hlavičkovém souboru *Buffer.h* (dostupný ve formě elektronické přílohy).

K reprezentaci obecných požadavků přenášených mezi objekty bloku řízení chování byla implementována abstraktní třída *ControlMsg*. Tato třída obsahuje pouze členské proměnné potřebné k řízení případné fragmentace zpráv. Potomci této třídy pak reprezentují jednotlivé typy zpráv s požadavky a informace o nich jsou uvedeny v následujících kapitolách. Detaily o třídě *ControlMsg* lze spolu s informacemi o třídách jejích potomků lze nalézt v hlavičkovém souboru *ControlMsg.h*.

Pro reprezentaci obecného objektu využívajícího samostatné vlákno byla implementována abstraktní třída *ThreadedObject* (definována v hlavičkovém souboru *ThreadedObject.h*). Tato třída je využita jako předek většiny objektů bloku řízení chování a obsahuje následující členské proměnné.

- ***threadHandle*** – proměnná typu p*thread_t* sloužící jako handle na vlákno objektu.
- ***b_isRunning*** – volatilní proměnná typu *bool*, která určuje, zda vlákno daného objektu má, či nemá běžet.
- ***blockTimeout*** – proměnná typu *unsigned int*, která určuje délku časového limitu blokujících operací v objektu v milisekundách. Tato členská proměnná je staticky inicializována v rámci konstruktoru třídy.
- ***p_errCollHandle*** – tato proměnná je ukazatelem na objekt chybového kolektoru, který je využit pro sběr výskytů chyb. Více informací o tomto objektu bude uvedeno v následujících kapitolách.

Tato abstraktní třída obsahuje řadu veřejných metod definujících rozhraní pro řízení obecného objektu a také jednu chráněnou metodu reprezentující činnost vlákna objektu. Následující z nich jsou nejvýznamnější.

- ***Start(…)*** – tato veřejná metoda slouží ke spuštění vlákna objektu. V rámci této metody je voláním funkce *pthread_create(...)* vytvořeno nové vlákno, které provádí činnosti daného objektu. Zároveň je v rámci této funkce nastavena hodnota členské proměnné *b_isRunning* na hodnotu *true*. Návratová hodnota této funkce určuje úspěch či neúspěch operace.
- ***Stop(...)*** – tato veřejná metoda slouží k ukončení vlákna objektu. V rámci této metody je nejprve nastavena hodnota členské proměnné *b_isRunning* na hodnotu *false* a poté je vlákno objektu

zrušeno voláním funkce *pthread_cancel(...)*. Návratová hodnota této funkce určuje úspěch či neúspěch operace.

- ***IsRunning(…)*** – tato veřejná metoda vrací hodnotu privátní členské proměnné *b_isRunning*.
- ***ThreadFunction(…)*** – tato chráněná abstraktní metoda představuje činnost prováděnou vláknem objektu a je přetížena v každém potomkovi třídy *ThreadedObject*. V jejím rámci by měly objekty ve smyčce provádět příjem požadavků, jejich zpracování a vykonávání operací specifických pro daný objekt.

Více informací o metodách a členských proměnných abstraktní třídy *ThreadedObject* je k dispozici v hlavičkovém souboru *ThreadedObject.h* (dostupný ve formě elektronické přílohy).

### 3.3.2 Kontrolér

Jak již bylo řečeno, účelem objektu kontroléru je řízení celého bloku řízení chování. Jelikož je tato činnost důležitá, je prováděna v kontextu hlavního vlákna aplikace tohoto bloku. K reprezentaci tohoto objektu byla implementována třída *Controller* a v celém bloku existuje její jediná instance, která je staticky vytvářena v okamžiku spuštění aplikace bloku. Tato třída obsahuje následující privátní členské proměnné.

- ***config*** – tato členská proměnná je strukturou, která obsahuje proměnné představující platnou konfiguraci bloku řízení chování a také klíče k frontám zpráv použitých v komunikačním kanálu bloku. Položky konfigurace jsou stejné, jako navržené položky uvedené v kapitolách 2.3.5 Konfigurace a 2.5.6 Řízení chování. Více informací o proměnných v této struktuře lze najít v hlavičkovém souboru *Controller.h* (dostupný ve formě elektronické přílohy).
- ***blockStatus*** – jedná se o strukturu, jejíž proměnné slouží k udržení informace o aktuálním stavu bloku. Obsahuje následující členské proměnné.
    - ***currState*** – proměnná typu *T_COMM_BLK_STATUS* určující aktuální stav bloku.
    - ***reqState*** – proměnná typu *T_COMM_BLK_STATUS* určující stav bloku požadovaný externí změnou.
    - ***b_stateRequested*** – volatilní proměnná typu *bool* určující, zda byla externě vyžádána změna stavu.
    - ***b_running*** – proměnná typu *bool* určující, zda má programová smyčka objektu kontroléru, a tudíž i celý blok, běžet. Jedná se o obdobu členské proměnné *b_isRunning* abstraktní třídy *ThreadedObject*.
    - ***b_debugMode*** – proměnná typu *bool* určující, zda je blok spuštěn v tzv. ladícím režimu.
    - ***b_registered*** – proměnná typu *bool* určující, zda je blok zaregistrován u bloku jádra.
- ***commChannel*** – proměnná typu *T_IPC_MSGQ_CHANNEL* představující obousměrný komunikační kanál bloku.
- ***handles*** – tato členská proměnná je strukturou, která obsahuje ukazatele na veškeré ostatní objekty bloku a je využita k jejich řízení.

Třída *Controller* obsahuje následující privátní členské proměnné.

- ***ParseArguments(…)*** – tato metoda slouží k načtení argumentů aplikace bloku řízení chování. V současné době může mít aplikace nula, jeden, nebo dva argumenty. První argument vždy specifikuje cestu ke konfiguračnímu souboru bloku, a pokud je blok spuštěn bez argumentů, je použito výchozí umístění konfiguračního souboru, tedy cesta *„etc/controld.cfg“*. V případě, že je tento argument použit, tato funkce ověří, zda specifikovaný soubor existuje a v případě, že tomu tak není, je opět použito zmíněné výchozí umístění. Druhým argumentem může být povolení ladícího režimu bloku. V tomto režimu objekt kontroléru vynechává veškeré registrační procesy a blok je tak možné spustit i bez bloku jádra. Tato metoda vrací hodnotu *true*, pokud bylo načtení argumentů úspěšné a hodnotu *false* v opačném případě.
- ***Register(…)*** – tato metoda provádí pokus o registraci bloku u bloku jádra. S využitím funkce *ipc_msgq_snd(...)* a členské proměnné *commChannel* odešle žádost o registraci. Poté čeká konfigurovanou dobu na odpověď od bloku jádra. Pokud dorazí pozitivní odpověď (potvrzení registrace), vrací tato metoda hodnotu *true*. V opačném případě, tedy pokud dorazí negativní odpověď (zamítnutí registrace), pokud nedorazí v časovém limitu žádná odpověď nebo pokud se žádost o registraci nepodaří do uplynutí časového limitu odeslat, vrací tato metoda hodnotu *false*.
- ***Unregister(…)*** – tato metoda odesílá do bloku jádra žádost o zrušení registrace.
- ***SubscribeEvents(...)*** – tato metoda odesílá do bloku jádra žádost o registraci k oběru notifikací o událostech. Obdobně jako metoda *Register(...)*, vrací tato metoda hodnotu určující úspěch či neúspěch registrace, akorát s tím rozdílem, že na tuto zprávu neexistuje žádná odpověď a tedy je sledován pouze úspěch odeslání žádosti.
- ***LoadConfiguration(…)*** – tato metoda provádí pokus o načtení konfigurace z konfiguračního souboru bloku řízení chování. Jelikož je konfigurační soubor ve formátu XML, je k načtení jeho obsahu využita knihovna *libxml2* a jí poskytnuté prostředky. V případě, že načítání konfigurace selže, vrací tato metoda hodnotu *false*, v opačném případě vrací hodnotu *true*.
- ***ParseMessage(…)*** – tato metoda slouží k načtení a zpracování obsahu zprávy kritické komunikace a k případnému vytvoření odpovědi na tuto zprávu. Pokud je vytvořena odpověď, která by měla být odeslána, vrací tato metoda hodnotu *true*, v opačném případě vrací hodnotu *false*.
- ***StartObjects(…)*** – v rámci této metody jsou spuštěny všechny ostatní spustitelné objekty bloku řízení chování. Metoda vrací hodnotu určující úspěch či neúspěch spuštění.
- ***StopObjects(…)*** – v rámci této metody jsou ukončeny všechny běžící objekty bloku řízení chování.

S třídou *Controller* také souvisí funkce *SignalHandler(...)*. Tato funkce je z pohledu třídy *Controller* tzv. *friend* funkcí, což znamená, že tato funkce má absolutní přístup ke všem členským proměnným a metodám této třídy. Tato funkce je využita jako handler RT signálů bloku řízení chování. Při přijetí signálu je tedy tato funkce zavolána a v případě, že je signál a jeho parametr platný, vyžádá tato funkce v globální instanci třídy *Controller* změnu stavu prostřednictvím jejích členských proměnných *blockStatus.reqState* a *blockStatus.b_stateRequested*.

Třída *Controller* obsahuje jedinou veřejnou metodu, kterou je metoda *BlockLoop(...)*. Tato metoda představuje hlavní programovou smyčku objektu kontroléru a tedy i celého bloku řízení chování. Tato metoda je volána jako jediná v rámci funkce *main(...)* aplikace bloku řízení chování. Její návratová hodnota také určuje návratovou hodnotu aplikace. V rámci této metody je vykonáváno veškeré řízení bloku a veškerá kritická komunikace, a tudíž v ní nelze provádět žádné blokující operace. V těle této metody je prováděna aplikační smyčka objektu. V jejím rámci je nejprve ověřeno, zda nebyla externě vyžádána změna stavu bloku a pokud ano, tak je aktuální stav změněn na požadovaný. Poté je vykonán krok stavového automatu bloku. V definovaných stavech bloku jsou v souladu s návrhem uvedeným v kapitole 2.5.1 Obecný návrh prováděny následující operace.

- **Studený start** – v tomto stavu jsou nejprve pomocí metody *ParseArguments(...)* načteny argumenty aplikace bloku, poté je vytvořen obousměrný komunikační kanál bloku a nakonec je navázána metoda *SignalHandler(...)* na RT signál vynucení změny stavu. V případě jakéhokoliv neúspěchu je stav změněn na Ukončování, jinak je stav změněn na Horký start.
- **Horký start** – v tomto stavu je nejprve voláním metody *StopObjects(...)* a zničením všech ostatních objektů bloku zajištěno, že s výjimkou staticky vytvářených objektů nejsou žádné objekty v bloku vytvořeny ani spuštěny. Poté je načtena konfigurace bloku pomocí funkce *LoadConfiguration(...)*. V případě neúspěchu jsou načteny výchozí hodnoty všech parametrů definované v hlavičkovém souboru *Defaults.h*. Poté jsou vytvořeny všechny ostatní objekty bloku řízení chování s příslušnými parametry. Pokud vytváření objektů selže, je stav změněn na Ukončování a pokud uspěje, jsou prostřednictvím metod *Register(...)* a *SubscribeEvents(...)* provedeny registrace u bloku jádra a registrace odběru notifikací o událostech. V případě selhání registrace u bloku jádra je stav bloku změněn na Ukončování, avšak v případě selhání registrace odběru notifikací o událostech je stav změněn na Chybový. Procesy registrace jsou přeskočeny v případě, že je blok spuštěn v režimu ladění. Dále je provedeno napojení příslušných objektů na sebe prostřednictvím nastavení příslušných ukazatelů a nakonec je zavoláním metody *StartObjects(...)* proveden pokus o spuštění objektů bloku. V případě úspěchu spuštění všech objektů je stav bloku změněn na Běh, v opačném případě je stav bloku změněn na Ukončování.
- **Běh** – V tomto stavu je nejprve volána metoda *StartObjects(...)*, aby bylo zajištěno, že všechny spustitelné objekty skutečně běží. Poté je prováděn pokus o vyčtení kritické zprávy z komunikačního kanálu a v případě úspěchu zpracování této metodou *ParseMsg(...)* a případě odeslání kritické zprávy s odpovědí. Poté je prostřednictvím objektu chybového kolektoru provedena kontrola aktuálního chybového okna. V případě, že tato kontrola neuspěje, je stav bloku změněn na Chybový.
- **Nečinnost** – v tomto stavu nejsou prováděny žádné činnosti s výjimkou volání metody *StopObjects(...)* k zajištění, že objekty nejsou v tomto stavu spuštěné.
- **Chybový stav** – v tomto stavu jsou prováděny stejné činnosti, jako ve stavu Běhu, kromě kontroly aktuálního chybového okna.

- **Ukončování** – v tomto stavu jsou nejprve všechny spuštěné objekty zastaveny voláním metody *StopObjects(...)* a poté jsou zničeny. Nakonec je členská proměnná *blockStatus.b_running* nastavena na hodnotu *false*, což v dalším cyklu programové smyčky způsobí její ukončení.

Detailní informace o třídě *Controller*, jejích členských proměnných a metodách lze nalézt v hlavičkovém souboru *Controller.h* (dostupný ve formě elektronické přílohy).

### 3.3.3 Chybový kolektor

Objekt chybového kolektoru nemá, na rozdíl od ostatních objektů bloku řízení chování, vyhrazené vlastní vlákno, neboť sám žádnou činnost neprovádí. Pouze poskytuje prostředky ke sběru chybových hlášení a ke kontrole počtu chyb v chybovém okně, čímž implementuje dříve popsaný mechanismus chybového čítače. Objekt chybového kolektoru je implementován prostřednictvím třídy *ErrorCollector*. V rámci celého bloku řízení chování existuje jediná, objektem kontroléru vytvořená, instance této třídy a všechny objekty bloku řízení chování mají k dispozici ukazatel na tuto instanci. Třída *ErrorCollector* obsahuje vnořenou třídu *Error*, která obsahuje pouze tři členské proměnné – titulek chyby a popis chyby, což jsou textové řetězce reprezentované datovým typem *std::string* a časovou značku (datový typ *time_t*) vzniku chyby. Třída *ErrorCollector* pak obsahuje následující privátní členské proměnné.

- ***errTimeWindow*** – proměnná typu *time_t*, která představuje délku chybového okna ve vteřinách. Tato hodnota je staticky inicializována v konstruktoru třídy.
- ***maxErrFrequency*** – proměnná typu *unsigned int*, která představuje maximální počet chyb za jedno chybové okno.
- ***blockTimeout*** – časový limit blokujících operací v milisekundách.
- ***errors*** – proměnná typu *std::vector<Error>* představující chybový vektor kolektoru a obsahuje všechny chyby sesbírané za aktuální chybové okno.

Za účelem ochrany přístupu vzájemným vyloučením obsahuje třída *ErrorCollector* také ochranný mezivláknový mutex (typ *pthread_mutex_t*).

Za účelem sběru a kontroly chyb disponuje třída *ErrorCollector* následujícími veřejnými metodami.

- ***ReportError(…)*** – metoda sloužící ke sběru chyb. V rámci této metody je specifická chyba vložena do chybového vektoru kolektoru. Vložení je chráněno ochranným mezivláknovým mutexem objektu chybového kolektoru. Návratová hodnota určuje úspěch či neúspěch operace. Tato metoda je volána ostatními objekty bloku v případě výskytu jakékoliv neočekávané chyby.
- ***CheckErrors(…)*** – metoda sloužící ke kontrole chybového okna a je volána pravidelně objektem kontroléru bloku ve stavu běhu. V rámci této metody jsou nejprve z chybového vektoru odstraněny chyby starší, než je nastavená délka chybového okna a poté je zkontrolováno, zda je aktuální velikost chybového vektoru menší, než je nastavená mez. V průběhu celé kontroly je uzamčen ochranný mezivláknový mutex objektu. Pokud kontrola uspěje, nebo pokud se v časovém limitu nepodaří uzamknout ochranný mutex, metoda vrací hodnotu *true*, v opačném případě vrací hodnotu *false*.

Další informace o třídě *ErrorCollector*, jejích členských proměnných a metodách lze nalézt v hlavičkovém souboru *ErrorCollector.h* (dostupný ve formě elektronické přílohy).

### 3.3.4 Logger

Obdobně jako objekt chybového kolektoru, objekt loggeru nevyužívá žádné separátní vlákno a pouze poskytuje ostatním objektům rozhraní pro logování. Tento objekt je implementován třídou *Logger*, jejíž jediná existující instance v rámci bloku bude staticky vytvářena při spouštění bloku. Tato třída je popsána v hlavičkovém souboru *Logger.h*, neobsahuje žádné členské proměnné a obsahuje jedinou veřejnou metodu *OutputLog(...)*, která slouží k zapsání zadané informace do logu. V současné době je log pouze vypisován na standardní výstup a slouží k výpisu veškerých logů, nicméně v pozdější fázi vývoje dojde k přechodu na logovací systém syslog a objekt tedy bude využit pouze pro nouzové logování dle návrhu.

### 3.3.5 Komunikátor

Jak bylo uvedeno v předchozích kapitolách, účelem objektu komunikátoru je řízení provozní komunikace bloku řízení chování s blokem jádra a tedy poskytování všem ostatním objektům bloku rozhraní pro komunikaci s ostatními bloky. Tento objekt zajišťuje příjem příchozích zpráv meziblokové komunikace, jejich překlad do příslušných zpráv meziobjektové komunikace a přesměrování přeložených zpráv do odpovídajících bloků. Dále také přebírá zprávy meziobjektové komunikace, které jsou určeny k odeslání do bloku jádra a ty překládá do zpráv meziblokové komunikace.

Pro reprezentaci tohoto objektu byla implementována třída *Communicator*, jejíž definice je v hlavičkovém souboru *Communicator.h*. Tato třída sice disponuje stejným rozhraním jako třída *ThreadedObject* a její potomci, nicméně potomkem této třídy není. Místo toho obsahuje dvě členské proměnné; jeden objekt multiplexoru a jeden objekt demultiplexoru. Objekt multiplexoru provádí veškeré činnosti spojené s komunikací, která je z pohledu bloku odchozí, kdežto objekt demultiplexoru vykonává činnosti spojené s komunikací z pohledu bloku příchozí. Tyto objekty jsou instancemi tříd *CommMux* a *CommDemux*, které obě dědí z abstraktní třídy *ThreadedObject*. Činnosti objektu komunikátoru jsou tedy prováděny nikoliv jedním, ale hned dvěma separátními vlákny. Metody třídy *Communicator* napodobující rozhraní třídy *ThreadedObject* volají odpovídající funkce objektů multiplexoru a demultiplexoru (metody *Start(...)*, *Stop(..)* a *IsRunning(...)*). Obě zmíněné třídy jsou také definovány v hlavičkovém souboru *Communicator.h*.

Třída *CommMux*, popisující objekt multiplexoru, obsahuje členskou proměnnou *inBuffer* datového typu *Buffer<ControlMsg *>*, která představuje vyrovnávací paměť multiplexoru. Veškeré zprávy meziobjektové komunikace, které by měli být objektem multiplexoru zpracovány, jsou ukládány do tohoto bufferu. Aby poskytovala ostatním objektům rozhraní pro vkládání zpráv do svého vnitřního bufferu, obsahuje třída *CommMux* veřejnou metodu *QueryMsg(...)*, která vkládá specifický ukazatel na zprávu do tohoto bufferu. Veřejnou metodu s totožnou signaturou obsahuje také třída *Communicator* a ta slouží jako wrapper této metody objektu multiplexoru. V rámci svého vlákna (tj. zděděné metody *ThreadFunction(...)*) objekt multiplexoru vyčítá zprávy ze svého interního bufferu, prostřednictvím

privátní metody *ParseMessage(...)* tyto zprávy překládá do zpráv meziblokové komunikace a pomocí funkce *ipc_msgq_snd(...)* knihovny *libipc* tyto zprávy poté odesílá do bloku jádra.

Objekt demultiplexoru je popsán třídou *CommDemux* a obsahuje členskou proměnnou *p_channelHandle* typu ukazatel na komunikační kanál pro zprávy knihovny *libipc* (datový typ *T_IPC_MSGQ_CHANNEL* *) a také řadu členských proměnných, které jsou ukazateli na objekty bloku řízení chování. V kontextu svého vlákna (tj. zděděné metody *ThreadFunction(...)*) objekt demultiplexoru vyčítá zprávy z komunikačního kanálu prostřednictvím funkce *ipc_msgq_rcv(...)* knihovny *libipc*, přijaté zprávy překládá do zpráv meziobjektové komunikace voláním privátní metody *ParseMessage(...)* a tyto zprávy poté přeposílá do příslušných objektů.

### 3.3.6 DLMS klient

Objekt DLMS klienta je reprezentován třídou *DLMSClient*, která je potomkem třídy *ThreadedObject*. Jelikož však v době vniku této diplomové práce nebyla k dispozici vhodná klientská aplikace standardu DLMS/COSEM, nebyla tato třída plně implementována a neprovádí v rámci svého vlákna žádné činnosti. Dokončení implementace této třídy bude předmětem další fáze vývoje po získání vhodné klientské aplikace.

### 3.3.7 Řadič stavových automatů

Objekt řadiče stavových automatů, implementován ve třídě *AutomataDriver* dědící ze třídy *ThreadedObject*, slouží k realizaci požadované konfigurovatelné autonomie datového koncentrátoru. Ta je, dle návrhu, uskutečněna využitím systému stavových automatů definovaného v XML souboru respektujícím deklarační jazyk nástroje UPPAAL (viz. [5]). Tento verifikační, simulační a návrhový nástroj využívá tzv. systému časovaných stavových automatů. Systém časovaných automatů se skládá z celočíselných, logických, synchronizačních, plošně synchronizačních a časových proměnných a ze stavových automatů Mealyho typu jednoznačně odlišených identifikačním textovým řetězcem. Časované proměnné jsou celočíselné proměnné, jejichž hodnota je neustále periodicky zvyšována. Proměnné mohou být buď globální, a tedy společné pro všechny stavové automaty, nebo lokální pro jeden specifický stavový automat. To se však netýká obou typů synchronizačních proměnných, neboť ty mohou být pouze globální. Všechny přechody stavových automatů jsou popsány třemi následujícími textovými řetězci.

- **Synchronizační akce** – slouží ke svázání dvou a více hran prostřednictvím jedné synchronizační proměnné. Synchronizační akce je vždy ve tvaru *<Název proměnné>?* nebo *<Název proměnné>!* v závislosti na tom, zda je daný přechod na synchronizaci závislý, nebo zda jí vyvolává. V případě, že je proveden přechod vyvolávající synchronizaci nad určitou synchronizační proměnnou, musí být proveden buď právě jeden (synchronizační proměnná), nebo všechny (plošná synchronizační proměnná) synchronizačně závislé přechody, jejichž synchronizační akce využívá stejnou synchronizační proměnnou.
- **Ochranná podmínka** – jedná se o algebraický výraz obsahující proměnné systému. Přechod lze provést pouze v případě, že je ochranná podmínka splněna.

- **Přiřazovací operace** – jedná se skupinu přiřazení hodnot do proměnných systému. Přiřazená hodnota může být dána algebraickým výrazem. Přiřazení je provedeno v okamžiku, kdy je daný přechod proveden.

Dále je možno deklarovat globální či lokální funkce, které mají syntax podobnou jazyku C a které mohou provádět operace nad proměnnými.

V rámci simulace chodu automatů v nástroji UPPAAL je v každém simulačním kroku vybrán jeden z možných přechodů, tj. přechodů, jejichž ochranná podmínka je splněna a jehož synchronizační akci lze provést) a tento přechod je vykonán. Pokud tento přechod vyvolává synchronizaci, je v závislosti na typu synchronizační proměnné proveden jeden, či všechny ze synchronizačně závislých přechodů, které využívají stejnou synchronizační podmínku a jejichž ochranná podmínka je splněna. V okamžiku provedení přechodu jsou vykonány veškeré přiřazovací operace přechodu. Jak je vidět, tento průběh simulací je velice nedeterministický, a tedy jeho použití v objektu řadiče stavových automatů není vhodné. Vhodnější je v každém kroku běhu provést přechod každého stavového automatu, u kterého bude provedení přechodu možné. Zároveň je také vhodné využívat pouze plošné synchronizační proměnné. Pokud tedy v následujícím textu budou zmiňovány synchronizační proměnné, bude se jednat pouze o celoplošné synchronizační proměnné. Tato opatření sice nezajistí úplný determinismus běhu automatů, nicméně jej přinejmenším mohou zvýšit.

Jelikož stavové automaty řízené objektem řadiče stavových automatů budou muset reagovat na řadu událostí a budou generovat žádosti o provedení různých akcí v rámci celého softwarového vybavení koncentrátoru, je třeba deklarační jazyk systému UPPAAL využít tak, aby to bylo možné. Výskyt události je velmi podobný synchronizační akci, neboť v okamžiku výskytu události je žádoucí provedení všech možných přechodů, které jsou na výskytu dané události závislé. Reakce na výskyt události tedy lze v definičním souboru stavových automatů popsat jako synchronizační akci daného přechodu ve tvaru *E_<Název události>?*. V rámci této diplomové práce byla implementována pouze podpora pro výskyt události aktualizace tabulky měřidel v DLC bloku. Závislost nějakého přechodu na výskytu této události lze v definičním souboru tedy popsat jako synchronizační akci tohoto přechodu ve tvaru *E_EMTableUpdate?*.

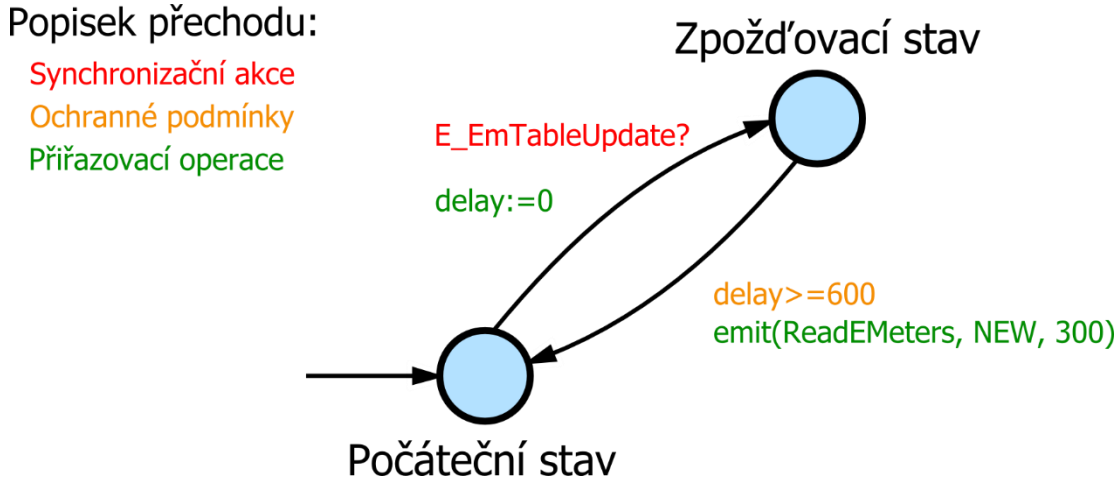
Vyžádání vykonání určité akce přibližně odpovídá přiřazovacím operacím na přechodech. Jelikož deklarační jazyk systému UPPAAL umožňuje definování funkcí, je vhodné vyžádání provedení akce realizovat jako volání funkce. V případě, že má být na přechodu provedeno vyžádání provedení specifické akce, je tedy nutné k přiřazovacím operacím daného přechodu přidat přiřazovací operaci ve tvaru *emit(<Název akce>, <Parametry>, <Časový limit v sekundách>)*. V rámci této diplomové práce byla implementována podpora pro akci vyčtení měřidel, která tedy může být popsána jako přiřazovací operace *emit(ReadEMeters, <NEW nebo ALL>, <Časový limit v sekundách>)*. Hodnota druhého parametru (*NEW* nebo *ALL*) určuje, zda mají být vyčteny pouze nově připojená či všechna měřidla. Je možné, že v dalších částech vývoje, poté co dojde ke specifikaci dalších typů akcí, přibydou všem akcím další parametry. Například bude vhodné, když u akce vyčtení měřidel bude možné určit měřidla, která by měla být vyčtena, nebo COSEM objekty, které budou z měřidel vyčteny. Také je možné, že dojde ke změně formátu

deklarace žádostí o vykonání akcí tak, že každá akce bude specifikována jako přiřazovací operace ve tvaru *<Název akce>(<Časový limit v sekundách>, <Parametry>)*.

Dále je žádoucí, aby stavový automat měl možnost se dozvědět, zda akce, o jejíž provedení naposledy žádal, byla dokončena úspěšně či neúspěšně. Tento mechanismus je, stejně jako mechanismus reakcí na události, podobný synchronizačním akcím. V případě, že na určitém přechodu má být provedeno čekání na úspěšné či neúspěšné dokončení akce, lze tuto skutečnost vyjádřit prostřednictvím synchronizační akce daného přechodu ve tvaru *W<S nebo F>_<Název akce>?*. Například v případu reakce na úspěšné dokončení akce vyčtení měřidel bude mít synchronizační akce příslušného přechodu tvar *WS_ReadEMeters?*.

Aby bylo možné navržené stavové automaty ověřit pomocí verifikačního nástroje sytému UPPAAL, je nezbytné, aby v deklarační části definičního souboru byla definována jedna synchronizační proměnná s odpovídajícím názvem pro každou událost a úspěšné či neúspěšné dokončení specifické akce, na kterou dané stavové automaty mohou reagovat. Ze stejného důvodu je dále nutné, aby funkce *emit(...)*, představující odeslání žádosti o provedení akce, byla také uvedena v definičním souboru systému stavových automatů. Aby mohl být v rámci verifikace systému simulován vznik událostí a dokončení žádaných akcí, je nezbytné, aby definiční soubor systému obsahoval také jednoduché automaty, které pouze budou vyvolávat synchronizace nad odpovídajícími synchronizačními proměnnými. Aby v rámci běhu stavových automatů v bloku řadiče stavových automatů nedošlo k interferencím se skutečným výskytem událostí či dokončení žádaných akcí, budou identifikační textové řetězce těchto automatů začínat slovem *virtual_*.

Na následujícím obrázku je zobrazen příklad stavového automatu k vyčítání nově připojených měřidel.




**Obr. 11:** Příklad automatu vyčítání nově připojených měřidel

Po detekci události aktualizace tabulky měřidel počká automat 10 minut a poté zažádá o vyčtení všech nově připojených měřidel s 5minutovým časovým limitem.

Pro účely reprezentace systému stavových automatů bylo nutné implementovat řadu tříd. K reprezentaci obecné proměnné byla implementována abstraktní třída *AutVar* definovaná v hlavičkovém souboru *Variables.h*. V tomto hlavičkovém souboru jsou také definovány třídy jejích potomků. Těmi jsou

třídy *SyncVar*, *IntVar*, *TimedVar* a *BoolVar* reprezentující proměnné jednotlivých podporovaných typů. Každá z těchto tříd obsahuje členské proměnné určující název (datový typ *std::string*) a hodnotu této proměnné (datový typ *int*). Abstraktní třída *AutVar* také obsahuje vnořenou třídu *Hasher* sloužící k výpočtu rozptylového kódu proměnných.

V hlavičkovém souboru *Events.h* je definována abstraktní třída *Event* reprezentující obecnou událost a také třídy jejích potomků. Třída *Event* obsahuje veřejnou členskou proměnnou určující typ události a veřejnou členskou proměnnou datového typu *std::*string určující textový popis dané události. Tento textový popis musí odpovídat názvu události použitému v definičním souboru systému stavových automatů. V rámci této diplomové práce byla implementována jediná třída dědící z třídy *Event*, a to třída *EmTableUpdateEvent* představující událost aktualizace tabulky měřidel v DLC bloku. Tato třída obsahuje vnořenou třídu *EMStatus* reprezentující stav jednoho konkrétního měřidla, a veřejnou členskou proměnnou datového typu *std::vector<EMStatus>* představující vektor aktualizovaných záznamů tabulky měřidel. Pro reprezentaci prováděných akcí byla implementována abstraktní třída *Action* definovaná v hlavičkovém souboru *Actions.h*. Tato třída obsahuje následující veřejné členské proměnné.

- ***type*** – tato proměnná definovaného výčtového datového typu *ActionType* určuje typ akce. V současné době tento výčtový typ obsahuje pouze hodnotu *ActionReadEMeters*.
- ***descr*** – textový řetězec (datový typ *std::string*) určující textový popis akce. Tento popis se musí shodovat s názvem akce použitým v definičním souboru systému stavových automatů.
- ***prior*** - tato proměnná datového typu *unsigned* určuje prioritu dané akce a je využita pouze objektem plánovače při rozvrhování úlohy spojené s danou akcí.
- ***timeOut*** – proměnná typu *time_t* určující časový limit akce ve vteřinách.
- ***b_reportBack*** – pokud je tato členská proměnná datového typu *bool* nastavená na hodnotu *true*, má být výsledek dané akce zaslán zpět do objektu řadiče stavových automatů.
- ***b_reportSuccess*** – tato proměnná datového typu *bool* určuje, zda má být objektu řadiče stavových automatů zasíláno oznámení o úspěchu či neúspěchu akce.

Tento soubor také obsahuje definice tříd potomků třídy *Action*. V rámci této diplomové práce byla implementována pouze třída *ReadEMetersAction* reprezentující akci vyčtení měřidel. Tato třída navíc obsahuje členské proměnné určující, zda mají být data vyčteny ze všech, či pouze z nově připojených měřidel a také obsahuje vektor interních identifikátorů vyčítaných měřidel (datový typ *std::vector<unsigned>*). Třída *Action* je spolu s třídami jejích potomků využita také objektem plánovače (detaily následují v kapitole 3.3.8 Plánovač).

V rámci hlavičkového souboru *Automata.h* je definována třída *SyncAction* reprezentující synchronizační akci přechodu. Tato třída obsahuje veřejné členské proměnné určující textový řetězec synchronizační akce (datový typ *std::string*), název související synchronizační proměnné (datový typ *std::string*), události či akce, dále typ synchronizace, tj. zda se jedná o reakci na výskyt události, na dokončení poslední vyžádané akce či provedení synchronizace přes synchronizační proměnnou a nakonec informaci o tom, zda, v případě využití synchronizační proměnné, je daná akce vyvoláním synchronizace či čekáním na ni (vše datové typy *bool*).

Hlavičkový soubor *Automata.h* dále obsahuje definici třídy *GuardCond*. Tato třída představuje ochrannou podmínkou přechodu a obsahuje jedinou veřejnou členskou proměnnou – textový řetězec (datový typ *std::string*) představující algebraicko-logický výraz ochranné podmínky.

K reprezentaci přiřazovacích operací je v hlavičkovém souboru *Automata.h* definována třída *AssignOp* reprezentující přiřazovací operaci přechodu. Tato třída obsahuje vnořenou třídu *VarAssign* představující přiřazení hodnoty algebraicko-logického do proměnné. Třída *VarAssign* obsahuje členské proměnné pro specifikaci názvu proměnné (datový typ *std::string*), algebraicko-logického výrazu určujícího její hodnotu (datový typ *std::string*) a hodnoty tohoto výrazu (datový typ *int*). Třída *AssignOp* obsahuje následující veřejné členské proměnné.

- ***varsAssign*** – proměnná datového typu std::*vector*<*VarAssign*> představující vektor přiřazení do proměnných.
- ***actions*** – proměnná datového typu std::*vector*<*Action* *> představující vektor akcí k provedení.

Jelikož ochranné podmínky a některé přiřazovací operace jsou realizovány jako algebraicko-logické výrazy, je v hlavičkovém souboru *Automata.h* definována třída *Evaluator* poskytující statické funkce k vyhodnocování těchto výrazů. V rámci těchto výrazů je možné použít veškeré standardní operátory, s výjimkou unárního operátoru negace a ternárního operátoru.

V hlavičkovém souboru *Automata.h* jsou dále implementovány třídy *State*, *Transition* a *Automaton* pro reprezentaci stavových automatů. Třída *State* představuje stav automatu a obsahuje následující veřejné členské proměnné.

- ***id*** – členská proměnná datového typu *std::string* sloužící jako unikátní identifikátor stavu.
- ***timeStamp*** – tato proměnná datového typu *time_t* určující časovou značku posledního vstupu stavového automatu do daného stavu slouží zejména k diagnostickým účelům.
- ***transition*** – tato proměnná datového typu std::*vector*<*Transition* *> je vektorem ukazatelů na odchozí přechody daného stavu.

Třída *Transition* reprezentuje přechod stavového automatu a obsahuje následující veřejné členské proměnné.

- ***source*** – tato proměnná představuje zdrojový stav přechodu (datový typ *State &*).
- ***sink*** – tato proměnná představuje cílový stav přechodu (datový typ *State &*).
- ***sync*** – synchronizační akce přechodu (datový typ *SyncAction*).
- ***guard*** – ochranná podmínka přechodu (datový typ *Guard*).
- ***assign*** – přiřazovací operace přechodu (datový typ *AssignOp*).

Třída *Automaton* byla implementována za účelem reprezentace stavového automatu v systému časovaných stavových automatů. Obsahuje následující veřejné členské proměnné.

- ***name*** – textový řetězec ve formě datového typu *std::string* sloužící jako unikátní identifikátor stavového automatu.
- ***b_loaded*** – proměnná datového typu *bool*, která určuje, zda byl daný automat úspěšně načten z definičního souboru.

- ***b_running*** – proměnná datového typu *bool*, která určuje, zda má daný stavový automat běžet v rámci objektu řadiče stavových automatů.
- ***b_stateChanged*** – tato proměnná datového typu *bool* je využita při běhu stavového automatu a určuje, zda daný stavový automat v rámci aktuálního kroku provedl změnu stavu.
- ***lastIssuedAct*** – tato členská proměnná je strukturou obsahující informace o poslední vyžádané akci, jako je její textový identifikátor (datový typ *std::string*) a příznak určující, zda byla akce dokončena a zda úspěšně či nikoliv.
- ***p_initState*** – ukazatel na počáteční stav (datový typ *State* *) automatu.
- ***p_currState*** – ukazatel na aktuální stav (datový typ *State* *) automatu.
- ***states*** – vektor možných stavů automatu (datový typ *std::vector<State>*).
- ***transitions*** – vektor všech přechodů stavového automatu (datový typ *std::vector<Transition>*).
- ***timedVars*** – hashovaná množina lokálních časových proměnných stavového automatu (datový typ *std::unordered_set<TimedVar, TimedVar::Hasher>*).
- ***intVars*** – hashovaná množina lokálních celočíselných proměnných stavového automatu (datový typ *std::unordered_set<IntVar, IntVar::Hasher>*).
- ***boolVars*** – hashovaná množina lokálních logických proměnných stavového automatu (datový typ *std::unordered_set<BoolVar, BoolVar::Hasher>*).

Třída *Automaton* dále obsahuje veřejné metody k vytvoření lokální proměnné z její deklarace (metoda *ParseVariableDeclaration(...)*), aktualizaci hodnot lokálních časových proměnných daného automatu (metoda *UpdateTimedVars(...)*) a ke zrušení stavového automatu (metoda *Clear(...)*).

Poslední třída definovaná v rámci hlavičkového souboru *Automata.h* je třída *AutTrans* reprezentující jeden pár automat – přechod. Obsahuje pouze dvě veřejné členské proměnné, a to ukazatele na příslušný stavový automat a příslušný přechod.

Detailní informace o všech uvedených třídách jsou dostupné v hlavičkových souborech definujících tyto třídy (*Automata.h* je ve zkrácené verzi dostupný v příloze B, ostatní jsou dostupné formou elektronické přílohy).

Jak bylo uvedeno, třída *AutomataDriver* reprezentující objekt řadiče stavových automatů je definována v hlavičkovém souboru *AutomataDriver.h*. Obsahuje následující členské proměnné.

- ***descrFilePath*** – textový řetězec (datový typ *std::string*) obsahující cestu k definičnímu souboru systému stavových automatů.
- ***automata*** – vektor stavových automatů (datový typ *std::vector<Automaton>*).
- ***syncVars*** – hashovaná množina globálních synchronizačních proměnných systému (datový typ *std::unordered_set<SyncVar, SyncVar::Hasher>*).
- ***timedVars*** – hashovaná množina globálních časových proměnných systému (datový typ *std::unordered_set<TimedVar, TimedVar::Hasher>*).
- ***intVars*** – hashovaná množina globálních celočíselných proměnných systému (datový typ *std::unordered_set<IntVar, IntVar::Hasher>*).

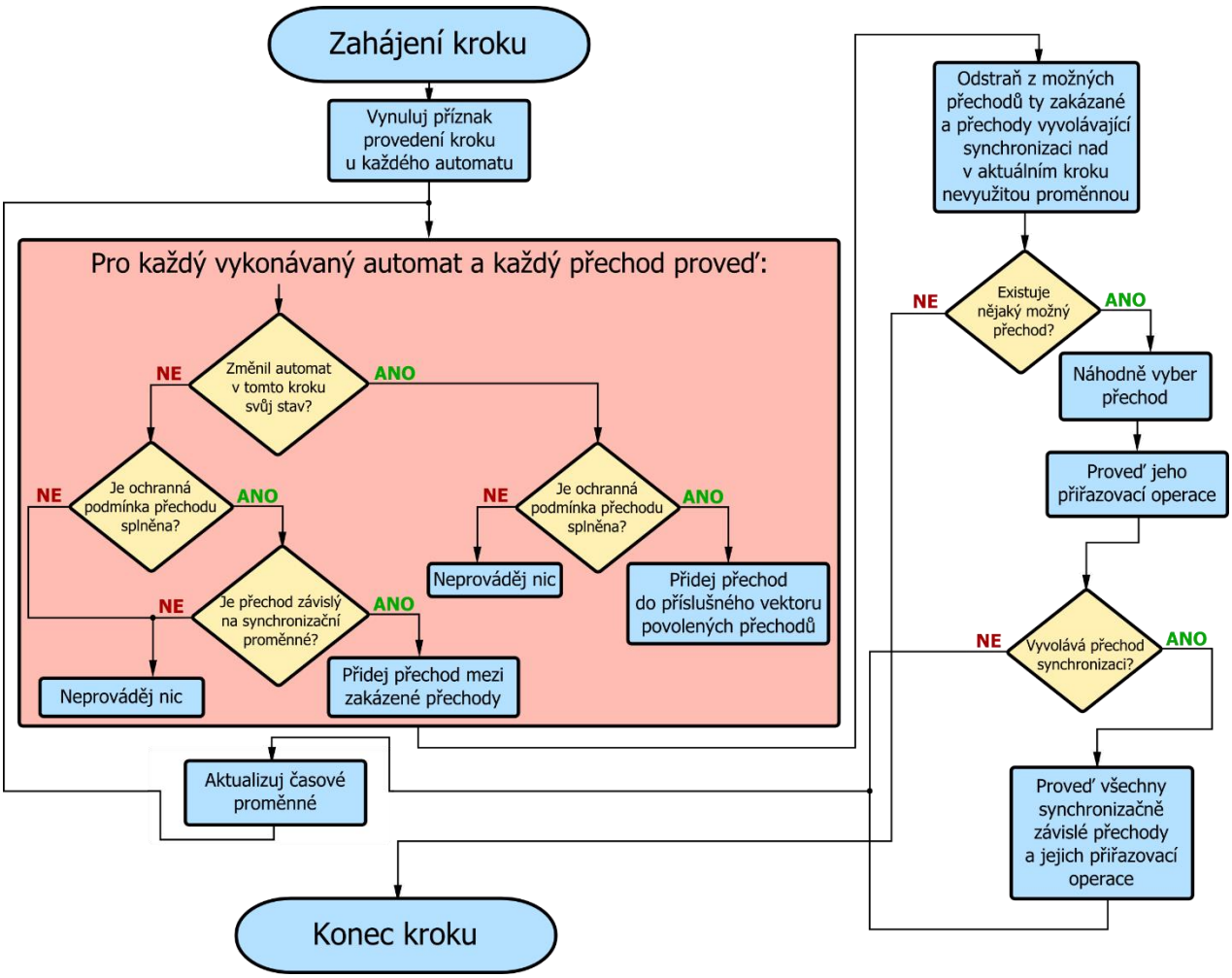
- ***boolVars*** – hashovaná množina globálních logických proměnných systému (datový typ *std::unordered_set<BoolVar, BoolVar::Hasher>*).
- ***autVars*** – tato členská proměnná je vektorem ukazatelů na proměnné (datový typ *std::vector<AutVar *>*). Obsahuje ukazatele na veškeré lokální i globální proměnné v rámci aktuálně načteného systému stavových automatů. Tato členská proměnná je využita při vyhodnocování algebraických výrazů, kdy je reference na ní předána statické metodě *Evaluate(...)* třídy *Evaluator*, díky čemuž může tato metoda provést dosazení hodnot proměnných do výrazů.
- ***events*** – tato proměnná datového typu *std::deque<Event *>* představuje dvou koncovou frontu příchozích událostí.
- ***inBuffer*** – proměnná datového typu *Buffer<ControlMsg *>* sloužící jako meziobjektová vyrovnávací paměť. Veškeré příchozí zprávy meziobjektové komunikace jsou do ní vkládány a později zpracovávány.
- ***p_schedHandle*** – tato proměnná datového typu ukazatel na objekt plánovače je používána k odesílání žádostí o vykonání úlohy do objektu plánovače.
- ***p_commHandle*** – tato proměnná datového typu ukazatel na objekt komunikátoru slouží k případnému odesílání požadavků do objektu komunikátoru. V současné době není nijak využita, nicméně v dalších fázích implementace bude využita k odesílání požadavků na zápis dat do logovací databáze.

Třída *AutomataDriver* obsahuje následující privátní metody.

- ***LoadAutomata(…)*** – tato metoda slouží k inicializaci systému stavových automatů. Nejprve je provedeno načtení proměnných a automatů voláním metody *ParseAutomata(...)* a poté je, v případě úspěchu, naplněn vektor ukazatelů na proměnné (členská proměnná *autVars*). Nakonec jsou inicializovány všechny příznaky stavového automatu. Tato funkce vrací hodnotu *true*, pokud načtení celého systému proběhlo bez problémů. V opačném případě vrací hodnotu *false*.
- ***ParseAutomata(…)*** – tato metoda slouží k načtení systému stavových automatů z definičního souboru. Jelikož jsou definiční soubory ve formátu XML, je k jeho načtení využito nástrojů poskytnutých knihovnou *libxml2*. Návratová hodnota této metody určuje úspěch či neúspěch načítání.
- ***ParseMessage(…)*** – tato metoda slouží k přečtení obsahu přijaté meziobjektové zprávy, vykonání případných operací spojených s přijatou zprávou a vytvoření případné meziobjektové zprávy s odpovědí. Pokud byla vytvořena zpráva s odpovědí, vrací funkce hodnotu *true*, v opačném případě vrací hodnotu *false*. Jestliže zpracovaná zpráva oznamuje úspěch či neúspěch některé z žádaných akcí, je tato akce u příslušného stavového automatu označena jako úspěšně či neúspěšně dokončena zápisem do členské proměnné *lastIssuedAct*. Tato metoda je volána v rámci metody *ThreadFunction(...)*, zděděné z abstraktní třídy ThreadedObject, pouze v případě úspěšného vyčtení meziobjektové zprávy z bufferu.

- ***ParseVariableDeclaration(…)*** – tato metoda je obdobou stejnojmenné metody třídy *Automaton* a slouží k vytvoření globální proměnné z její deklarace. Pokud byla proměnná úspěšně vytvořena, vrací hodnotu *true*, v opačném případě vrací hodnotu *false*.
- ***UpdateGlobalVars(…)*** – metoda sloužící k aktualizaci hodnot globálních časových proměnných.
- ***CheckAutomata(…)*** – tato metoda je jádrem celého objektu řadiče stavových automatů a představuje jeden krok v řízení chodu stavových automatů. Tento krok je prováděn pouze nad vykonávanými stavovými automaty (tj. automaty, jejichž členská proměnná *b_running* je nastavena na hodnotu *true*) následujícím způsobem. Nejprve je u všech stavových automatů nastavena hodnota jejich členské proměnné *b_stateChanged* na hodnotu *false*, je načtena událost z čela fronty příchozích událostí a je vytvořena řada vektorů obsahujících možné přechody (datové typy *std::vector<AutTrans>*). Jedná se o vektor možných přechodů reagujících na událost, vektor možných přechodů reagujících na dokončení vyžádané akce, vektor možných přechodů závislých na synchronizaci (využívající nějaké synchronizační proměnné) a nakonec vektor všech ostatních možných přechodů. Tyto vektory jsou tvořeny pouze z přechodů automatů, které v aktuálním kroku dosud nezměnily svůj stav. Dále je vytvořen vektor zakázaných přechodů tvořený možnými synchronizačně závislými přechody těch automatů, které v aktuálním kroku svůj stav již změnily. Z vektoru ostatních možných přechodů jsou poté odstraněny přechody vyvolávající synchronizaci nad proměnnou, na jejíž synchronizaci není závislý žádný z nalezených možných přechodů a přechody vyvolávající synchronizaci nad proměnnou, na níž je synchronizačně závislý některý ze zakázaných přechodů. To, zda je daný přechod možný, je ověřeno vyhodnocením ochranné podmínky přechodu. Poté je vybrán jeden náhodný možný a nezakázaný přechod tak, že přechody reagující na události jsou upřednostňovány před přechody reagujícími na dokončení žádaných akcí, které jsou zase upřednostňovány před ostatními možnými přechody. Pokud byl vybrán možný přechod, je proveden a jsou vykonány s ním svázané přiřazovací operace. Pokud daná přiřazovací operace obsahuje akce k provedení, jsou žádosti o jejich provedení odeslány do objektu plánovače formou meziobjektové zprávy s žádostí o vykonání úlohy. Pokud zvolený přechod vyvolává synchronizaci prostřednictvím určité synchronizační proměnné, jsou provedeny všechny přechody synchronizačně závislé na stejné proměnné a jsou vykonány přiřazovací operace těchto přechodů. U automatů, u kterých byl proveden přechod, je poté nastaven příznak *b_stateChanged* na hodnotu *true*. Poté je voláním metod *UpdateTimedVars(...)* a *UpdateGlovalVars(...)* provedena aktualizace hodnot všech časových proměnných. Celý tento proces, s výjimkou počátečního nastavení členské proměnné *b_stateChanged* u všech automatů na hodnotu *false*, je opakován tak dlouho, dokud není možné vybrat žádný přechod. V tom okamžiku je daný krok běhu ukončen a dochází k návratu z této metody. Tato metoda je volána v rámci metody *ThreadFunction* zděděné z abstraktní třídy *ThreadedObject*.

Na následujícím obrázku je shrnut proces jednoho kroku běhu systému stavových automatů prováděný v rámci metody *CheckAutomata(...)*.




**Obr. 12:** Proces kroku stavových automatů

Třída *AutomataDriver* dále disponuje veřejnými metodami pro nastavení ukazatelů na objekty komunikátoru a plánovače a veřejnou metodou *QueryMsg(...)*, která slouží ke vložení ukazatele na meziobjektovou zprávu do interního bufferu.

Objekt řadiče stavových automatů využívá řadu zpráv meziobjektové komunikace. K získání aktuálních informací o stavových automatech slouží zpráva s žádostí o tyto informace (třída *GetAutInfoControlMsg*) a jako odpověď na tento požadavek slouží zpráva obsahující tyto informace (třída *AutInfoControlMsg*). Jako žádost o provedení operace se stavovým automatem slouží zpráva reprezentovaná třídou *AutOperationControlMsg*. Touto zprávou lze vynutit ukončení či zahájení vykonávání specifického stavového automatu, změnu jeho aktuálního stavu, případně znovunačtení celého systému stavových automatů. Výsledek operace je pak zasílán zprávou reprezentovanou pomocí třídy *AutOpResultControlMsg*. Pro reprezentaci meziobjektové zprávy notifikace o výskytu události slouží třída *EvtNotifyControlMsg*. Tato třída obsahuje veřejnou členskou proměnnou ukazatele na událost (datový typ *Event* *). Všechny tyto zprávy vznikají v objektu komunikátoru překladem odpovídajících přijatých zpráv meziblokové komunikace. Ke komunikaci s objektem plánovače slouží třída *ActRequestControlMsg* reprezentující zprávu s žádostí o vykonání úlohy obsahující členskou proměnnou ukazatele na akci, která by měla být v rámci úlohy vykonána (datový typ *Action* *) a také třída *ActFinishedControlMsg* představující zprávu ohlašující dokončení úlohy obsahující členské proměnné poskytující informace

o úloze a jejím úspěchu či neúspěchu. Veškeré zmíněné třídy meziobjektových zpráv jsou potomky třídy *ControlMsg*.

Detaily o třídě *AutomataDriver*, jejích členských proměnných a metodách jsou k dispozici v hlavičkovém souboru *AutomataDriver.h* (jeho zkrácená verze je k dispozici v příloze B).

### 3.3.8 Plánovač

Objekt plánovače, jenž je realizován implementovanou třídou *Scheduler*, definovanou v hlavičkovém souboru *Scheduler.h*, která je potomkem třídy *ThreadedObject*, má za úkol rozvrhovat požadavky na vykonání úloh vzniklé v objektu řadiče stavových automatů na jednotlivé zdroje. Těmito zdroji budou v typickém případě DLP bloky, jelikož standard DLMS/COSEM znemožňuje otevření více současných komunikačních relací na jediné síti měřidel. Nicméně v průběhu vývoje se mohou objevit i jiné typy zdrojů, jako například konzolové rozhraní či databáze. V současném stavu je implementována podpora jediného zdroje – PLC-DLP bloku.

Aby byly zdroje využívány efektivně, je třeba vhodně využívat systém rozvrhování navržený v kapitole 2.5.6 Řízení chování. Vzhledem k tomu, že objekt řadiče stavových automatů nemá žádné informace o přibližné době provádění požadovaných úloh, může provedení celé úlohy trvat velice dlouhou dobu. Příkladem takové úlohy je vyčítání měřidel. Pokud je vyžádána úloha vyčtení všech měřidel, může, vzhledem ke komunikačním rychlostem na PLC sítích, tato úloha trvat i několik hodin (v závislosti na rušení v síti a počtu měřidel). Samotné požadované úlohy tedy nejsou vhodným subjektem rozvrhování a objekt plánovače proto bude provádět tzv. atomizaci úloh. Ta spočívá v rozdělení úlohy na dílčí úkoly tak, že tyto úkoly již nejde dále rozdělit. V případě úlohy vyčtení všech měřidel tedy dojde k atomizaci na úkoly vyčtení jednotlivých měřidel. Tyto úkoly jsou již vhodným subjektem rozvrhování, a tedy budou rozděleny do rozvrhovacích subsystémů jednotlivých zdrojů, jejichž úkolem bude efektivně rozvrhovat úkoly přidělené jednomu konkrétnímu zdroji. Jak bylo určeno v návrhu uvedeném v kapitole 2.5.6 Blok řízení chování, tyto rozvrhovací subsystémy jsou v současné implementaci založeny na prioritních FSFS frontách s mechanismem zrání priorit. Každá akce, definovaná potomky dříve popsané třídy *Action*, má přiřazen časový limit. Při procesu atomizace je tento časový limit použit i jako časový limit akcí dílčích úkolů. Tento časový limit je primárně hlídán objektem provádějícím danou akci, nicméně kvůli zvýšení odolnosti proti chybám je vhodné časové limity sledovat také v rámci rozvrhování. Proto je tedy objektem plánovače hlídáno překročení časových limitů dílčích úkolů o 20 %.

V hlavičkovém souboru *PQueSys.h* (dostupném ve zkrácené verzi v příloze C) je definována třída *PQueSys*, která reprezentuje jeden systém prioritních FCFS front a představuje tak rozvrhovací subsystém jednoho zdroje. Tato třída obsahuje vnořenou třídu, *ActionPlan*, která je použita k reprezentaci jednoho atomického úkolu a obsahuje veřejné členské proměnné pro sledování stavu daného úkolu a ukazatel na příslušnou akci (typ *Action* *), která má být v rámci daného úkolu provedena. Třída *PQueSys* obsahuje následující členské proměnné.

- ***pqSys*** – tato proměnná je pole (*std::array*) proměnných typu dvou koncová fronta ukazatelů na úkol (*std::deque<ActionPlan *>*). V rámci tohoto pole existuje tolik front, kolik existuje

priorit, a tudíž velikost tohoto pole je definováno hodnotou preprocesorové konstanty *PQSYS_MAX_PRIORITY*. Tato proměnná představuje systém prioritních FCFS front.

- ***p_issuedPlan*** – tato proměnná je ukazatelem na aktuálně vykonávaný úkol (datový typ *ActionPlan* *).

Třída *PQueSys* obsahuje následující veřejné metody.

- ***SubmitActionPlan(…)*** – tato metoda slouží ke vložení jednoho úkolu (datového typu *ActionPlan* *) do rozvrhovacího subsystému. Při vkládání je kontrolována nejprve fronta odpovídající prioritě přidružené akce. V případě, že v této frontě již není místo (velikost fronty je definována konstantou preprocesoru *PQSYS_MAX_QUEUE_SIZE*), jsou postupně kontrolovány fronty s nižší a poté i s vyšší prioritou. Ukazatel na vkládaný úkol je vložen na konec první nalezené fronty, ve které je volné místo.
- ***Check(…)*** – tato metoda provádí plánovací operace nad rozvrhovacím subsystémem. V rámci této funkce jsou nejprve ze systému prioritních front odstraněny ukazatele na ty úkoly, které byly ukončeny. Poté je prověřeno, zda byl dokončen aktuálně prováděný úkol. Pokud ano, je nahrazen novým z čela fronty s nejvyšší prioritou. Následně je provedeno zrání úkolů. To je realizováno tím, že u každé fronty, kromě té s nejvyšší prioritou, je vložen úkol z jejího čela na konec fronty s prioritou o 1 větší, pokud je v ní volné místo. V případě, že byl aktuálně vykonávaný úkol nahrazen novým, vrací funkce hodnotu *true* a ukazatel na tento úkol je vrácen pomocí výstupního argumentu funkce. V opačném případě vrací funkce hodnotu *false*.

Třída *Scheduler* obsahuje vnořenou třídu *ActionRequest* reprezentující požadavek na vykonání úlohy. Tato třída obsahuje veřejné členské proměnné určené ke sledování stavu úlohy, k identifikaci stavového automatu, který tento požadavek vyslal, ukazatel na akci (typ *Action* *), která má být v rámci úlohy provedena a také vektor úkolů (typ *std::vector<PQueSys::ActionPlan>*), na něž byla úloha rozdělena v procesu atomizace. Třída *Scheduler* také definuje podporované zdroje prostřednictvím výčtového datového typu *Resource*. V současné době tento výčtový typ obsahuje pouze zdroj PLC-DLP bloku (*ResourcePLCProxy1*). Třída *Scheduler* obsahuje následující privátní členské proměnné.

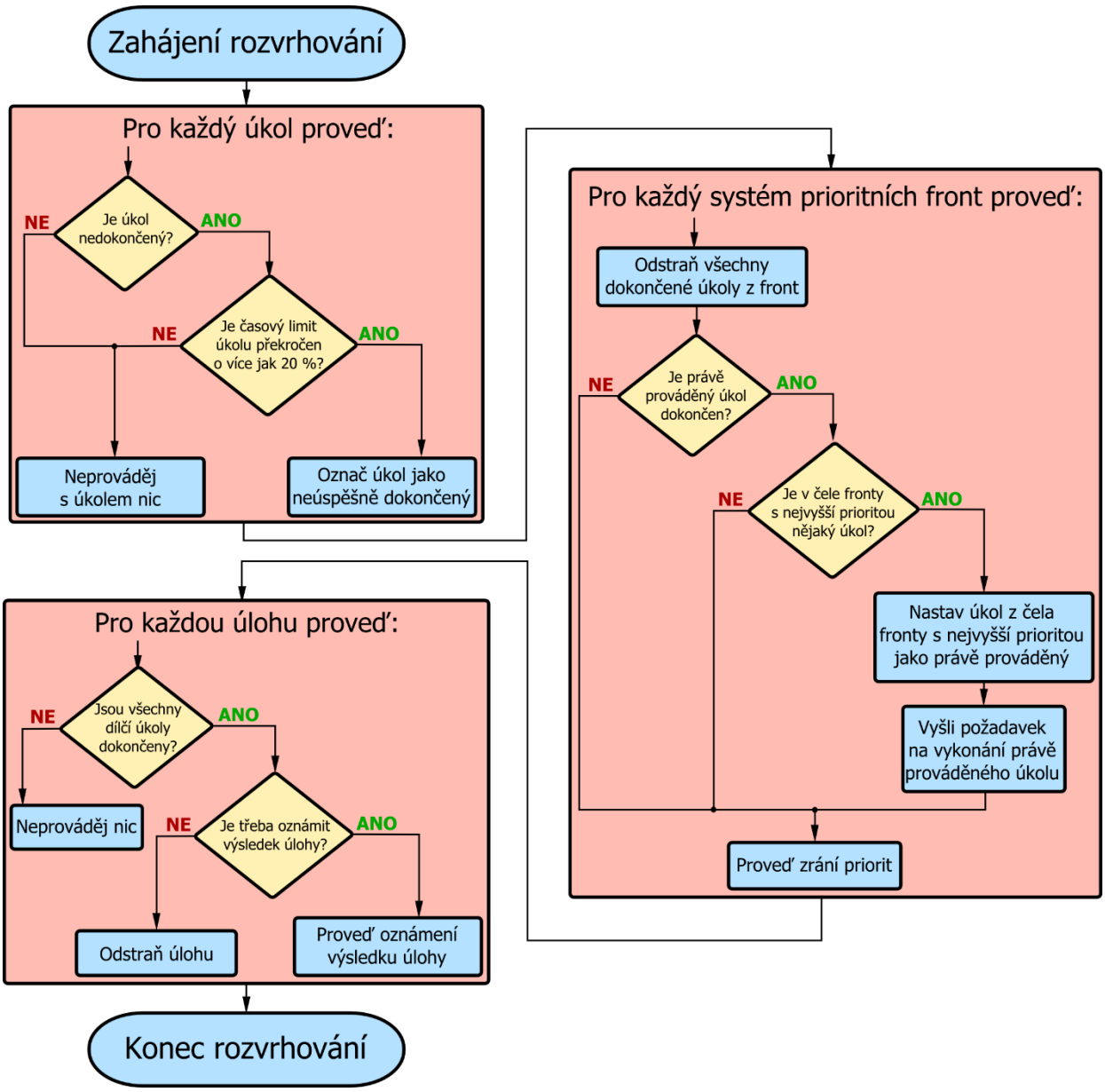
- ***priorQueSystems*** – tato proměnná je pole systémů prioritních front (datový typ *std::array<PQueSys>*). Velikost tohoto pole je určena počtem podporovaných zdrojů, který je definován konstantou preprocesoru *SCHEDULER_RESOURCES_COUNT*.
- ***pendingRequests*** – tato členská proměnná je vektor úloh čekajících na vyřízení (datový typ *std::vector*<ActionRequest>).
- ***inBuffer*** – tato proměnná typu *Buffer<ControlMsg* *> slouží jako meziobjektová vyrovnávací paměť objektu plánovače a jsou v ní ukládány příchozí meziobjektové zprávy.
- ***p_commHandle*** – tato proměnná je typu ukazatel na objekt komunikátoru. Jejím prostřednictvím jsou odesílány požadavky na provedení operací v jiných blocích do objektu komunikátoru.
- ***p_autDriverHandle*** – tato proměnná datového typu ukazatel na objekt řadiče stavových automatů slouží k odesílání potvrzení o dokončení úlohy tomuto objektu.

- ***p_dlmsClientHandle*** – tato proměnná typu ukazatel na objekt DLMS klienta nemá v současné implementaci, díky absenci klientské aplikace, žádné využití, nicméně v budoucnu bude sloužit k provádění operací týkajících se komunikace s měřidly.

Třída *Scheduler* obsahuje následující privátní metody.

- ***ParseMessage(…)*** – tato metoda slouží k přečtení obsahu přijaté meziobjektové zprávy, vykonání případných operací spojených s přijatou zprávou a vytvoření případné meziobjektové zprávy s odpovědí. Pokud byla vytvořena zpráva s odpovědí, vrací funkce hodnotu *true*, v opačném případě vrací hodnotu *false*. Pokud zpracovaná zpráva oznamuje úspěch či neúspěch některého z prováděných úkolů, je příslušný úkol označen jako úspěšně či neúspěšně dokončený. Tato metoda je volána v rámci metody *ThreadFunction(…)*, zděděné z abstraktní třídy ThreadedObject, pouze v případě úspěšného vyčtení meziobjektové zprávy z bufferu.
- ***AddActionRequest(...)*** – účelem této metody je přidat do rozvrhovacího systému novou úlohu. V rámci této metody je prováděna atomizace přidávané úlohy a vytvořené úkoly jsou vkládány do rozvrhovacích subsystémů odpovídajících zdrojů voláním metody *SubmitActionPlan(…)* systému prioritních front příslušného zdroje.
- ***IssueAction(…)*** – tato metoda je použita pro vykonání specifické akce reprezentované dříve popsaným datovým typem *Action*. V jejím rámci je vytvořena zpráva meziobjektové komunikace s odpovídajícím požadavkem a je odeslána příslušnému objektu.
- ***Schedule(…)*** – tato metoda v sobě zahrnuje veškeré operace rozvrhování prováděné objektem plánovače. V rámci této funkce jsou nejprve zkontrolovány časové limity všech probíhajících úkolů, a pokud došlo u některého z nich k překročení časového limitu o více než 20 %, je tento úkol považován za neúspěšně dokončený. Poté je provedena operace rozvrhování u každého rozvrhovacího subsystému zavoláním výše popsané metody *Check(…)* příslušného systému prioritních front. V případě, že je v rozvrhovacím subsystému daného zdroje k dispozici nový úkol k vykonání, je tento úkol označen jako právě vykonávaný a zavoláním metody *IssueAction(…)* na akci spojenou s tímto úkolem je jeho vykonávání zahájeno. Poté je provedena kontrola dokončení úloh. U každé probíhající úlohy je prověřeno, zda je každý její dílčí úkol dokončen. Pokud ano a výsledek akce spjaté s úlohou má být oznámen stavovému automatu, ve kterém vznikl požadavek na vykonání této úlohy, je do objektu řízení stavových automatů odeslána příslušná zpráva. Úloha je považována za úspěšně dokončenou, pokud byl úspěšně dokončen každý z jejích dílčích úkolů. Poté je dokončená úloha odstraněna z vektoru probíhajících úloh. Tato metoda je volána v rámci zděděné metody *ThreadFunction(…)*.

Na následujícím obrázku je vidět shrnutí algoritmu rozvrhování použitého objektem plánovače v rámci metody *Schedule(…)*.




**Obr. 13:** Proces rozvrhování

Třída *Scheduler* dále disponuje veřejnými metodami pro nastavení ukazatelů na ostatní objekty a veřejnou metodou *QueryMsg(...)*, která slouží ke vložení ukazatele na meziobjektovou zprávu do interního bufferu. Objekt plánovače je schopný zpracovávat pouze zprávy s požadavkem na vykonání úlohy (reprezentované třídou *ActRequestControlMsg* dědící z třídy *ControlMsg*) a zprávy oznamující úspěšné dokončení akce prováděné v rámci nějakého úkolu. Jak bylo uvedeno, v metodě *IssueAction(...)* jsou z objektu plánovače odesílány meziblokové zprávy s požadavky na vykonání konkrétních akcí. Jak bylo zmíněno v kapitole 3.3.7 Řadič stavových automatů, v současné době je podporována pouze akce vyčtení měřidel (reprezentována třídou *ReadEMetersAction*, která je potomkem třídy *Action*) a tedy jediná zpráva odesílána v rámci metody *IssueAction(...)* je zpráva požadavku na vyčtení měřidla (reprezentována třídou *ReadEMeterControlMsg*, která je potomkem třídy *ControlMsg*). Tato zpráva instruuje objekt DLMS klienta k vyčtení COSEM objektů ze zadaného měřidla. Dále jsou při vykonávání metody *Schedule(...)* do objektu řadiče stavových automatů odesílány meziobjektové zprávy s potvrzením o dokončení požadované úlohy. Tyto zprávy jsou reprezentovány třídou *ActFinishedControlMsg*, která je potomkem

třídy *ControlMsg* a obsahuje informace identifikující akci dokončenou v rámci úlohy, původní požadavek na vykonání úlohy, identifikátor stavového automatu, ve kterém tento požadavek vznikl a informace o úspěchu či selhání úlohy.

Více informací o třídě *Scheduler* lze najít v hlavičkovém souboru *Scheduler.h* (dostupný ve zkrácené verzi v příloze C).

## 3.4 Testování a verifikace

Vzhledem k účelu a budoucímu využití zařízení CAM3600 tvoří proces testování a verifikace jeho softwarového vybavení velmi důležitou součást jeho vývoje. Jak již bylo několikrát řečeno, je totiž žádoucí, aby, na rozdíl od řady konkurenčních zařízení, byl koncentrátor CAM3600 co nejvíce spolehlivý a příčiny případných chyb byly snadno dohledatelné. Řadu testů lze samozřejmě provést už v době vývoje, nicméně některé, zejména tzv. vektorové testy, je vhodnější provádět až v pozdějších fázích vývoje.

V průběhu vývoje knihovny *libipc* byly její nízko-úrovňové wrappery testovány prostřednictvím vytvořených testovacích aplikací. Knihovna byla jako celek testována ihned po jejím dokončení prostřednictvím vytvořené aplikace *libipc_example*. Pomocí této aplikace bylo ověřeno, že prostředky knihovny zvládají bezchybné provádění toho, k čemu byly navrženy, tedy že zvládají odesílání provozních a kritických zpráv, manipulaci se sadami semaforů, práci s RT signály a realizaci velkokapacitních přenosů, včetně jejich synchronizace. Jelikož je knihovna aktivně využívána při vývoji jednotlivých bloků, lze při jejich vývoji odhalit případné skryté chyby v knihovně *libipc* a určitý počet chyb v knihovně byl skutečně zachycen až při vývoji jednotlivých bloků. U knihovny *libipc* by provádění vektorových testů pravděpodobně mnoho nových informací neposkytlo a nebylo by příliš k užitku.

V průběhu vývoje jednotlivých bloků je třeba, aby každý implementátor testoval specifické činnosti bloku samostatně a nezávisle na ostatních, neboť činnosti prováděné jednotlivými bloky se značně liší. Jelikož jsou však všechny bloky navzájem propojené, je vhodné testovat jejich interakci již v průběhu vývoje. Taková je situace například u bloku jádra. Jelikož je tento blok nadřazený většině ze všech bloků, jsou při jeho testování současně testovány i jiné bloky. Blok řízení chování, implementovaný v rámci této diplomové práce, byl v průběhu vývoje samozřejmě testován také. Po dokončení objektu kontroléru byla ověřena schopnost bloku kritické komunikace s blokem jádra a schopnost úspěšně provést registrace u tohoto bloku. Poté byla ověřena, s využitím ladícího režimu bloku, schopnost objektu kontroléru vytvářet a spouštět všechny ostatní objekty bloku a bylo prověřeno, zda každý spuštěný blok skutečně běží. Nakonec byla ověřena schopnost reakce na externí změny stavu prostřednictvím RT signálů. Dále byla prověřena funkčnost objektu chybového kolektoru a schopnost objektu komunikátoru přijímat provozní meziblokové zprávy, vytvářet na jejich základě meziobjektové zprávy a ty odesílat do příslušných objektů a vytvářet na základě příchozích meziobjektových zpráv odpovídající zprávy provozní komunikace. Tyto objekty však byly prověřeny pouze prostřednictvím triviálních testů. Nakonec byla prověřena schopnost řadiče stavových automatů provádět kroky chodu stavových automatů, ale pouze v základním měřítku. Nebyl proveden test čekání na dokončení poslední žádané akce, ani test funkčnosti synchronizací a další podobné prvky. Prověřen zatím nebyl ani systém rozvrhování bloku plánovače. V budoucí fázi vývoje je tedy nezbytné nejprve dokončit testování veškerých neotestovaných prvků v bloku řízení. Po kompletním

dokončení bloku a po dokončení testování jeho součástí bude nutné blok otestovat v kooperaci s jinými bloky, například bloky jádra, řízení databáze a bloků downlink rozhraní.

U bloků je také po jejich dokončení vhodné provést tzv. vektorové testy. Ty by prověřily korektnost chování bloku při reakcích na požadavky a také v případě jeho přetížení. Tyto testy by byly automatizované a testovací aplikace by náhodně generovala trojce dotaz – časový limit – očekávaná odpověď. Testovací aplikace bude přijímat odpovědi od bloku, bude je porovnávat s očekávanými a bude zaznamenávat případy, kdy se neshodovaly. V testu přetížení bude testovací aplikace nadměrně zahlcovat blok požadavky. Pokud budou všechny ochranné mechanismy v bloku funkční, blok by měl zareagovat přechodem do chybového stavu. Takto by mělo být softwarové vybavení koncentrátoru testováno i jako funkční celek. V případě bloků, které využívají ke svému chodu specifické hardwarové prostředky, je také nutné otestovat jejích chování v případě selhání těchto prostředků. Vektorové testy a simulace selhání budou bezpochyby důležitou fází vývoje, která bude následovat po dokončení většiny součástí softwarového vybavení koncentrátoru. V současné fázi vývoje je třeba spoléhat se na jednoduché testy prováděné v průběhu vývoje daných součástí (bloků).

Samostatnou kapitolou testování je detekce paměťových úniků. Blok řízení chování byl v průběhu vývoje pravidelně testován pomocí nástroje *valgrind*. V tomto bloku byl detekován jediný paměťový únik, nicméně ten v současné době pravděpodobně není možné odstranit, neboť je zřejmě způsoben vnitřní chybou knihovny *libxml2*. Nejedná se však o veliký únik (42 bajt) a tedy ho není potřeba akutně vyřešit. V rámci budoucích testů je důležité pečlivě paměťové úniky hlídat u všech vyvíjených bloků.

# 4 Závěr

V rámci této diplomové práce byla navržena struktura softwarového vybavení datového koncentrátoru CAM3600 společnosti ZPA Smart Energy Trutnov a.s.. Aby byly splněny požadavky na jeho snadnou rozšiřitelnost a univerzálnost, byla navržena struktura využívající systém hierarchicky propojených součástí, tzv. bloků. Celé softwarové vybavení tedy bylo rozděleno na blok jádra zodpovídající za bezproblémový chod celého koncentrátoru, blok řízení databáze, bloky rozhraní k sítím měřidel (tzv. downlink rozhraní), bloky rozhraní k zákaznickým sítím (tzv. uplink rozhraní), bloky konzolového a webového rozhraní a blok řízení chování zajišťující požadovanou podporu standardu DLMS/COSEM a konfigurovatelnou autonomii celého koncentrátoru. Dále byly navrženy meziblokové komunikační procedury respektující navrženou hierarchickou strukturu a byly vytvořeny konkrétní návrhy struktur jednotlivých bloků, s výjimkou bloků uplink rozhraní, jejichž návrh byl znemožněn absencí specifikací požadavků na rozhraní k zákaznickým sítím.

Další částí diplomové práce byla implementace sdílené knihovny pro meziblokovou komunikaci. Tato knihovna, označována jako *libipc*, poskytuje jednotné prostředky pro realizaci navržených komunikačních procedur a bude využita všemi bloky softwarového vybavení koncentrátoru.

Poslední částí této diplomové práce byla implementace bloku řízení chování. Ten byl rozdělen na několik součástí – kontrolér, chybový kolektor, DLMS klient, komunikátor, řadič stavových automatů a plánovač. Kontrolér se stará o bezchybný chod bloku a chybový kolektor zajišťuje sběr případných chyb. Blok DLMS klienta nebyl v rámci této diplomové práce implementován z důvodu chybějící specifikace a absence klientské aplikace standardu DLMS/COSEM. Komunikátor bloku zajišťuje jeho spojení s ostatními bloky softwarového vybavení. Nejdůležitějšími součástmi bloku jsou však řadič stavových automatů a plánovač. Řadič stavových automatů zajišťuje autonomnost datového koncentrátor využíváním systému programovatelných stavových automatů. Ten je založen na systému pro návrh, verifikaci a popis časovaných stavových automatů UPPAAL. Tyto stavové automaty jsou definovány prostřednictvím XML souboru a mohou jednak reagovat na různé události vznikající jak v koncentrátoru samotném, tak i v připojených sítích a také mohou vyvolávat provádění určitých akcí. Plánovač bloku řízení chování pak tyto vyvolané akce přebírá a s využitím systému prioritních FCFS front je rozděluje na jednotlivé zdroje, kterými jsou zejména jednotlivá rozhraní k sítím. Právě úpravou onoho definičního souboru stavových automatů lze snadno změnit autonomní chování koncentrátoru dle určité zákaznické specifikace, čímž je splněn zmíněný požadavek na konfigurovatelnou autonomii.

V rámci projektu vývoje zmíněného datového koncentrátoru jsem byl pověřen vedením tříčlenného týmu vývoje softwarového vybavení. Bylo tedy mým úkolem zajistit povědomí členů týmu o aktuální verzi návrhu, vybrat vhodné prostředky týmové kooperace, tyto prostředky spravovat a v neposlední řadě také zajišťovat, že veškerý vývoj softwarového vybavení pokračuje v souladu s vytvořeným návrhem.

V dalších fázích vývoje bude vhodné nejprve vylepšit některé z funkcí knihovny *libipc*. Jedná se zejména o potenciálně blokující funkce knihovny, které však mohou volající vlákno blokovat v krajním případě i nekonečně dlouho. To není, z hlediska odolnosti proti chybám, žádoucí a tedy bude více než

vhodné funkce předělat tak, aby blokovaly s nastavitelným časovým limitem. Po reimplementaci těchto funkcí bude nutné celou knihovnu opět důkladně otestovat.

Z hlediska bloku řízení chování je nezbytné nejprve provést důkladnější testy dosud implementovaných součástí a poté dokončit implementaci objektu DLMS klienta. Pak bude možné nadefinovat typy událostí a akcí, které bude možné využít při programování autonomie prostřednictvím stavových automatů. Dále budou implementovány třídy pro reprezentaci těchto událostí a akcí. Také bude nutné rozšířit množinu zdrojů plánovače bloku řízení chování. Veškeré tyto nové součásti bloku bude nezbytné důkladně otestovat. Po dokončení bloku řízení chování budou v rámci projektu pokračovat práce na ostatních blocích softwarového vybavení, zejména na blocích řízení databáze a blocích downlink rozhraní. Také bude nutné specifikovat požadavky na podporu zákaznických sítí, aby bylo možné navrhnout konkrétní strukturu uplink bloků a tyto bloky poté implementovat.

# Reference

1. **DLMS User Association.:** *DLMS/COSEM Architecture and Protocols Specification*, 2010

2. *Linux Documentation*, http://www.linux.die.net, [Online]

3. **David A. Wheeler:** *http://tldp.org/HOWTO/Program-Library-HOWTO/index.html, Program library HOWTO.* [Online] 2000

4. **Sven Goldt:** *http://www.tldp.org/LDP/lpg/node7.html, Linux Interprocess Communications.* [Online] 1995

5. **Gerd Behrmann, Alexandre David, and Kim G. Larsen:** *http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, A Tutorial on Uppaal 4.0.* [Online] 2006

6. **PRIME Alliance:** *http://www.prime-alliance.org/wp-content/uploads/2013/03/MAC_Spec_white_paper_1_0_080721.pdf, PRIME Technology Whitepaper.* [Online] 2008

# Příloha A

Zkrácená verze hlavičkového souboru *libipc.h*. Úplná verze je dostupná ve formě elektronické přílohy.

```
//! Message severities
typedef enum {
  IPC_MSGQ_MSG_SEV_CRITICAL    = 01,           //!< Critical messages
  IPC_MSGQ_MSG_SEV_NORMAL                      //!< Normal messages
} T_IPC_MSGQ_MSG_SEVERITY;

//! Channel types
typedef enum {
  IPC_MSGQ_CHANNEL_TW,                         //!< Two-way message queue channel
  IPC_MSGQ_CHANNEL_OW                          //!< One-way message queue channel
} T_IPC_MSGQ_CHANNEL_TYPE;

//! Message header
typedef struct __attribute__ ((__packed__)) {
  long      sev;                    //!< Severity (critical or normal)
  uint8_t   ver;                    //!< Version of used protocol
  struct {
    pid_t   pid;                    //!< Block PID
    uint8_t sid;                    //!< Session ID
  }         cid;                    //!< Connection ID
  struct {
    uint8_t type;                   //!< Type
    uint8_t subtype;                //!< Subtype
  }         content;                //!< Content identification
  uint8_t   prior;                  //!< Priority
  uint16_t  size;                   //!< Number of valid bytes in message body
  uint8_t   reserved[4];            //!< Reserved for future use
} T_IPC_MSGQ_MSG_HEADER;

//! Message header size
#define IPC_MSGQ_MSG_HEADER_SIZE (sizeof(T_IPC_MSGQ_MSG_HEADER) - sizeof(long))

//! Message body maximum size
#define IPC_MSGQ_MSG_DATA_MAX_SIZE 256

//! Message total maximum size
#define IPC_MSGQ_MSG_TOTAL_MAX_SIZE (IPC_MSGQ_MSG_HEADER_SIZE +
       IPC_MSGQ_MSG_DATA_MAX_SIZE)

//! Message
typedef struct __attribute__ ((__packed__)) {
  T_IPC_MSGQ_MSG_HEADER head;                                  //!< Header
  uint8_t               data[IPC_MSGQ_MSG_DATA_MAX_SIZE];      //!< Body
} T_IPC_MSGQ_MSG;

//! Message queue communication channel
typedef struct {
  union {
    struct {
      int                id_rx;       //!< Rx message queue ID (input)
      int                id_tx;       //!< Tx message queue ID (output)
    }                    tw;          //!< Two-way message queue channel
    struct {
      int                id;          //!< Message queue ID
      int                reserved;    //!< Reserved for future use
    }                    ow;          //!< One-way message queue channel
  };
  T_IPC_MSGQ_CHANNEL_TYPE type;       //!< Channel type (two-way, one-way)
} T_IPC_MSGQ_CHANNEL;
```

```
////////////////////////////////////////////////////////////////////////////////
/// Creates two-way message queue communication channel
/// @param  ap_channel = Pointer to communication channel
/// @param  a_rx_key = Rx queue key
/// @param  a_tx_key = Tx queue key
/// @return Returns 0 on success, -1 otherwise
////////////////////////////////////////////////////////////////////////////////
int ipc_msgq_create_tw(T_IPC_MSGQ_CHANNEL *ap_channel, key_t a_rx_key, key_t a_tx_key);

////////////////////////////////////////////////////////////////////////////////
/// Creates one-way message queue communication channel
/// @param  ap_channel = Pointer to communication channel
/// @param  a_key = Queue key
/// @return Returns 0 on success, -1 otherwise
////////////////////////////////////////////////////////////////////////////////
int ipc_msgq_create_ow(T_IPC_MSGQ_CHANNEL *ap_channel, key_t a_key);

////////////////////////////////////////////////////////////////////////////////
/// Destroys message queue communication channel
/// @param  ap_channel = Pointer to communication channel
/// @return Nothing
////////////////////////////////////////////////////////////////////////////////
void ipc_msgq_destroy(T_IPC_MSGQ_CHANNEL *ap_channel);

////////////////////////////////////////////////////////////////////////////////
/// Sends message using message queue communication channel
/// @param  ap_channel = Pointer to communication channel
/// @param  ap_message = Pointer to message
/// @param  ab_block = TRUE, if sending can block
/// @param  a_timeout = Timeout in milisecond (ignored, if @ab_block is TRUE)
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 22 (EINVAL) - Queue was destroyed and its ID is invalid
////////////////////////////////////////////////////////////////////////////////
int ipc_msgq_snd(T_IPC_MSGQ_CHANNEL *ap_channel, const T_IPC_MSGQ_MSG *ap_message,
       int ab_block, unsigned a_timeout);

////////////////////////////////////////////////////////////////////////////////
/// Receives message of specified severity using message queue communication channel
/// @param  ap_channel = Pointer to communication channel
/// @param  a_severity = Message severity
/// @param  ap_buffer = Pointer to buffer for received message
/// @param  ab_block =  TRUE, if receiving can block
/// @return Returns number of received bytes minus size of long
///         (return value is 4 less then real content in @ap_buffer)
///         on success, -1 otherwise<BR>
///         errno = 22 (EINVAL)- Queue was destroyed and its ID is invalid<BR>
///         errno = 42 (ENOMSG)- No valid messages in message queue to receive)<BR>
///         errno =  7 (E2BIG) - Received message is larger then @ap_buffer
////////////////////////////////////////////////////////////////////////////////
int ipc_msgq_rcv(T_IPC_MSGQ_CHANNEL *ap_channel, T_IPC_MSGQ_MSG_SEVERITY a_severity,
       T_IPC_MSGQ_MSG *ap_buffer, int ab_block);

//! Semaphores set
typedef struct {
  int      id;                          //!< Set ID
  unsigned size;                        //!< Set size (number of semaphores in set)
} T_IPC_SEMS_SET;
```

```
////////////////////////////////////////////////////////////////////////////////
/// Creates semaphores set
/// @param  ap_set = Pointer to semaphores set
/// @param  a_key = Semaphores set key
/// @param  a_size = Set size (number of semaphores in set)
/// @return Returns 0 on success, -1 otherwise
////////////////////////////////////////////////////////////////////////////////
int ipc_sems_create(T_IPC_SEMS_SET *ap_set, key_t a_key, unsigned a_size);

////////////////////////////////////////////////////////////////////////////////
/// Destroys semaphores set
/// @param  ap_set = Pointer to semaphores set
/// @return Nothing
////////////////////////////////////////////////////////////////////////////////
void ipc_sems_destroy(T_IPC_SEMS_SET *ap_set);

////////////////////////////////////////////////////////////////////////////////
/// Gets value of specific semaphore in specific set
/// @param  ap_set = Pointer to semaphores set
/// @param  a_sem_num = Index of semaphore in set
/// @return Returns semaphore value on success, -1 otherwise<BR>
///         errno = 22 (EINVAL)- Semaphore set was destroyed and its ID is invalid
////////////////////////////////////////////////////////////////////////////////
int ipc_sems_get_val(T_IPC_SEMS_SET *ap_set, unsigned a_sem_num);

////////////////////////////////////////////////////////////////////////////////
/// Sets value of specific semaphore in specific set
/// @param  ap_set = Pointer to semaphores set
/// @param  a_sem_num = Index of semaphore in set
/// @param  a_val = New value of semaphore
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 22 (EINVAL)- Semaphore set was destroyed and its ID is invalid
////////////////////////////////////////////////////////////////////////////////
int ipc_sems_set_val(T_IPC_SEMS_SET *ap_set, unsigned a_sem_num, int a_val);

////////////////////////////////////////////////////////////////////////////////
/// Takes value from specific semaphore in specific set
/// @param  ap_set = Pointer to semaphores set
/// @param  a_sem_num = Index of semaphore in set
/// @param  a_val = Taken value
/// @param  ab_block = TRUE, if taking can block
/// @param  a_timeout = Timeout in miliseconds (ignored, if @ab_block is FALSE)
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 22 (EINVAL)- Semaphore set was destroyed and its ID is invalid<BR>
///         errno = 11 (EAGAIN)- Taking would block or timeout has expired
////////////////////////////////////////////////////////////////////////////////
int ipc_sems_take(T_IPC_SEMS_SET *ap_set, unsigned a_sem_num, unsigned a_val,
      int ab_block, unsigned a_timeout);

////////////////////////////////////////////////////////////////////////////////
/// Gives value to specific semaphore in specific set
/// @param  ap_set = Pointer to semaphores set
/// @param  a_sem_num = Index of semaphore in set
/// @param  a_val = Given value
/// @param  ab_block = TRUE, if giving can block
/// @param  a_timeout = Timeout in miliseconds (ignored, if @ab_block is FALSE)
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 22 (EINVAL)- Semaphore set was destroyed and its ID is invalid<BR>
///         errno = 11 (EAGAIN)- Giving would block or timeout has expired
////////////////////////////////////////////////////////////////////////////////
int ipc_sems_give(T_IPC_SEMS_SET *ap_set, unsigned a_sem_num, unsigned a_val,
      int ab_block, unsigned a_timeout);
```

```
//! RT signals
typedef enum {
  IPC_RTSIG_SIG_FORCE_STATE    = 35                              //!< Force state signal
} T_IPC_RTSIG_SIGNAL;

//! RT signal information
typedef siginfo_t T_IPC_RTSIG_SIG_INFO;

//! RT signal handler
typedef void( *T_IPC_RTSIG_HANDLER)(int, T_IPC_RTSIG_SIG_INFO *, void *);

//! RT signal data
typedef union sigval T_IPC_RTSIG_DATA;

////////////////////////////////////////////////////////////////////////////////
/// Sets a RT signal handler to specific function
/// @param  a_sig_num = RT signal
/// @param  a_sig_handler = Pointer to function
/// @return Returns 0 on success, -1 otherwise
////////////////////////////////////////////////////////////////////////////////
int ipc_rtsig_set_handler(T_IPC_RTSIG_SIGNAL a_sig, T_IPC_RTSIG_HANDLER a_sig_handler);

////////////////////////////////////////////////////////////////////////////////
/// Sends RT signal and data to a specific process
/// @param  a_sig_num = RT signal
/// @param  a_target = PID of target process
/// @param  a_data = Sent data
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 3 (ESRCH)- Process with specified PID was not found
////////////////////////////////////////////////////////////////////////////////
int ipc_rtsig_snd(T_IPC_RTSIG_SIGNAL a_sig, pid_t a_target, T_IPC_RTSIG_DATA a_data);

//! Named pipes channel modes
typedef enum {
  IPC_NPIPES_MODE_READ,                                          //!< Reading mode
  IPC_NPIPES_MODE_WRITE                                          //!< Writing mode
} T_IPC_NPIPES_MODE;

//! Named pipes communication channel
typedef struct {
  char *            psz_name;          //!< Name of channel
  int               fd;                //!< File descriptor
  T_IPC_NPIPES_MODE mode;              //!< Channel mode (reading or writing)
  int               b_block;           //!< TRUE, if operations with channel can block
  T_IPC_SEMS_SET    sync;              //!< Synchronization semaphore
} T_IPC_NPIPE_CHANNEL;

////////////////////////////////////////////////////////////////////////////////
/// Creates named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @param  apsz_name = Channel name
/// @param  a_mode = Channel mode
/// @param  a_key_sync = Synchronization semaphore key
/// @param  ab_block = TRUE, if operations on channel can block
/// @return Returns 0 on success, -1 otherwise
////////////////////////////////////////////////////////////////////////////////
int ipc_npipe_create(T_IPC_NPIPE_CHANNEL *ap_channel, char *apsz_name,
       T_IPC_NPIPES_MODE a_mode, key_t a_key_sync, int ab_block);
```

```
////////////////////////////////////////////////////////////////////////////////
/// Opens named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 6 (ENXIO)- No process has this named pipe channel created
////////////////////////////////////////////////////////////////////////////////
int ipc_npipe_open(T_IPC_NPIPE_CHANNEL *ap_channel);

////////////////////////////////////////////////////////////////////////////////
/// Closes named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @return Nothing
////////////////////////////////////////////////////////////////////////////////
void ipc_npipe_close(T_IPC_NPIPE_CHANNEL *ap_channel);

////////////////////////////////////////////////////////////////////////////////
/// Destroys named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @return Nothing
////////////////////////////////////////////////////////////////////////////////
void ipc_npipe_destroy(T_IPC_NPIPE_CHANNEL *ap_channel);

////////////////////////////////////////////////////////////////////////////////
/// Waits for notification on named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @param  a_timeout = Timeout in miliseconds
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 11 (EAGAIN)- Waiting would block
///                              or timeout has expired
////////////////////////////////////////////////////////////////////////////////
int ipc_npipe_wait(T_IPC_NPIPE_CHANNEL *ap_channel, unsigned a_timeout);

////////////////////////////////////////////////////////////////////////////////
/// Sends notification for named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @param  a_timeout = Timeout in miliseconds
/// @return Returns 0 on success, -1 otherwise<BR>
///         errno = 11 (EAGAIN)- Notifying would block
///                              or timeout has expired
////////////////////////////////////////////////////////////////////////////////
int ipc_npipe_notify(T_IPC_NPIPE_CHANNEL *ap_channel, unsigned a_timeout);

////////////////////////////////////////////////////////////////////////////////
/// Sends data using named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @param  ap_data = Pointer to data
/// @param  a_num = Number of sent bytes
/// @return Returns number of sent bytes on success, -1 otherwise<BR>
///         errno = 9 (EBADF)- Named pipe channel is not opened<BR>
///         errno = 11 (EAGAIN)- Sending would block
////////////////////////////////////////////////////////////////////////////////
int ipc_npipe_snd(T_IPC_NPIPE_CHANNEL *ap_channel, const void *ap_data, unsigned a_num);

////////////////////////////////////////////////////////////////////////////////
/// Receives data using named pipe communication channel
/// @param  ap_channel = Pointer to channel
/// @param  ap_buf = Pointer to buffer for data
/// @param  a_num = Number of received bytes
/// @return Returns number of received bytes on success, -1 otherwise<BR>
///         errno = 9 (EBADF)- Named pipe channel is not opened<BR>
///         errno = 11 (EAGAIN)- Receiving would block
////////////////////////////////////////////////////////////////////////////////
int ipc_npipe_rcv(T_IPC_NPIPE_CHANNEL *ap_channel, void *ap_buf, unsigned a_num);
```

# Příloha B

Zkrácený obsah hlavičkových souborů *PQueSys.h* a *Scheduler.h*. Úplné verze těchto souborů jsou dostupné ve formě elektronických příloh.

### *PQueSys.h*

```
//! Priority queues system maximum priority
#define PQUESYS_MAX_PRIORITY    5

//! Priority queues system maximum queue size
#define PQUESYS_MAX_QUEUE_SIZE  (1000 / PQUESYS_MAX_PRIORITY)
////////////////////////////////////////////////////////////////////////////////
/// Class for representation of priority queues system.
////////////////////////////////////////////////////////////////////////////////
class PQueSys
{
public:
    static unsigned g_sessionIDCnt;                  //!< Global session ID counter

    //! Structure representing single action plan
    struct ActionPlan {
        Action *p_action;      //!< Pointer to planned action
        bool b_issued;         //!< True, if action plan has been issued
        bool b_finished;       //!< True, if action plan has been finished
        bool b_succeeded;      //!< True, if action plan has succeeded
        time_t timeStamp;      //!< Time stamp of moment, that action has been requested
        unsigned sessionID;    //!< Action plan session ID

        ////////////////////////////////////////////////////////////////////////
        /// Creates new specific single action plan
        /// @param ap_action = Pointer to planned action
        /// @param a_timeStamp = Action issuing moment time stamp
        /// @param a_sessionID = Action issuing session ID
        ////////////////////////////////////////////////////////////////////////
        ActionPlan(Action *ap_action, time_t a_timeStamp, unsigned a_sessionID = 0)
            : p_action(ap_action), b_issued(false), b_finished(false),
                    b_succeeded(false), timeStamp(a_timeStamp), sessionID(a_sessionID);
    };
private:
    array<deque<ActionPlan *>,
            PQUESYS_MAX_PRIORITY + 1> pqSys; //!< Array of queues for action plans
    ActionPlan *p_issuedPlan;                //!< Pointer to last issued action plan
public:
    ////////////////////////////////////////////////////////////////////////////
    /// Creates priority queues system
    ////////////////////////////////////////////////////////////////////////////
    PQueSys(void);

    ////////////////////////////////////////////////////////////////////////////
    /// Submits specific single action plan into priority queues system
    /// @param  ap_actionPlan = Pointer to action plan
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////
    bool SubmitActionPlan(ActionPlan *ap_actionPlan);

    ////////////////////////////////////////////////////////////////////////////
    /// Performs planning operations on priority queues system
    /// @param  ap_issuedPlan = Pointer to issued action plan (if any)
    /// @return Returns true, when new action has been issued, false otherwise
    ////////////////////////////////////////////////////////////////////////////
    bool Check(ActionPlan *&ap_issuedPlan);
```

```
};
```

### *Scheduler.h*

```
//! Number or scheduler's resources
#define SCHEDULER_RESOURCES_CNT     1

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
/// Class for representation of scheduler object
/// @inherit ThreadedObject
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
class Scheduler :
    public ThreadedObject
{
    //! Enumeration of resource types
    enum Resource {
        ResourcePCLProxy1      = 0    //!< PLC proxy 1 resource
    };

    //! Structure representing action request
    struct ActionRequest {
        Action *p_action;              //!< Pointer to planned action
        bool b_finished;               //!< True, if action request has been finished
        bool b_success;                //!< True, if action request has been successfull

        //! Report back configuration
        struct ReportBackConfig {
            bool b_enabled;            //!< True, if action's result should be reported

            //! Action result types
            enum ActResultType {
                ActResultSuccess,      //!< Action success result
                ActResultFailure       //!< Action failure result
            } type;                    //!< Action result type
        } reportBack;                  //!< Report back configuration
        unsigned sessionID;            //!< Action issuing session ID
        string autName;                //!< Issuing automaton name
        vector<PQueSys::ActionPlan> plans;   //!< Vector of action plans of request

        ////////////////////////////////////////////////////////////////////////
        /// Creates new specific single action request
        /// @param ap_action = Pointer to planned action
        /// @param a_sessionID = Action issuing session ID
        /// @param a_autName = Issuing automaton name
        ////////////////////////////////////////////////////////////////////////
        ActionRequest(Action *ap_action = nullptr, unsigned a_sessionID = 0,
                      string a_autName = "")
            : p_action(ap_action), b_finished(false), b_success(false),
                    reportBack({ ap_action->b_reportBack,
                    ap_action->b_reportSuccess ? ReportBackConfig::ActResultSuccess
                    : ReportBackConfig::ActResultFailure }), sessionID(a_sessionID),
                    autName(a_autName);
    };

    array<PQueSys, SCHEDULER_RESOURCES_CNT> priorQueSystems; //!< Priority queue systems
    vector<ActionRequest> pendingRequests;        //!< Vector of pending action requests
    Buffer<ControlMsg *> inBuffer;                //!< Buffer for incoming messages
    Communicator *p_commHandle;                   //!< Communicator handle
    AutomataDriver *p_autDriverHandle;            //!< Automata driver handle
    DLMSClient *p_dlmsClientHandle;               //!< DLMS client handle
```

```
    ////////////////////////////////////////////////////////////////////////////////
    /// Handles incoming messages, processes them and create outcoming messages
    /// (executed by thread of scheduler object)
    /// @return Returns pointer to returned data
    ////////////////////////////////////////////////////////////////////////////////
    virtual void *ThreadFunction(void);

    ////////////////////////////////////////////////////////////////////////////////
    /// Parses specific incoming message and creates outcoming message from it
    /// @param  ap_inMsg = Parsed incoming message
    /// @param  ap_outMsg = Created outcoming message
    /// @return Returns true, if outgoing message was created, false otherwise
    ////////////////////////////////////////////////////////////////////////////////
    bool ParseMessage(ControlMsg *&ap_inMsg, ControlMsg *&ap_outMsg);

    ////////////////////////////////////////////////////////////////////////////////
    /// Checks priority queue systems for all resources and performs scheduling
    /// operations on them
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////////
    void Schedule(void);

    ////////////////////////////////////////////////////////////////////////////////
    /// Adds specific action request into scheduling system
    /// @param  ap_actRequestMsg = Pointer to action request message
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////////
    bool AddActionRequest(ActRequestControlMsg *ap_actRequestMsg);

    ////////////////////////////////////////////////////////////////////////////////
    /// Issues specific action.
    /// @param  ap_action = Pointer to issued
    /// @param  a_sessionID = Session ID
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////////
    void IssueAction(Action *ap_action, unsigned a_sessionID);
public:
    ////////////////////////////////////////////////////////////////////////////////
    /// Creates new scheduler object with all necessary objects
    /// @param a_blockTimeout = Blocking operations timeout in miliseconds
    /// @throw Error message string on fatal failure
    ////////////////////////////////////////////////////////////////////////////////
    Scheduler(unsigned a_blockTimeout = DEFAULT_BLOCK_TIMEOUT);

    ////////////////////////////////////////////////////////////////////////////////
    /// Destroys scheduler object and releases all used objects
    ////////////////////////////////////////////////////////////////////////////////
    virtual ~Scheduler(void);

    ////////////////////////////////////////////////////////////////////////////////
    /// Sets handle to communicator object
    /// @param  ap_commHandle = Communicator object handle
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////////
    void SetCommHandle(Communicator *ap_commHandle);

    ////////////////////////////////////////////////////////////////////////////////
    /// Sets handle to automata driver object
    /// @param  ap_autDriverHandle = Automata driver object handle
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////////
    void SetAutDriverHandle(AutomataDriver *ap_autDriverHandle);
```

```
    ////////////////////////////////////////////////////////////////////////////////
    /// Sets handle to DLMS client object
    /// @param  ap_dlmsClientHandle = DLMS client object handle
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////////
    void SetDLMSClientHandle(DLMSClient *ap_dlmsClientHandle);

    ////////////////////////////////////////////////////////////////////////////////
    /// Queries specific message to be processed by scheduler object
    /// @param  ap_msg = Queried message
    /// @param  a_timeOut = Calling thread blocking timeout
    /// @throw  Error message string on fatal failure
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////////
    bool QueryMsg(ControlMsg *ap_msg, int a_timeOut);
};
```

# Příloha C

Zkrácený obsah hlavičkových souborů *Automata.h* a *AutomataDriver.h*. Úplné verze těchto hlavičkových souborů jsou dostupné ve formě elektronických příloh.

### *Automata.h*

```
////////////////////////////////////////////////////////////////////////////////
/// Class for representation of transition synchronization action
////////////////////////////////////////////////////////////////////////////////
class SyncAction
{
public:
    const string str;           //!< Synchronization action string
    string name;                //!< Synchronization variable or event name
    bool b_isEvent;             //!< True, if synchronization action is event
    bool b_isWait;              //!< True, if synchronization action is waiting
    bool b_waitsForFailure;     //!< True, if waiting is for action failure
    bool b_waitsForSuccess;     //!< True, if waiting is for action success
    bool b_setSync;             //!< True, if synchronization variable is set by action

    ////////////////////////////////////////////////////////////////////////////
    /// Creates new transition synchronization action with specific string
    /// @param a_str = Synchronization action string
    ////////////////////////////////////////////////////////////////////////////
    SyncAction(string a_str = "")
        : str(a_str), b_isEvent(false), b_isWait(false), b_waitsForFailure(false),
                b_waitsForSuccess(false), b_setSync(false);
};


////////////////////////////////////////////////////////////////////////////////
/// Class for representation of transition guard condition
////////////////////////////////////////////////////////////////////////////////
class GuardCond
{
public:
    string str;                         //!< Guard condition string

    ////////////////////////////////////////////////////////////////////////////
    /// Creates new transition guard condition with specific string
    /// @param a_str = Guard condition string
    ////////////////////////////////////////////////////////////////////////////
    GuardCond(string a_str = "");
};

////////////////////////////////////////////////////////////////////////////////
/// Class for representation of transition assignment operation
////////////////////////////////////////////////////////////////////////////////
class AssignOp
{
public:
    const string str;                       //!< Assignment operation string

    //! Variable assignment
    struct VarAssign
    {
        string name;                        //!< Variable name
        string expr;                        //!< Variable assigned expression
        int value;                          //!< Evaluated value
```

```
        ////////////////////////////////////////////////////////////////////////
        /// Creates new variable assignment
        /// @param a_name = Assigned variable name
        /// @param a_expr = Assign expression
        ////////////////////////////////////////////////////////////////////////
        VarAssign(string a_name, string a_expr)
            : name(a_name), expr(a_expr), value(0);
    };

    vector<VarAssign> varsAssigns;          //!< Vector of variables assignments
    vector<Action *> actions;               //!< Vector of pointers to executed actions

    ////////////////////////////////////////////////////////////////////////
    /// Creates new transition assignment operation with specific string
    /// @param a_str = Assignment operation string
    ////////////////////////////////////////////////////////////////////////
    AssignOp(string a_str = "") : str(a_str);
};

////////////////////////////////////////////////////////////////////////
/// Class for representation of automaton transition
////////////////////////////////////////////////////////////////////////
class Transition
{
public:
    const State &source;                                //!< Source state
    const State &sink;                                  //!< Sink state
    const SyncAction sync;                              //!< Synchronization action
    const GuardCond guard;                              //!< Guard condition
    const AssignOp assign;                              //!< Assignment operation

    ////////////////////////////////////////////////////////////////////////
    /// Creates new transition
    /// @param a_source = Source state
    /// @param a_sink = Sink state
    /// @param a_sync = Synchronization action string
    /// @param a_guard = Guard condition string
    /// @param a_assign = Assign string
    ////////////////////////////////////////////////////////////////////////
    Transition(const State &a_source, const State &a_sink,
            string a_sync, string a_guard, string a_assign)
        : source(a_source), sink(a_sink),
                sync(a_sync), guard(a_guard), assign(a_assign);
};


////////////////////////////////////////////////////////////////////////
/// Class for representation of automaton state
////////////////////////////////////////////////////////////////////////
class State
{
public:
    const string id;                            //!< State ID
    time_t timeStamp;                           //!< Last state entrance time stamp
    vector<Transition *> transitions;           //!< Vector of sourcing transitions

    ////////////////////////////////////////////////////////////////////////
    /// Creates new state with specefic ID
    /// @param a_id = State ID
    ////////////////////////////////////////////////////////////////////////
    State(string a_id) : id(a_id), timeStamp(0);
};
```

```
////////////////////////////////////////////////////////////////////////////////
/// Class for representation of automaton
////////////////////////////////////////////////////////////////////////////////
class Automaton
{
public:
    string name;                    //!< Automaton name
    bool b_loaded;                  //!< True, if automaton was sucessfully loaded
    bool b_running;                 //!< True, if automaton is running
    bool b_stateChanged;            //!< True, if automaton has recently changed state
    struct {
        unsigned sessionID;         //!< Session ID of last issued action's request
        string descr;               //!< Descriptor of last issued action
        bool b_finished;            //!< True, if last issued action was finished
        bool b_success;             //!< True, if last issued action was successfull
    } lastIssuedAct;
    State *p_initState;             //!< Pointer to initial state
    State *p_currState;             //!< Pointer to current state
    vector<State> states;                                   //!< Vector of states
    vector<Transition> transitions;                         //!< Vector of transitions
    unordered_set<TimedVar, TimedVar::Hasher> timedVars;    //!< Timed variables
    unordered_set<IntVar, TimedVar::Hasher> intVars;        //!< Integer variables
    unordered_set<BoolVar, TimedVar::Hasher> boolVars;      //!< Boolean variables

    ////////////////////////////////////////////////////////////////////////////
    /// Creates new unloaded automaton
    ////////////////////////////////////////////////////////////////////////////////
    Automaton(void);

    ////////////////////////////////////////////////////////////////////////////////
    /// Clears automaton and mark it as unloaded
    ////////////////////////////////////////////////////////////////////////////////
    void Clear(void);

    ////////////////////////////////////////////////////////////////////////////////
    /// Updates all automaton's timed variables
    ////////////////////////////////////////////////////////////////////////////////
    void UpdateTimedVars(void);

    ////////////////////////////////////////////////////////////////////////////////
    /// Parses specific string for description of automaton timed variable
    /// @param  a_declaration = Declaration string of automaton timed variable
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////////
    bool ParseVariableDeclaration(string a_declaration);
};

////////////////////////////////////////////////////////////////////////////////
/// Struct for representation of automaton - transition pairing.
////////////////////////////////////////////////////////////////////////////////
struct AutTrans
{
    Automaton *p_aut;                                       //!< Pointer to automaton
    Transition *p_trans;                                    //!< Pointer to transition

    ////////////////////////////////////////////////////////////////////////////////
    /// Creates new automaton - transition pairing
    /// @param ap_aut = Pointer to automaton
    /// @param ap_trans = Pointer to transition
    ////////////////////////////////////////////////////////////////////////////////
    AutTrans(Automaton *ap_aut = nullptr, Transition *ap_trans = nullptr)
       : p_aut(ap_aut), p_trans(ap_trans);
};
```

### *AutomataDriver.h*

```
////////////////////////////////////////////////////////////////////////////////
/// Class for representation of automata driver object
/// @inherit ThreadedObject
////////////////////////////////////////////////////////////////////////////////
class AutomataDriver :
    public ThreadedObject
{
    string descrFilePath;                                  //!< Description file path
    vector<Automaton> automata;                            //!< Vector of automata
    unordered_set<SyncVar, SyncVar::Hasher> syncVars;      //!< Sync variables
    unordered_set<TimedVar, TimedVar::Hasher> timedVars; //!< Timed variables
    unordered_set<IntVar, TimedVar::Hasher> intVars;       //!< Integer variables
    unordered_set<BoolVar, TimedVar::Hasher> boolVars;     //!< Boolean variables
    vector<AutVar *> autVars;                 //!< Vector of pointers to all variables
    deque<Event *> events;                    //!< Double ended queue of incoming events
    Buffer<ControlMsg *> inBuffer;            //!< Buffer for incoming messages
    Scheduler *p_schedHandle;                 /!< Scheduler object handle
    Communicator *p_commHandle;               //!< Communicator object handle

    ////////////////////////////////////////////////////////////////////////////
    /// Handles incoming messages, processes them and create outcoming messages
    /// (executed by thread of automata driver object)
    /// @return Returns pointer to returned data
    ////////////////////////////////////////////////////////////////////////////
    virtual void *ThreadFunction(void);

    ////////////////////////////////////////////////////////////////////////////
    /// Loads automata and variables described in specific file
    /// @param  a_filePath = Description file path
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////
    bool LoadAutomata(string a_filePath);

    ////////////////////////////////////////////////////////////////////////////
    /// Adds automata described in specific description file into automata driver object
    /// @param  a_filePath = Description file path
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////
    bool ParseAutomata(string a_filePath);

    ////////////////////////////////////////////////////////////////////////////
    /// Parses specific incoming message and creates outcoming message from it
    /// @param  ap_inMsg = Parsed incoming message
    /// @param  ap_outMsg = Created outcoming message
    /// @return Returns true, if outgoing message was created, false otherwise
    ////////////////////////////////////////////////////////////////////////////
    bool ParseMessage(ControlMsg *&ap_inMsg, ControlMsg *&ap_outMsg);

    ////////////////////////////////////////////////////////////////////////////
    /// Checks and changes states of all running automata
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////
    void CheckAutomata(void);

    ////////////////////////////////////////////////////////////////////////////
    /// Updates all global variables
    ////////////////////////////////////////////////////////////////////////////
    void UpdateGlobalVars(void);
```

```
    ////////////////////////////////////////////////////////////////////////////
    /// Parses specific string for description of global automata variable
    /// @param  a_declaration = Declaration string of global automata variable
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////
    bool ParseVariableDeclaration(string a_declaration);
public:
    ////////////////////////////////////////////////////////////////////////////
    /// Creates new automata driver object with all necessary objects
    /// @param a_descrFilePath = Automata description file path
    /// @param a_blockTimeout = Blocking operations timeout in miliseconds
    /// @throw Error message string on fatal failure
    ////////////////////////////////////////////////////////////////////////////
    AutomataDriver(string a_descrFilePath, unsigned a_blockTimeout
            = DEFAULT_BLOCK_TIMEOUT);

    ////////////////////////////////////////////////////////////////////////////
    /// Destroys automata driver object and releases all used objects
    ////////////////////////////////////////////////////////////////////////////
    virtual ~AutomataDriver(void);

    ////////////////////////////////////////////////////////////////////////////
    /// Sets handle to scheduler object
    /// @param  ap_schedHandle = Scheduler object handle
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////
    void SetSchedHandle(Scheduler *ap_schedHandle);

    ////////////////////////////////////////////////////////////////////////////
    /// Sets handle to communicator object
    /// @param  ap_commHandle = Communicator object handle
    /// @return Nothing
    ////////////////////////////////////////////////////////////////////////////
    void SetCommHandle(Communicator *ap_commHandle);

    ////////////////////////////////////////////////////////////////////////////
    /// Queries specific message to be processed by automata driver object
    /// @param  ap_msg = Queried message
    /// @param  a_timeOut = Calling thread blocking timeout
    /// @throw  Error message string on fatal failure
    /// @return Returns true on success, false otherwise
    ////////////////////////////////////////////////////////////////////////////
    bool QueryMsg(ControlMsg *ap_msg, int a_timeOut);
};
```