FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Systém pro sběr dat s využitím OPC UA
Student:	Dominik Codl
Vedoucí:	Ing. Ladislav Šťastný, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem bakalářské práce je návrh a implementace systému pro automatizovaný sběr dat z měřících zařízení, která poskytují data skrze vlastní OPC UA servery. Práce se skládá ze dvou hlavních částí – aplikace pro sběr a její databáze.

Postup:

1. Analyzujte použitou technologii OPC UA, zejména si všimněte objektového návrhu informačního modelu a jeho realizace na serveru.

2. Vypracujte návrh a implementaci databáze, kde se budou ukládat data získaná systémem ze spravovaných zařízení.

3. Vypracujte návrh a implementaci aplikace přijímající skrze databázi požadavky na obsluhu a vyčítání dat ze spravovaných zařízení.

4. Zhodnoťte výsledek a navrhněte možná vylepšení do budoucna.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 23. listopadu 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Systém pro sběr dat s využitím OPC UA

Dominik Codl

Katedra softwarového inženýrství
Vedoucí práce: Ing. Ladislav Šťastný, Ph.D.

21. dubna 2022

Poděkování

Rád bych poděkoval své rodině a blízkým za doprovod na cestě za vzděláním, jejíž zastavkou z mnoha je i tato bakalářská práce. Rovněž děkuji své přítelkyni za podporu a trpělivost v době, kdy jsem trávil večery studiem materiálů a programováním.

Nakonec děkuji Ing. Ladislavovi Šťastnému, PhD. za odborné vedení a pomoc při zpracování mé bakalářské práce.

Prohlášení

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.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 21. dubna 2022 . . . . . .

Odkaz na tuto práci

Codl, Dominik. *Systém pro sběr dat s využitím OPC UA*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Bakalářská práce se věnuje tvorbě Systému pro sběr dat. Zařízení, odkud se shromažďují data, komunikují pomocí standardu OPC UA. Práce poskytuje seznámení se zmíněným standardem. Dále dělí řešení do dílčích projektů: databáze, knihovna pro OPC UA a aplikace. Každý z nich obsahuje návrh vytvořený na základě popisu domény a požadavků na Systém, implementaci a způsob testování.

Klíčová slova sběr dat, OPC UA, Scala, PostgreSQL, klient-server

Abstract

The bachelor thesis is dedicated to a development of the Data Collection System. Devices, from which data are collected, communicate using the OPC UA standard. The thesis provides information about the mentioned standard. Futhermore the solution is parted into subprojects: the database, the library for OPC UA and the application. Each of them contains a design based on the System requirements and the domain description, an implementation and testing methods.

Keywords data collection, OPC UA, Scala, PostgreSQL, client-server

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Neustálé zvyšování využití technologií sebou nese požadavky na infrastrukturu. Elektrické sítě se nacházejí prakticky všude a nikdo si již nedovede představit život bez elektrické energie. Ať už se jedná o domácnosti, firmy nebo města. S tím také rostou nároky na bezpečnost a korektní správu sítí, bezchybnou a efektivní komunikaci mezi síťovými komponentami. Je tedy nutná diagnostika, vyhodnocování stavu sítě, a zařízení k ní připojených. Ve zmíněném odvětví poskytuje svá řešení i česká společnost ModemTec.

Bakalářská práce se zabývá Systémem pro sběr dat pro firmu ModemTec. Zařízení, odkud jsou data získávána, používají komunikační standard OPC UA. Osudem nashromážděných dat je zpracování a využití k diagnostice. Práce je logicky členěna do následujících kapitol.

První kapitola uvádí cíl práce. Ten je rozveden do dílčích cílů. Každý z nich seznamuje čtenáře s odpovídajícím bodem ze zadání a přidává další popis či motivaci.

Druhá kapitola předkládá použité technologie, se kterými se lze setkat v částech věnujících se implementaci. Dané projekty jsou popsány především z hlediska principu a základních funkcionalit.

V třetí kapitole je popsán komunikační standard OPC UA, který staví na architektuře klient-server. Text klade důraz na nastínění informačního modelování, služby, které lze využít k získaní dat, a bezpečnost připojení. Obecně se jedná o informace užitečné především pro práci s OPC UA klienty.

Čtvrtá kapitola se věnuje problémové doméně a požadavkům kladeným na Systém. Je zde představen zadavatel a zasazení práce do kontextu. Zdůvodňuje, proč je řešení rozděleno do dílčích projektů: databáze, knihovna, aplikace. Kapitola dále vyjmenovává funkční a nefunkční požadavky, kterým se tato práce věnuje. Nakonec je vykreslen konceptuální model, který zobrazuje data a vztahy mezi nimi.

Pátá kapitola popisuje návrh, implementaci a testování databáze. Text analyzuje doménu a požadavky v kontextu uložení dat a je vytvořen relační

datový model s integritními omezeními. Rovněž se zde vybírá vhodná implementace relační databáze.

V šesté kapitole se práce zaměřuje na návrh, implementaci a testování pomocné knihovny Scalable OPC UA. Projekt vznikl jako reakce na současné knihovny pro standard OPC UA. Na základě domény představuje zejména reprezentaci datových typů a hodnot. Navíc poskytuje klienta se základními funkcionalitami pro získávání dat.

Sedmá kapitola obsahuje návrh, implementaci a testování aplikace. Dochází k propojení dílčích projektů a doména s požadavky je zde analyzována v kontextu aplikace. V rámci návrhu je zvolena architektura a zobrazen model datových tříd.

V poslední kapitole dochází na diskuzi k možným vylepšením a v závěru jsou shrnuty výsledky práce.

Kapitola 1

Cíl práce

Cílem práce je vývoj Systému pro sběr dat firmy ModemTec, který bude součástí komplexního diagnostického řešení. Tato práce má poskytnout rozšiřitelný návrh a znovupoužitelné komponenty. Dále má být vytvořena implementace návrhu, která bude otestována. Zařízení, odkud probíhá sběr dat, komunikují pomocí standardu OPC UA. Vzhledem k očekávanému použití v energetice, by měl být Systém kompatibilní s normami IEC 61580. Všechno zmíněné je rozděleno do dílčích cílů:

- Prvním dílčím cílem je analýza komunikačního standardu OPC UA, jelikož k úspěšné tvorbě návrhu musí mít vývojář představu, co od povinného standardu očekávat. Standard využívá objektový návrh informačních modelů, různé možnosti autentizace autorizace a poskytuje množství služeb pro čtení/zápis dat.
- Druhým dílčím cílem je návrh a implementace databáze. Ta poskytuje uchování dat, které systém nashromáždil z OPC UA serverů. Také se definují integritní omezení, jenž jsou kladeny na uložené informace.
- Třetím dílčím cílem je vypracování návrhu a implementace aplikace. Zde se střetává používání služeb poskytovaných OPC UA klienty s uchováním dat v databázi. Aplikace poskytuje rozhraní pro zadávání operací, které se mají provést nad zvolenými servery.
- Nakonec dochází k hodnocení výsledku práce. Také jsou nastíněny další vylepšení a cesty, po kterých vývoj Systému pro sběr dat může kráčet.

Kapitola 2

Použité technologie

Kapitola představuje technologie, které jsou použity při implementaci řešení.

2.1 Java Virtual Machine

Java Virtual Machine je implementace virtuálního stroje, na kterém běží Java programy. Nejprve se kompiluje zdrojový kód do tzv. bytecode. Ten je pak interpretován na různých systémech. Bytecode je intermediary language. Jedná se tedy o mezistupeň mezi zdrojovým kódem a konkrétním hostujícím systémem. Základními komponentami JVM jsou:

- Class Loader – zajišťuje načtení, verifikaci a linkování bytecode.
- Run-Time Data Areas – místa v paměti, které JVM využívá za běhu. Jedná se například o haldu pro alokovanou paměť programu nebo zásobník pro lokální proměnné či PC registry pro uložení současných instrukcí.
- Execution Engine – vykonává instrukce z paměti, používá interpretr, Just-In-Time Compiler a Garbage Collector.

Existuje vícero jazyků, které se využívají při programování pro JVM, avšak tato práce využívá jazyky Java a Scala. [5]

2.2 Slick

Projekt Slick je knihovna napsaná v jazyce Scala. Podporuje práci s relačními databázemi. Tedy modelování schéma, spouštění dotazů, vkládání a update dat. Za cíl si klade umožnit naprogramovat práci s databází ve stejném programovacím stylu, ve kterém se zapisují operace nad kolekcemi ve Scale. Vše je type-safe. Využívá koncept asynchronního programování, klasického dotazování v relačních databázích a akce. Ty lze spouštět nad databází, spojovat do větších akcí či tvořit transakce. [6]

2.3 ScalaTest

ScalaTest je knihovna ve Scale, která umožňuje tvorbu testů a snadněji čitelný zápis kódu testů. Ačkoliv se jedná o samostatný projekt, ScalaTest se stal prakticky standardem pro testování projektů ve Scale. Díky možnosti importu Java kódu do Scaly lze knihovnu využít i pro testování projektů v Jave. Podporuje totiž integraci s JUnit, TestNG, Ant, Maven, sbt, ScalaCheck, JMock, EasyMock, Mockito, ScalaMock, Selenium, Eclipse, NetBeans, a IntelliJ. Díky skládání komponent a vlastnostem jazyku Scala je možné využít ScalaTest jak pro malé, tak pro velké projekty. [7]

2.4 Typesafe Config

Jedná se o projekt naprogramovaný v Jave, který podporuje načítání konfigurace. Implementuje přečtení Java properties, JSON a JSON nadmnožiny zvané HOCON. To vše slouží jako konfigurační soubory. HOCON představuje formát JSON rozšířený o další prvky, např.:

- komentáře,
- záměna „=“ a „:“,
- vynechání uvozovek u názvů atributů,
- jednoduchá dědičnost.

Za knihovnou stojí společnost Lightbend. [8] Vzhledem k tomu, že projekt je v Jave, existují různá rozšíření pro Scalu za účelem využití vlastností jazyka. Například tato práce používá knihovnu PureConfig. [9]

2.5 sbt

Jedná se o build tool pro projekty ve Scale a Jave. Ve Scala komunitě se jedná o volbu číslo jedna. Nástroj poskytuje například:

- Nativní podpora Scaly a mnoha frameworků.
- Podporuje kontinuální kompilaci, testování a nasazení.
- Programování sestavení projektu je psáno ve Scale, v DSL pro sbt.
- Podporuje projekty, kde se nachází Java i Scala kód.
- Závislosti jsou řešeny skrze Apache Ivy, který rovněž podporuje i repozitáře ve formátu Maven.

Co se týče IDE, pro sbt existují pluginy například pro Intelij IDEA a Eclipse. Rovněž nabízí interaktivní konzoly pro Scala REPL. [10]

2.6 Lift-JSON

Knihovna Lift-JSON je součástí projektu Lift Framework. Dodává moduly pro parsování JSON a jeho formátování. To vše stojí na bázi abstraktního syntaktického stromu, do kterého je modelována struktura JSON dokumentu. Všechny funkcionality jsou tudíž implementovány nad AST. Knihovna dále dodává DSL pro přirozenější zápis kódu. Konkrétními funkcionalitami je (de)serializace, parsování JSON, převod AST do JSON či XML, renderování podle zadaných parametrů. [11]

2.7 PostgreSQL

PostgreSQL patří mezi objektově-relační databáze, které rozšiřují SQL. Podporuje pokročilé funkcionality jakými jsou pohledy (views), spouštěče (triggers), procedury (procedures) a funkce (functions). Umožňuje dotazování i v rámci JSON hodnoty a vkládání dat ve formátu XML. Pro psaní procedur/funkcí lze použít i jiné jazyky než SQL, např. Python. Postgres běží na majoritě operačních systému (Windows, Linux, macOS atd.). Od roku 2001 splňuje ACID. [12]

2.8 Eclipse Milo

Knihovna Eclipse Milo implementuje OPC UA v jazyce Java. Podporuje klienta, server a stack. Verze 6.4 obsahuje vyčítání hodnot klientem, zápis, prohledávání informačního modelu, volání metod a tvorba subskripce. K procesu autentizace lze využít anonymní připojení, uživatelské jméno a heslo, certifikáty X509. Dále klient je schopen objevovat koncové body servery a získat jejich popisy. Volání metod klienta je asynchronní. Server zpřístupňuje i modelování vlastních datových typů. [13]

2.9 open62541

Knihovna open62541 implementuje OPC UA v jazyce C. Podporuje klienta, server a stack. Verze 1.2 implementuje binární protokol OPC UA, podporuje Micro Embedded Device Server Profile a pár funkcionalit navíc. Knihovna vlastní synchronní a asynchronní funkce. Navíc umožňuje připojení klienta s využitím uživatelského jména, hesla nebo anonymní. Rovněž lze použít kódování zpráv skrze PKI. Připojený klient může číst, zapisovat data na server, volat metody, vytvořit subskripci a prohledávat informační model. Server podporuje vlastní datové typy. [14]

Kapitola 3

Komunikační standard OPC UA

Kapitola popisuje charakteristiky technologie OPC UA. Především se zaměřuje na datové modelování a získávání dat, tzn. služby a bezpečnostní prvky, které jsou k tomu nutné. Text kapitoly, není-li uvedeno jinak, je parafrázován ze zdroje [1].

3.1 Úvod

Standard OPC UA, kde OPC stojí pro Open Platform Communications a UA pro Unified Architecture, poskytuje otevřený komunikační protokol. Za jeho vznikem a rozvojem je nadace OPC Foundation. Standard vznikl jako reakce na model COM/DCOM od Microsoft Windows a snaží se o zjednodušení komunikace v průmyslovém odvětví. Za základní rysy lze zmínit:

- norma označena jako IEC 62541,
- licence GPL 2.0,
- nezávislost na platformě – od mikrokontrolérů po servery na cloud,
- bezpečnost – šifrovaní, autentizace a jiné,
- komplexní informační modelování.

S posledním zmíněným souvisí datové modely, v řeči OPC UA informační modely.

3.2 Informační model

Model lze chápat třemi způsoby. Buď skrze objektově-orientovaný pohled, nebo nahlížet na model jako strom uzlů, či uchopit data z hlediska referencí. Cílem je obsáhnout v modelu data a současně definice popisu dat.

Z hlediska objektového návrhu OPC UA podporuje: objekty, třídy, dědičnost, polymorfismus, zapouzdření, kompozici, metody. Navíc umožňuje specifikaci vztahů mezi objekty. U tříd si lze pokládat otázky, zda jsou v relaci kompozice nebo agregace, jedná-li se o vlastnost nebo organizaci, nepovinný atribut atp.

Uzly slouží k implementaci objektů a relací. Jsou různých druhů – od objektových přes proměnné po pohledy. Zároveň uzly mají své atributy, které pomáhají specifikovat technické detaily, tedy uživatelský přístup, vedení historických dat, identifikace uzlu, aj.

Reference představují relace mezi uzly. Pro implementaci modelu je prvoplánově rozhodující, zda se jedná o hierarchickou, nebo o nehierarchickou referenci. Jinak se jedná o pojmenování a zařazení vztahu pod definovaný typ (agregace, kompozice, atd.). Nicméně můžou být aplikace, které přesné definování reference vyžadují.

3.2.1 Uzel (Node)

Uzel je základní stavební jednotka informačního modelu. Standard dělí uzly na třídy (*NodeClass*):

- *Object* – objekt/instance třídy, kořen podstromu s uzly dané třídy;
- *Variable* – proměnná/atribut;
- *Method* – definice metody na objektu/instanci;
- *ObjectType* – definice třídy, resp. kořen podstromu s uzly třídy;
- *VariableType* – definice proměnné/atributu;
- *ReferenceType* – definice reference;
- *DataType* – definice datového typu;
- *View* – definice pohledu.

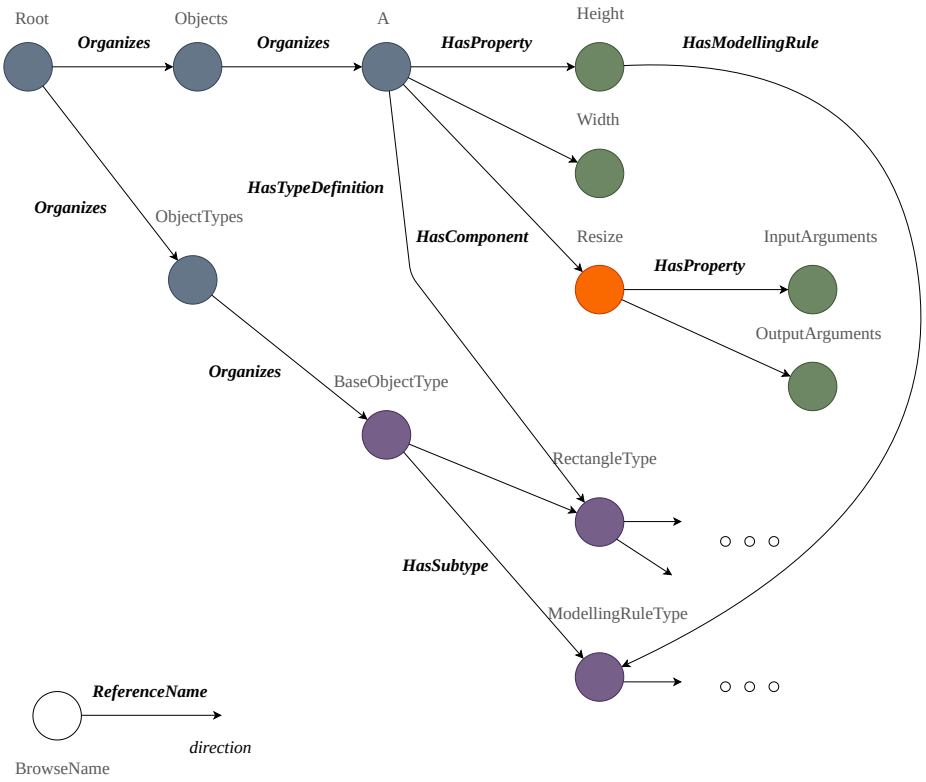
Uzel vlastní atributy, které například popisují jeho název, přístupová práva uživatele či hodnotu (v případě třídy *Variable* a *VariableType*). Konkrétní atributy se liší podle třídy (přehledová tabulka ve zdroji [2]).

3.2.2 Reference (Reference)

Reference představuje vztah mezi uzly. Hierarchické reference organizují uzly do struktury, stromu. Oproti tomu nehierarchické reference představují odkazy definice či vlastnosti. Dále se u referencí určuje směr mezi uzly. OPC UA podporuje jednosměrné a obousměrné. Podle směru se odlišuje jméno. Například

dopředná (*Forward*) reference *HasComponent* má zpětnou (*Inverse*) *ComponentOf*. Díky možnosti definovat vlastní typy referencí lze vytvářet modely s přesnými názvy a vlastnostmi vztahů mezi uzly.

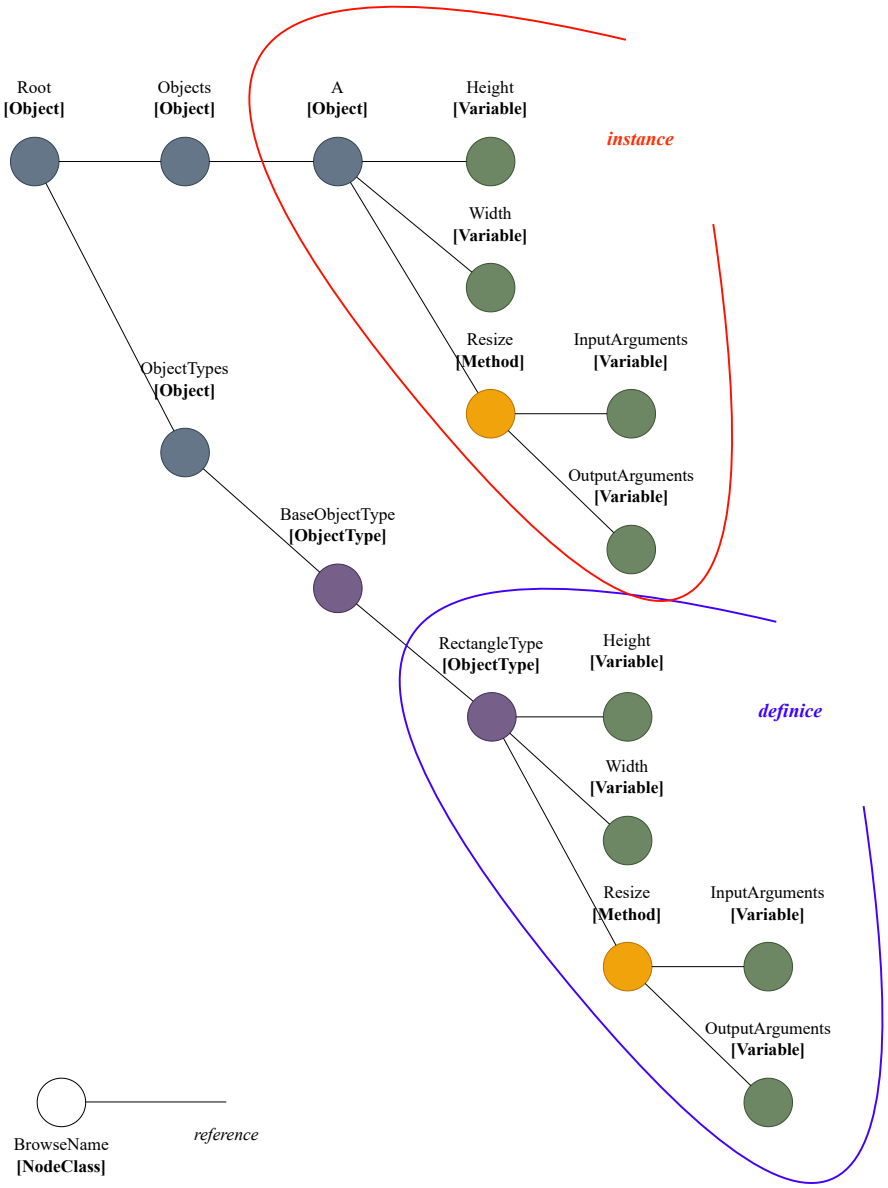
Obrázek 3.1 představuje výřez z informačního modelu s vyznačením všech referencí a jejich směry. Jedná se o orientovaný acyklický graf.



Obrázek 3.1: Uzly s vyznačením části referencí

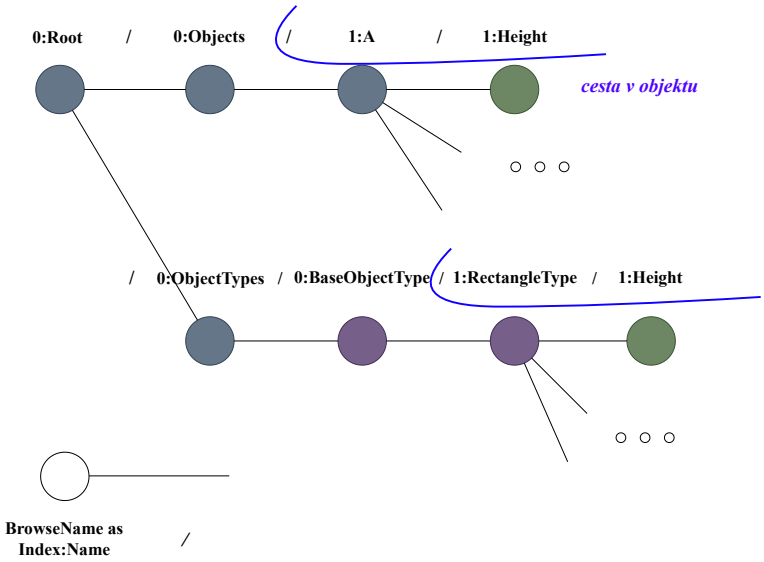
3.2.3 Třída (Object, ObjectType)

Informační model obsahuje jak definici třídy (*ObjectType*), tak instanci (*Object*). Lze vytvořit objekt (*Object*) i bez definice. Jak je vidět na obrázku 3.2, při vytváření instance se kopírují uzly. Modelovací nástroje nebo servery obvykle neobsahují optimalizaci, kdy by došlo například k referencování metody. Současně v obrázku 3.2 se nachází podstrom definic objektových typů (s kořenem *ObjectTypes*) a podstrom objektů (s kořenem *Objects*). Mezi uzly jsou vyznačeny pouze hierarchické reference, proto lze mluvit o podstromech.



Obrázek 3.2: Definice a instance třídy

V rámci práce se třídami lze použít atribut uzlu *BrowseName* s datovým typem *QualifiedName*. Jedná se o strukturovaný datový typ obsahující název a index jmenného prostoru. V případě *BrowseName* tedy název uzlu a index jmenného prostoru, odkud definice třídy pochází. Atribut *BrowseName* je součástí cesty k uzlu, která může být relativní nebo absolutní od kořene stromu informačního modelu. Cesty k uzlům v podstromu definice a instance jsou shodné, jak zobrazuje obrázek 3.3.



Obrázek 3.3: Cesta k uzlu v rámci třídy

3.2.4 Datový typ (DataType)

OPC UA podporuje jednoduché datové typy, výčty a struktury. Kromě základních typů si uživatel může definovat vlastní struktury, jednoduché a enumerační typy. Standard umožňuje kódování hodnot v binárním kódu, v XML či JSON. Lze také použít i vlastní kódování. Informační model obsahuje definice datových typů a jejich kódování.

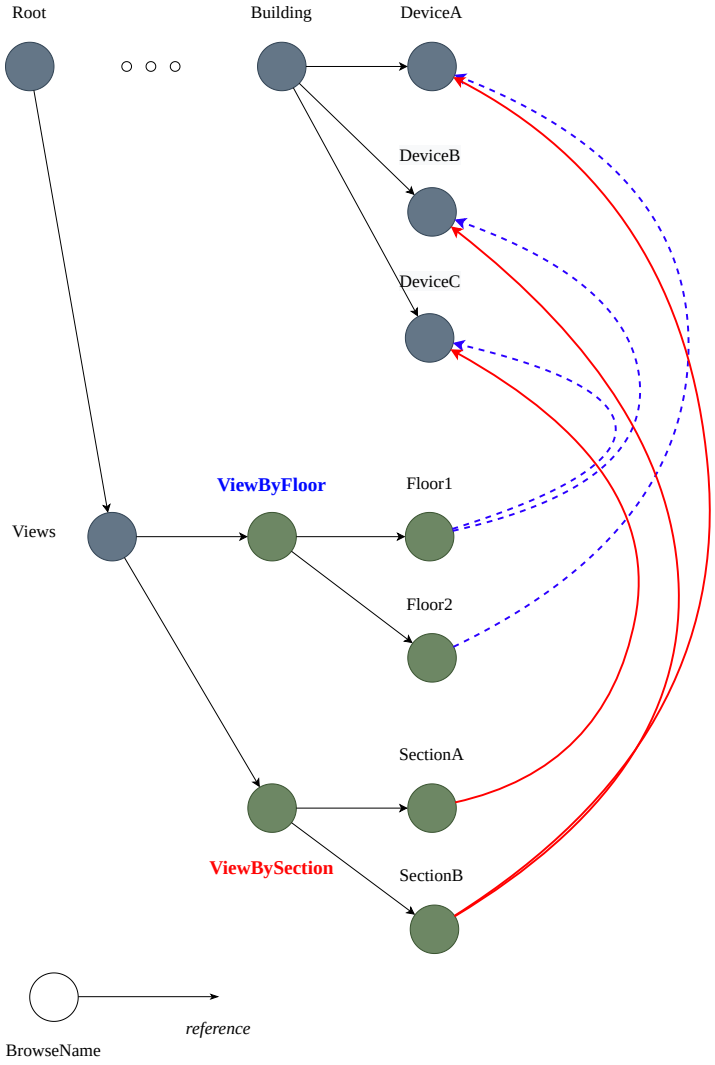
3.2.5 Jmenný prostor (Namespace)

Všechny uzly spadají pod nějaký jmenný prostor. Ten má svoje URL a index v rámci pole jmenných prostorů nacházejících se na serveru. Smyslem jmenných prostorů je zajištění unikátnosti identifikátorů v rámci celého informační modelu, kterými může být: *UInt32* (32-bitové bezznaménkové celé číslo), *String* (řetězec), *GUID* (globální unikátní identifikátor), *Base64String* (řetězec v Base64 kódování). Základním prostorem je OPC UA Foundation Namespace – často označovaný jako *Zero Namespace*, protože na každém serveru má index nula.

3.2.6 Pohled (View)

Autoři standardu přirovnávají pohledy k jazyku SQL. Mají stejnou funkci: vyhledávání nebo organizaci uzlů podle daného kritéria. Pohledy tvoří hierarchické struktury nad daty. Například mějme měřiče teploty v budově. Uživatel

si může uspořádat zařízení podle pater, ale taky i podle sekcí v rámci budovy. Vizualizaci referencí mezi uzly je vidět na obrázku 3.4



Obrázek 3.4: Příklad pohledů na data

3.3 Služby (Services)

Pro práci s daty jsou využívány především následující serverové služby. V závislosti na dané implementaci serveru/klienta mohou se chovat asynchronně či synchronně. V této části jsou služby popsány z hlediska myšlenky/principu. Tabulky s parametry žádostí a odpovědí serveru jsou k dohledání v referenci OPC UA (ve zdroji [3]).

3.3.1 Vyhledávání (BrowseService)

Služba je užívána za účelem prohledávání informačního modelu. Klient vytyčí startovní uzly a filtry, kterými specifikuje cíle vyhledávání. Následně server vrací seznam uzlů, které jsou dosažitelné z počátečních uzlů přes reference a které vyhovují zvoleným parametrům vyhledávání.

Tato služba je často používána v souvislosti budování stromové struktury nebo k získání metadat, např. atributy uzlů. OPC UA podporuje optimalizaci z hlediska paměti. Podpůrná služba *BrowseNextService* nabízí získání dalších uzlů, pokud server nebyl schopen odeslat všechny výsledky v jediné odpovědi.

3.3.2 Čtení (ReadService)

Na rozdíl od *BrowseService*, která k uzlům vrací maximálně atributy, jenž vlastní každý uzel, je služba *ReadService* orientována na získání jakýkoliv atributů zvolených uzlů. Stejně tak ji lze využít na vyčítání hodnot proměnných či jejich částí (pokud je například hodnotou pole).

Až na atribut *Value* lze typicky vyčíst hodnoty všech atributů. Hodnota atributu *Value* je ale omezena přístupovými právy. Ty jsou specifikovány v atributech *AccessLevel* a *UserAccessLevel*. Rozdíl mezi jmenovanými tkví v tom, že *UserAccessLevel* blíže specifikuje přístup přihlášenému uživateli. Navíc *UserAccessLevel* může omezit práva danému uživateli definovaná v *AccessLevel*, nesmí je ale rozšířit.

3.3.3 Zápis (WriteService)

Služba zprostředkovává úpravy vlastností zvolených uzlů. Jaké hodnoty atributů může klient měnit je definováno v atributech *WriteMask* a *UserWriteMask*. Možnost zápisu hodnoty proměnné je specifikována skrze *AccessLevel* a *UserAccessLevel*. Typicky pouze atribut *Value* je zapisovatelný. Ostatní atributy lze měnit, pouze pokud server dovoluje změny v informačním modelu.

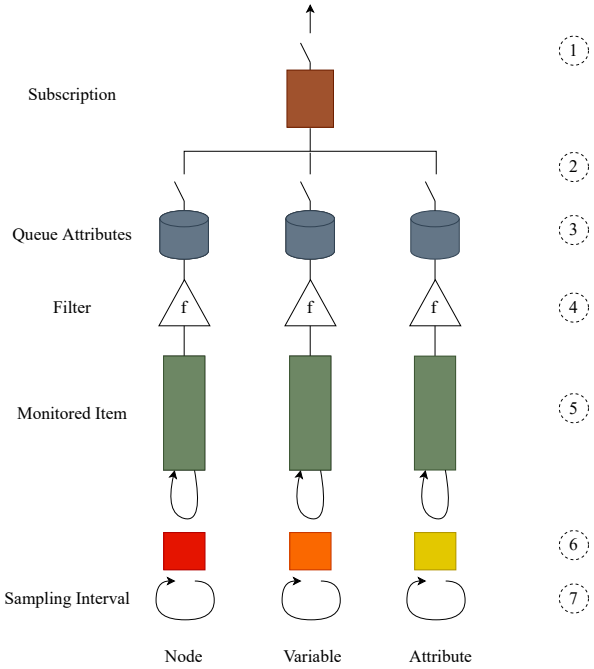
3.3.4 Odběr (Subscription)

Hlavním účelem této služby je zasílání notifikací klientovi. Zpravidla se jedná o upozornění na aktualizaci hodnoty v proměnné, nebo změnu v atributu, či ohlašování signálů a událostí na objektech. Klient na příchozí zprávu reaguje a má možnost získat nová data ze serveru.

Jedna subskripce se skládá z jedné či více monitorovaných položek (Monitored Item). Těmi jsou výše zmiňované atributy uzlů, signály, události a hodnoty proměnných. Lze tedy říci, že Subscription zabaluje/organizuje notifikace o jednotlivých změnách do *NotificationMessages*, které jsou společně odeslány serverem.

Při tvorbě Subscription klient nastaví *PublishingInterval*, který definuje časový úsek, po kterém dojde k zaslání notifikace ze serveru. Posílány jsou

pouze zprávy o událostech, jenž doposud nebyly zaslány. V případě, kdy nedošlo ke změnám v potřebném intervalu či je Subscription vytvořeno, server posílá keep-alive zprávu. Ta informuje, že subskripce je stále aktivní. Princip monitorování položek na straně serveru ukazuje obrázek 3.5.



Obrázek 3.5: Schéma průběhu monitorování položek na straně serveru

Popis bodů z obrázku:

1. Reportování je zapnuto/vypnuto pro Subscription.
2. Reportování Monitored Item.
3. Queue Attributes definují způsob uspořádání ve frontě.
4. Filter aplikuje parametry výběru na data.
5. Monitoring Mode určuje, zda je monitorování a reportování zapnuto/vypnuto.
6. Monitored Item může pozorovat atribut uzlu, hodnotu nebo uzel poskytující události.
7. Sampling Interval definuje časový úsek, po kterém dojde k shromažďování dat serverem.

3.3.5 Čtení historických dat (HistoryReadService)

Klient použije službu v případě, kdy chce získat předešlé hodnoty nebo události daného uzlu. Servery mají implementovaný přístup k historii skrze *HistoryReadService*, ev. *HistoryWriteService*, jelikož předchozí hodnoty nejsou viditelné v informačním modelu. Ten reprezentuje pouze aktuální stav. Pokud model obsahuje historická data, jsou explicitně modelovány objekty, jenž se postarají o uchování.

Při vytváření žádosti klient specifikuje, časový úsek a případně maximální počet hodnot na uzel. Pro případ, že přenášený objem dat je velký, nebo vyprší timeout pro klienta, existuje parametr *ContinuationPoint*. Bod v historii umožňuje vytvořit žádost o další data, které vyhovují zadání a navíc následují po *ContinuationPoint*.

3.3.6 Volání metody (MethodCallService)

Služba poskytuje volání metod. Zajišťuje jak zavolání s parametry, tak získání návratových hodnot. Lze specifikovat, jaké metody se mají zavolat, nelze ale definovat pořadí. V případě, že klient vyžaduje určité schéma volání, musí použít tuto službu vícekrát.

Aby nedocházelo k zavolání metod, které trvají příliš dlouho, server stanovuje timeout. Na jednu stranu to chrání klienta před čekáním, na druhou stranu to dává závazek implementaci serveru v podobě návrhu metod. V případě, kdy některé operace trvají na straně serveru příliš dlouho, projeví se řešení i na návrhu informačního modelu. Například časově náročná metoda bude nahrazena konečným automatem a klient/server budou reagovat dle stavů. Podobně se řeší práce s programy na straně serveru.

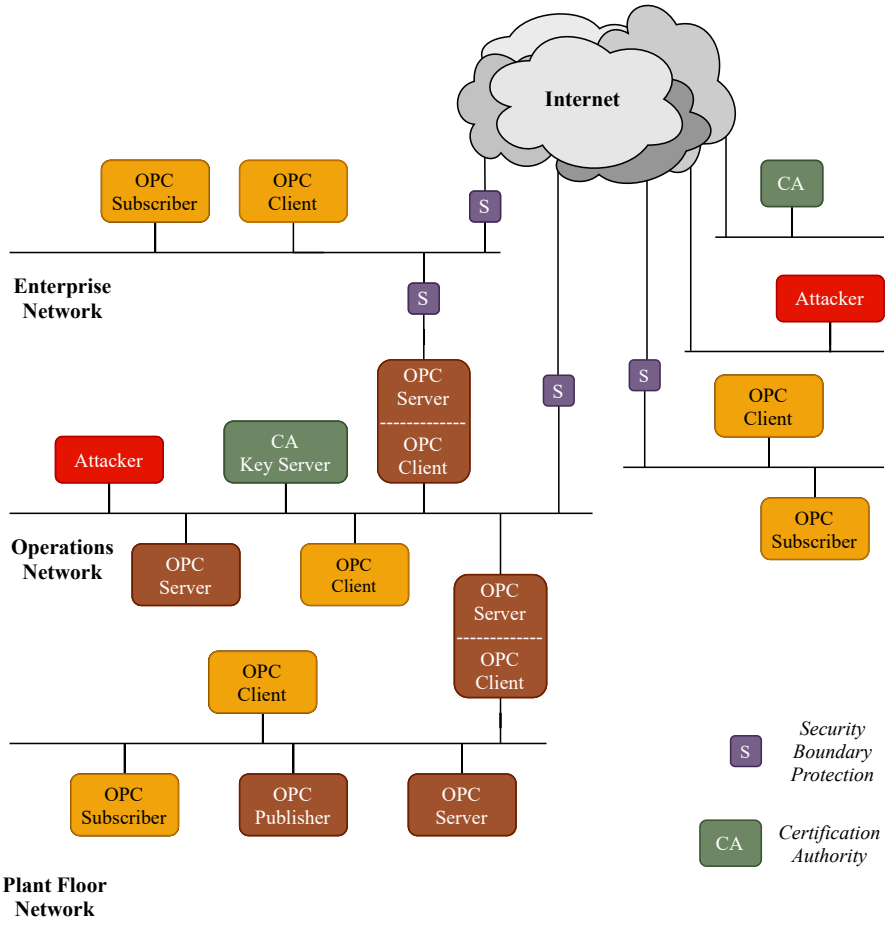
3.4 Bezpečnost

Na bezpečnost OPC UA se lze dívat z vícero úhlů. Následující text popisuje především prvky, se kterými uživatel přichází do kontaktu při práci s OPC UA klientem.

Jak bylo zmíněno v úvodu, protokol umožňuje budování sítě napříč zařízeními různých funkcí – měřiče, klimatizace, roboti v automatizované výrobě, cloudový server atd. Obrázek 3.6 představuje příklad části sítě. Nachází se v ní kromě OPC UA serverů a klientů, také i certifikační autority a útočníci. OPC UA systémy jsou ohrožovány stejnými hrozbami jako jiné síťové struktury. Útoky jsou kupříkladu:

- *Denial of Service* – útočník znefunkční služby pomocí využití chyby či zahlcení serveru požadavky;
- *Rogue Server* – útočník vloží serveru do sítě předstírající identitu již existujícího server, či jako nový;

- *Session Hijacking* – útočník získá informace o probíhajícím spojení a pomocí vlastních zpráv, jenž jsou z hlediska spojení validní, získá kontrolu nad komunikací;
- *Malformed Messages* – útočník modifikuje zprávu do chybného formátu/hodnoty za cílem vyvolat nežádoucí (neautorizované) efekty;
- *Message Replay* – útočník zachytí zprávy a pošle je znovu s cílem vyvolat nežádoucí (neautorizované) efekty.



Obrázek 3.6: Příklad OPC UA sítě

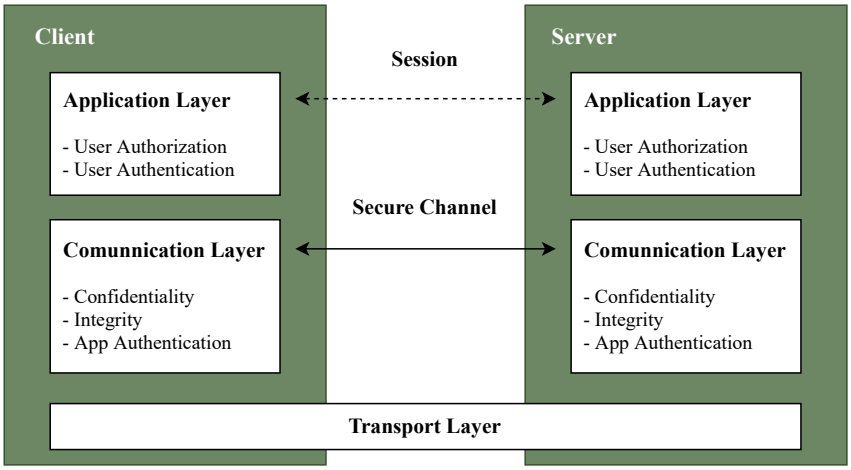
3.4.1 Architektura klient-server

OPC UA podporuje Publish-Subcribe (zkratkou *PubSub*) a Client-Server architekturu. Poslední jmenovaná je implementována v aplikační a komunikační vrstvě. OPC UA obsahuje komunikaci jak s vytvořením relace (*Session*), tak bez ní.

Aplikační vrstva si klade za cíl zajistit autorizaci a autentizaci. Relace probíhá nad zabezpečeným kanálem *(Secure Channel*). Pokud dojde k přerušení zabezpečeného kanálu a nepodaří se obnovit, relace přestane být validní.

Komunikační vrstva zajišťuje mechanismy pro zachování integrity, důvěryhodnosti a autentizaci aplikace. K tomu využívá *Secure Channel*, který poskytuje šifrování pro udržení důvěryhodnosti, podpisy zpráv pro integritu a certifikáty pro autentizaci aplikace.

Transportní vrstva může také implementovat důvěryhodnost a integritu. Využívá se protokol HTTPS. Za zmínku stojí, že HTTPS certifikáty jsou často sdíleny vícero aplikacemi. Existuje riziko, že budou kompromitovány mimo OPC UA systém.



Obrázek 3.7: Architektura klient-server v OPC UA

V případě, kdy probíhá komunikace bez vytvoření spojení (*session-less*), musí proběhnout autentizace uživatele. Lze přidat bezpečnostní prvek *Access Token*. Tedy token poskytnutý autorizační službou (*Authorization Service*), který reprezentuje udělená oprávnění.

3.4.2 Autentizace aplikace

OPC UA využívá tento koncept za účelem vzájemného předložení identifikace aplikacemi. Každá instance (*OPC UA Application Instance*) má svůj certifikát (*Application Instance Certificate*). Příjemce certifikátu ho porovná se seznamem, kde jsou uvedeny ty důvěryhodné (*Trust List*). Koncept *Trust*

Lists je implementován jako *Certificate Store*, kde administrátor uloží pouze důvěryhodné a validní certifikáty a certifikační autority. OPC UA podporuje certifikáty schválené CA, ale i *self-signed*.

3.4.3 Autentizace a autorizace uživatele

Servery podporují možnost anonymního přístupu. V praxi jsou ale z důvodu ochrany užívány uživatelská jména a hesla a PKI. OPC UA podporuje kromě definice práv pro konkrétního uživatele, také i koncept rolí (*Roles*). Uživatel se nachází v nějaké roli. Role jsou implementovány pomocí informačního modelu – vytvoření uzlů a referencí podle schéma daného standardem (tj. Annex F (normative) User Authorization z reference OPC UA [4]).

3.5 Shrnutí

Na OPC UA se lze dívat ze dvou úhlů. Prvním z nich je komunikační protokol a rozhraní. Druhým organizace dat.

Pro server existují doporučení uložení dat v paměti. Kromě standardní binární reprezentace je možné hodnoty provázat s databází, XML soubory, formátem JSON, atd. Například při použití na malých zařízeních se vyskytují situace, kdy se k serveru přistupuje zvenčí jakožto prezentace hodnot a rozhraní funkcionalit zařízení a zevnitř, kdy ostatní procesy ve firmwaru využívají OPC UA server jakožto rozhraní pro uložení hodnot.

Informační model umožňuje implementaci OOP. Podporuje modelování definic a instancí. Rozhraní může fungovat stavově a bezestavově. Skrze standard lze řídit i běh jiných programů, konstrukci konečných automatů či přístup do souborového systému.

Za účelem zajištění bezpečnosti standard implementuje přihlašování s využitím uživatelských jmen, hesel, rolí a Public-Key Infrastructure.

Na jednu stranu OPC UA umožňuje vytvoření jakéhokoliv systému, na druhou stranu tento fakt zvyšuje náročnost a komplexitu řešení, implementace.

Kapitola 4

Analýza domény a požadavků

Kapitola se věnuje seznámení se zadavatelem a zasazení práce do kontextu celkového diagnostického řešení. Dále obsahuje výsledky sběru požadavků a popis problémové domény. Na závěr organizuje strukturu řešení Systému pro sběr dat.

4.1 Představení zadavatele

Společnost ModemTec sídlící v Třinci působí na českém trhu více než 20 let. Svým zákazníkům poskytuje kompletní řešení pro diagnostiku VN i NN zařízení a komunikaci po elektrickém vedení všech napěťových hladin. Nabízí rovněž řešení pro diagnostiku částečných výbojů. Výzkum a vývoj probíhá v Centru pro inteligentní energetiku, do jehož vzniku společnost investovala. ModemTec dále spolupracuje s akademickými institucemi, např. ZČU Plzeň, ČVUT Praha, VUT Brno, VŠB Ostrava. [15]

4.1.1 Celkové diagnostické řešení

Práce je součástí řešení, které poskytuje sběr dat ze zařízení a jejich analýzu, zpracování a prezentaci uživateli. Pod daty si lze představit naměřené veličiny, které jsou nutné k diagnostice a predikci chování elektrických sítí. Práce je zaměřena na projekt zajišťující samotný sběr dat a jejich uložení do databáze.

Vzhledem k tomu, že data/zařízení jsou určeny pro elektrotechnickou oblast, řešení podléhá dodržení standardizace – v tomto případě technologie OPC UA a rodinu norem IEC 61850. Rozhraní měřícího zařízení bude zmíněné standardy implementovat.

V průběhu času se u informačních modelů na zařízeních očekávají změny, s čímž návrh musí počítat. Ideální návrh umí pracovat s obecnými informačním modelem. Zároveň poskytuje/ukládá data tak, aby se snížila, nebo eliminovala závislost dalších projektů, které budou s daty dále pracovat, na používání knihoven a software třetích stran podporující/implementující OPC UA.

4.2 Požadavky

V následujících textech jsou zmíněny a realizovány pouze požadavky, kterým se bude tato práce věnovat. Ačkoliv Systém bude do budoucna rozšiřován o další funkcionality.

4.2.1 Funkční požadavky

Funkční požadavky reprezentují produktové funkce či operace, které uživatel může v daném software provádět. Obecně řečeno popisují chování systému za určitých podmínek. [16] Funkčními požadavky pro tento systém jsou:

1. *UpdateTask* – vyčte aktuální hodnotu daného uzlu ze zařízení a uloží se zdrojovým časem do databáze.
2. *ReadInfoTask* – vrátí informace o zvoleném úkolu.

Jelikož se jedná o na popis krátké procesy, nejsou modelovány pomocí diagramů, např. UML diagram aktivit. Stejně tak nejsou zvlášť analyzovány případy užití, jenž by byly analogií k výše uvedeným funkčním požadavkům.

4.2.2 Nefunkční požadavky

Nefunkční požadavky představují obecné vlastnosti systému. Mnohdy se o nich hovoří jako o atributech kvality. [16] Nefunkčními požadavky kladené na analyzovaný systém jsou:

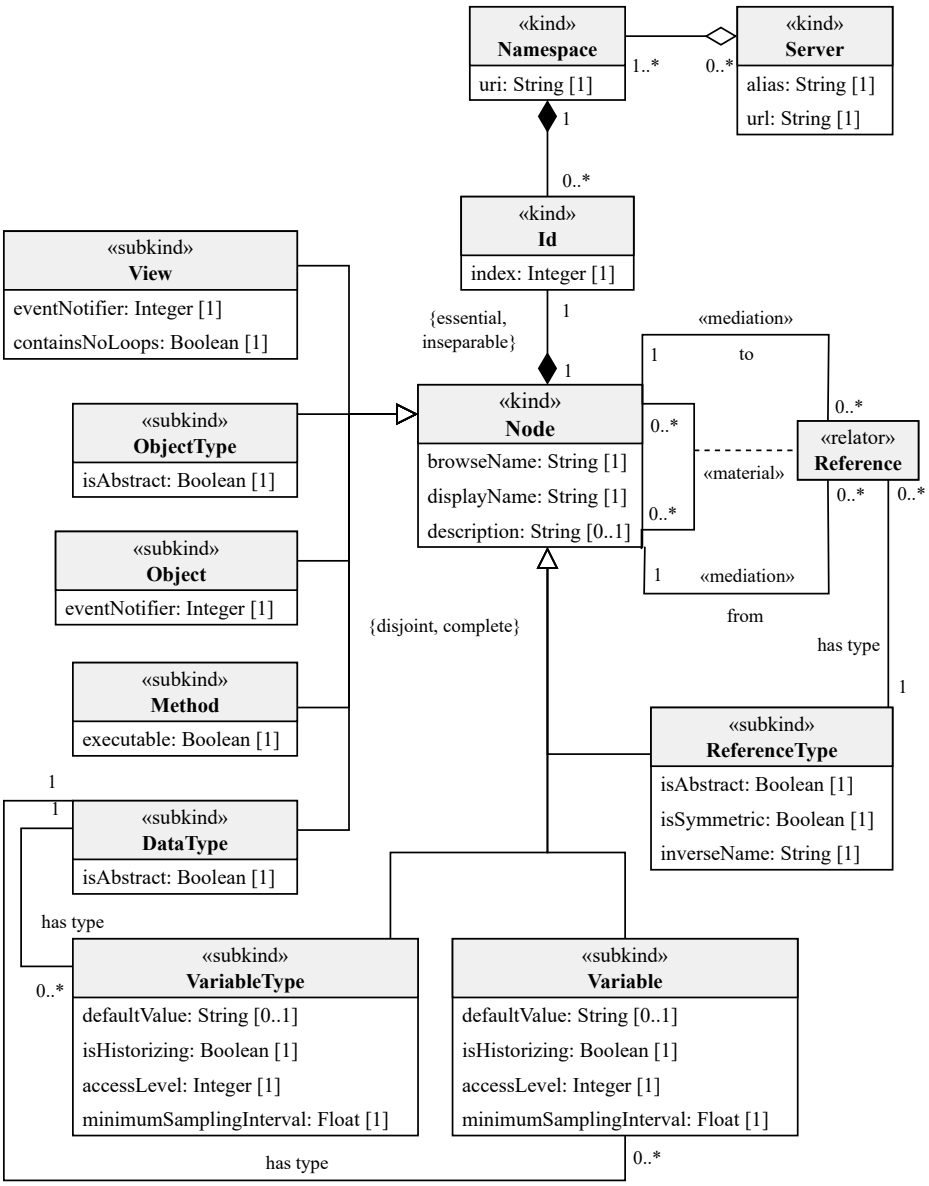
1. *rozšiřitelnost* – možnost přidání dalších příkazů.
2. *flexibilita* – možnost změny uživatelského rozhraní, databáze či aplikační logiky.
3. *zjednodušení OPC UA* – snížit závislost na OPC UA technologii při dalším zpracování dat.
4. *podpora JVM* – použití technologií s podporou JVM.
5. *podpora OPC UA* – technologie použitá pro komunikaci se zařízeními.
6. *podpora IEC 61580* – podpora datových typů z IEC 61850-7-3 a IEC 61850-7-4.
7. *zadávání příkazů skrze CLI* – ačkoliv v zadaní je použito zadávání skrze databázi, nakonec z praktických důvodů bylo vybráno rozhraní přes příkazovou řádku.

4.3 Doména

K modelování problémové domény je použit konceptuální diagram v notaci OntoUML. Je rozdělen do následujících částí.

4.3.1 Informační model

Informační model představuje rozhraní funkcionalit a dat serveru. [1]

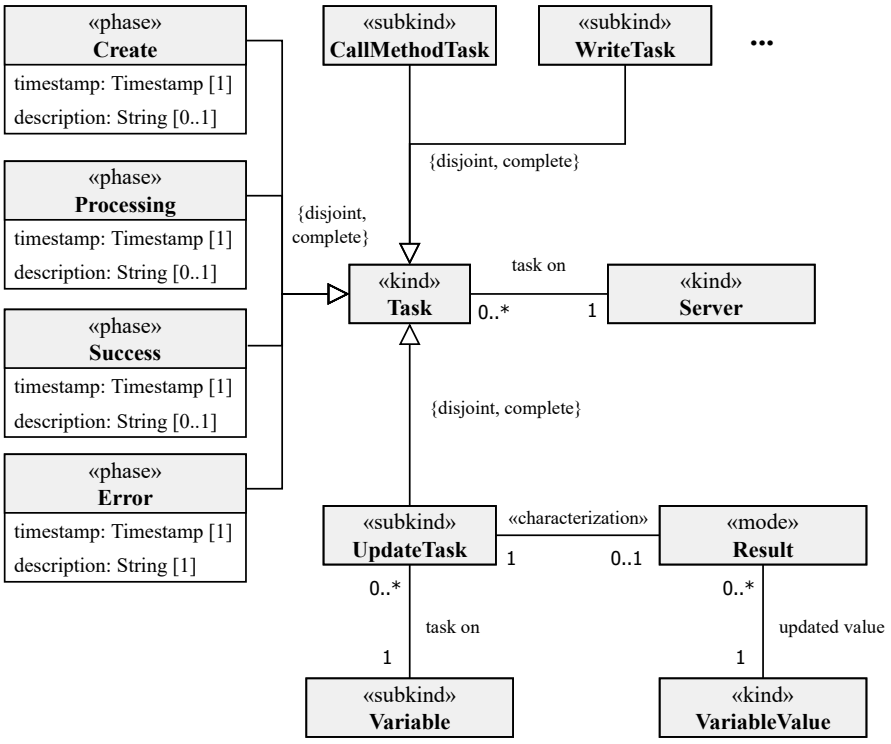


Obrázek 4.1: Informační model OPC UA (OntoUML)

Jak je vidět na obrázku 4.1 informační model je graf. [1] Typ *Node* reprezentuje uzel. *Node* vždy spadá do nějakého *Namespace*, který je možný přirozeně identifikovat pomocí URI. V rámci *Namespace* má *Node* identifikátor. Pouze ve výjimečných případech se nejedná o numerický identifikátor, proto třída *Id* obsahuje index uzlu v rámci jmenného prostoru a odkaz na daný *Namespace*. Hrany grafu tvoří reference, která je představována relátorem *Reference*. Ten je určitého typu a současně udává směr hrany. Ostatní třídy v diagramu 4.1 zobrazují *NodeClass*. Třída *Node* jako taková nemá instance, daný uzel musí být vždy instancí jedné z jejích podtříd. Diagram tudíž modeluje data, která jsou pro daný uzel, jmenný prostor a celkově informační model stálá. Tzn. nejsou uvedeny atributy, které závisí na konkrétním serveru.

4.3.2 Úkol (Task)

Obrázek 4.2 představuje model úkolu. *Task* reprezentuje úkol zadaný aplikaci, která může i nemusí být vztažen na konkrétní uzel. Například vytvoření subskripce není spjato s uzlem, ale monitorované položky již ano. Fáze *Error*, na rozdíl od ostatních fází úkolu, povinně obsahuje popis.

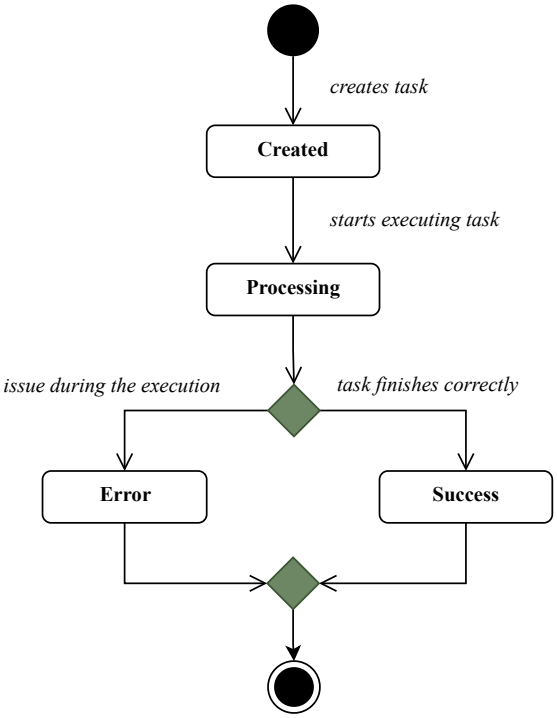


Obrázek 4.2: Úkol (OntoUML)

Co se týče samotného průběhu zpracování, úkol je nejprve přijat, poté zpracován a nakonec je ukončen s úspěchem či chybou. Proces zpracování lze vyjádřit stavy, ve kterých se úkol vždy nachází. Stavy jsou:

- *Created* – příkaz byl vytvořen a uložen do databáze;
- *Processing* – aplikace převzala příkaz a začala vykonávat;
- *Error* – během vykonávání/zpracování došlo k chybě a byl zastaven;
- *Success* – vše proběhlo v pořádku.

Přechody mezi stavy jsou modelovány pomocí UML diagramu stavů, který se nachází na obrázku 4.3.

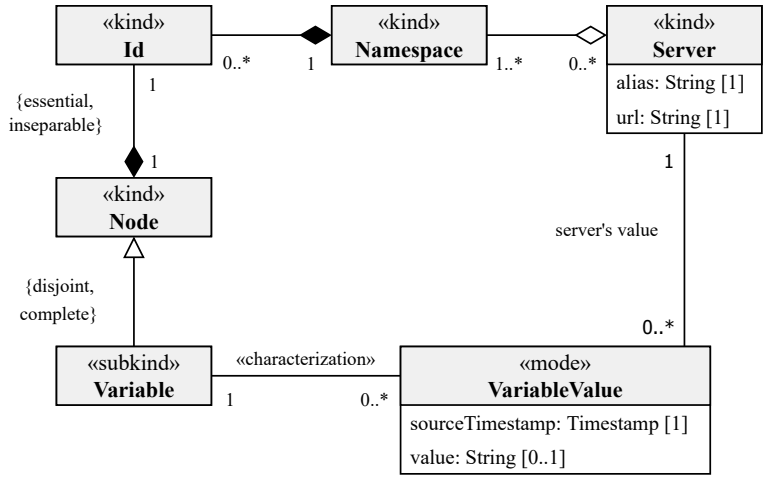


Obrázek 4.3: Stavy úkolu (UML)

Je-li úkol úspěšně ukončen, obsahuje výsledek (*Result*). Výsledkem je konkrétní hodnota proměnné v daném čase. Minimální využití tohoto faktu se očekává u podtypu úkolu *UpdateTask*. Úkol nelze instancializovat, vždy je nějakého svého podtypu. *UpdateTask* představuje úkol, kdy dojde k vyčtení hodnoty ze serveru a uložení, potažmo poskytnuti, hodnoty. U úkolu je sledován nejen údaj o uzlu z informačního modelu, ale i informace o serveru, kde se informačního model nachází.

4.3.3 Hodnota proměnné (VariableValue)

Proměnné v informačním modelu můžou mít evidovanou historii hodnot. [1] Vzhledem k tomu, že daný jmenný prostor, kam proměnná patří, může být přiřazen k vícero serverům, je konkrétní hodnota (*VariableValue*) přiřazena k serveru (*Server*). Rovněž hodnota vlastní čas, kdy vznikla.



Obrázek 4.4: Hodnota proměnné (OntoUML)

4.4 Struktura řešení Systému pro sběr dat

K rozdělení řešení na dva dílčí projekty databáze a aplikace, které plyne ze zadání, je přidána ještě knihovna. Tudíž celé řešení se skládá z:

- Databázový projekt se soustředí na uložení dat. Kromě perzistence dat aplikace zaštiťuje i uložení dat informačních modelů. Což lze využít ve více projektech, např. samostatný klient pro OPC UA server či aplikaci zobrazující informační model a jeho historické hodnoty.
- Knihovna pro OPC UA je projekt, který dodává další funkcionality k práci s OPC UA. Jedná se o kód odtržený od aplikace, protože ho lze využít i na jiných projektech, kde se s technologií OPC UA pracuje.
- Projekt aplikace představuje program, který vyloženě zpracovává požadavky, vytváří úkoly související se sběrem dat.

Kapitola 5

Databáze

Kapitola se věnuje dílčímu projektu zaměřenému na databázi. Obsahuje návrh relačního datového modelu, diskuzi nad volbou databázového stroje a způsob otestování zvolené implementace.

5.1 Návrh

V návrhové části databáze je řešeno, jaký typ databázového systému zvolit. Od toho se odvíjí, jakým způsobem a v jaké notaci bude databázové schéma. Nakonec je nakreslen datový model a z něho vyplývající integritní omezení. Návrh je proveden bez řešení uživatelských rolí a oprávnění, jelikož v době tvorby návrhu není známé, jaké aplikace/klienti budou mít přístup k datům. Databázové schéma je tudíž přiřazeno k nějaké výchozí roli, případně k testovací roli.

5.1.1 Relační vs. grafové databáze

Vzhledem k tomu, že informační model OPC UA je graf [1], nabízí se využití grafových databází. Grafové databáze evidují zejména vztahy mezi daty, jejich vlastnostmi a množstvím. Díky své struktuře dovolují snazší zápis a efektivnější vyhodnocení grafových algoritmů. Mezi případy užití patří analýza interakce uživatelů nebo sledování oběhu peněz. Zjednodušeně řečeno: hrany mezi uzly jsou primárně evidovanými daty. [17] Relační databáze se oproti tomu soustředí na relace, které jsou implementovány jako tabulky. Krátce popsáno: vztahy mezi daty jsou fixní, primárně jsou evidovány záznamy v tabulkách. [17]

Zatímco grafové databáze jsou silně orientované na řešení grafových problémů, relační lze použít na cokoliv. Zvolenou technologií se stává relační databáze. Hlavním důvodem je, že v případě evidování informačních modelů OPC UA nedochází ke změnám v grafové struktuře. K uzlům se přistupuje přes jejich identifikátory. Typicky je vyčten podstrom uzlů, a následně uživatel/apli-

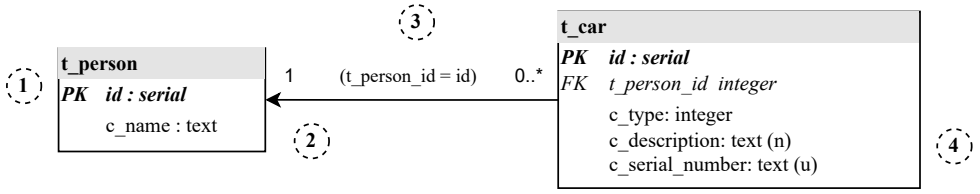
kace vybírá uzel. Nebo uživatel/aplikace má znalost o konkrétních identifikátorech. Dále je potřeba u uzlů třídy Variable umožnit evidování hodnot a navázat informační model na další data, která již grafový profil nemají. Nakonec, ve výjimečných případech dojde k nasazení jiných algoritmů, než jsou prohledávání do hloubky a do šířky.

5.1.2 Relační datový model

Návrh je vyhotoven v jazyce UML. Vzhledem k různým způsobům, jak zakreslit relační datový model v UML, jsou nejprve představeny aspekty notace použité v tomto návrhu. Názvy se řídí pravidly:

- prefix „s_“ pro schéma;
- prefix „t_“ pro tabulku;
- prefix „c_“ pro sloupeček;
- „id“ pro primární klíč;
- „tabulka_id“ pro cizí klíč.

Důvodem pro výjimku může být zvýšení čitelnosti/přehlednosti, nebo vznik duplicity v pojmenování. Prefixy slouží k zamezení potřeby kontroly, zda se jedná o klíčové slovo daného dialektu SQL. Což usnadní testování, kdy testovací databázový stroj není shodný s produkční databází. Navíc se modelují data, jenž obsahují obecné názvy (object, id, index, alias, atd.). Obrázek 5.1 zobrazuje způsob použití UML.

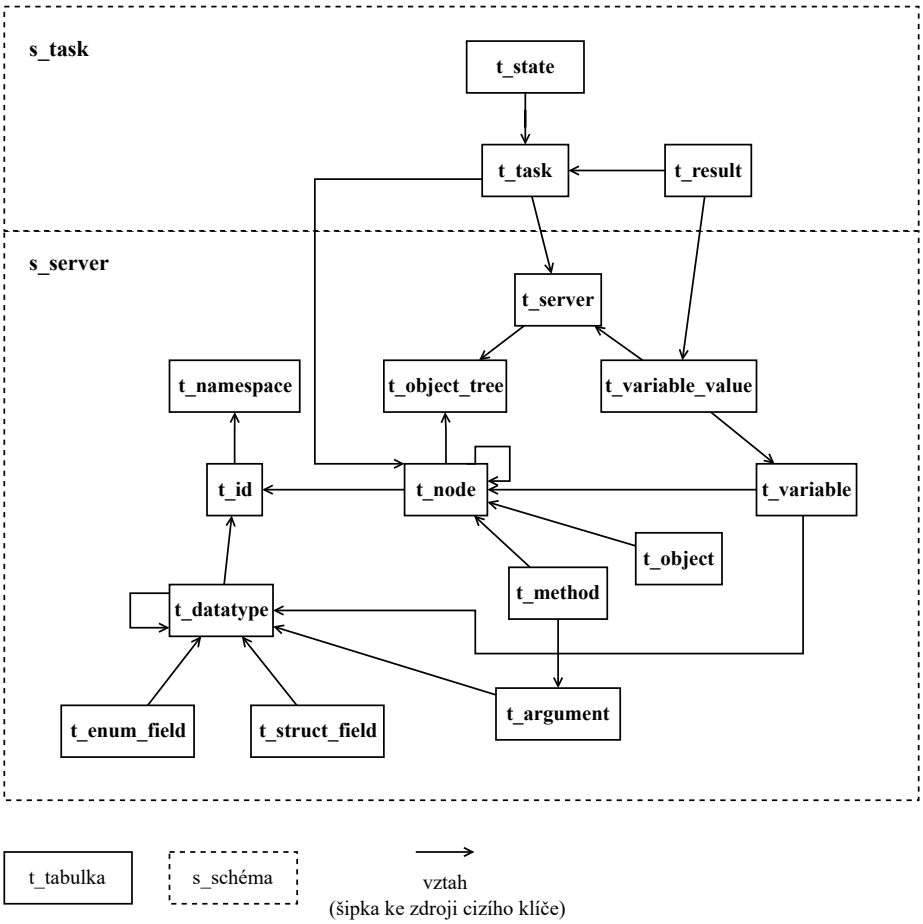


Obrázek 5.1: Legenda k relačnímu datovému modelu

1. označení *PK* (plus tučná kurzíva) – primární klíč, *FK* (plus kurzíva);
2. relace s šipkou ve směru od tabulky *t_car* k tabulce *t_person* – *t_car* vlastní cizí klíč *t_person*;
3. popis relace, resp. použité mapování klíčů: *(cizí klíč = název v tabulce, odkud pochází)*;
4. *(u)* – unikátní hodnota (značí se v případech, kdy se nejedná o primární klíč), *(n)* – dovolena prázdná hodnota (*null*).

Celkový pohled

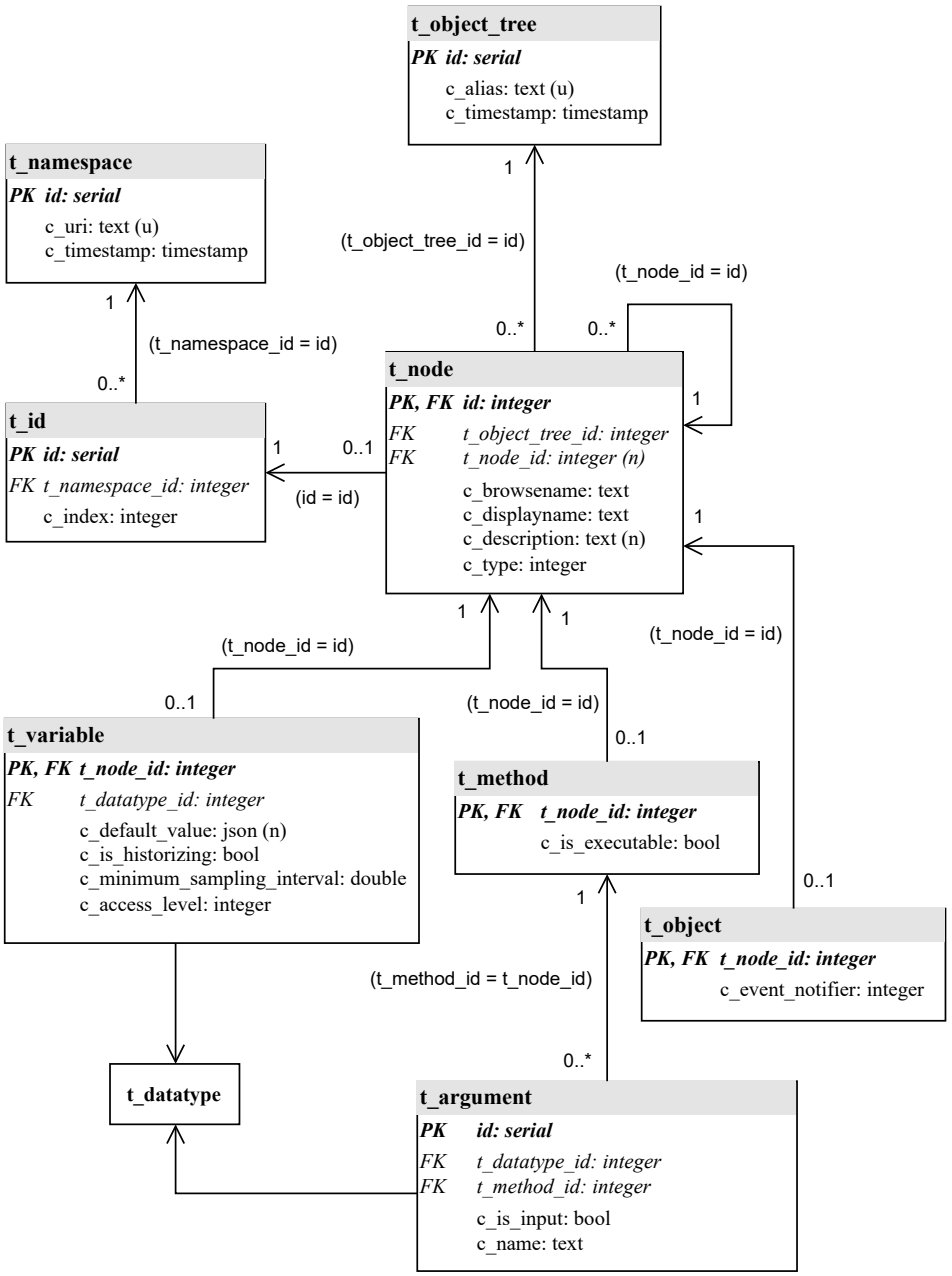
Řešení je rozděleno do dvou schémat (viz obrázek 5.2). Schéma *s_task* souvisí s prováděnými úkoly (*Tasks*). Zatímco schéma *s_server* se zaměřuje na uložení informačních modelů a hodnot na serverech. V případě použití návrhu na jiném projektu, může být i jedno ze schémat odstraněno, např. pro samostatného klienta nemusí být nutné ukládat data v schéma *s_task*.



Obrázek 5.2: Celkový pohled na databázové schéma

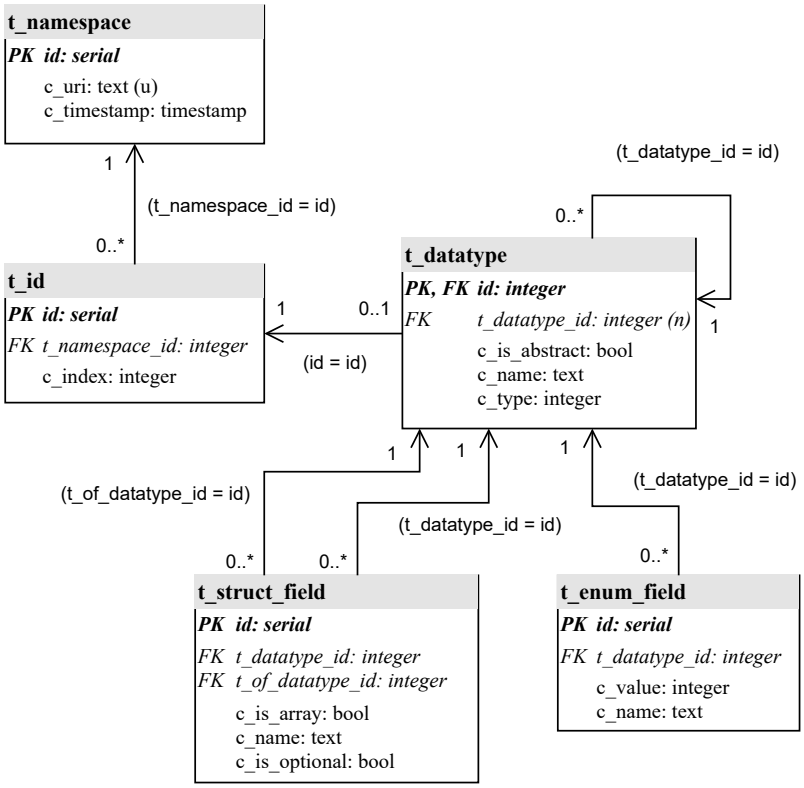
Strom objektů (t_object_tree)

Informační model OPC UA sice podporuje definice objektových typů, ale lze vytvářet objekty i bez jejich definice. V praxi se může jednat například o tvorbu objektů během inicializace na základě argumentů programu.



Obrázek 5.3: Detail na objektový strom (UML)

Přestože objekt vlastní definici, OPC UA podporuje jeho úpravu, tj. přidání/odebrání uzlů třídy Variable, Method atp. Všechny vytvořené objekty se nacházejí v podstromu uzlu Objects. Zde se ukládají i konkrétní hodnoty v uzlech třídy Variable. Pro současnou verzi databáze eviduje pouze strom objektů a datové typy. Jak je namodelováno na obrázku 5.3.



Obrázek 5.4: Detail na datový typ (UML)

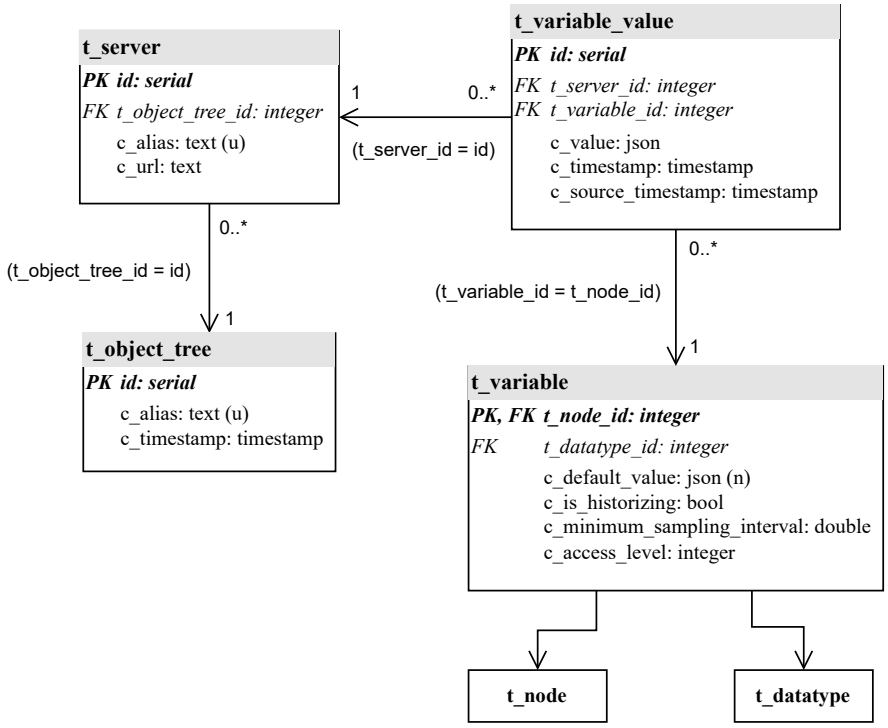
Oproti OPC UA se nejedná o celý graf uzlů včetně referencí. Účelem je zjednodušit formát dat, aby se dalo přímo dotazovat na datové typy či objekty. Aniž by bylo nutné hledat referenci a podle jména, případně dalších atributů, zjišťovat, zda představuje podtyp, komponentu atp., nebo získávat z obsahu proměnných argumenty metody, popisy atributů struktur atp.

Tedy informační modely OPC UA jsou předzpracovány před vložením do databáze. Na diagramu 5.3 lze spatřit, že některé uzly zmizely. Jedná se o uzly třídy DataType, jelikož datové typy jsou přímo modelovány. Uzly třídy View, ReferenceType, ObjectType a VariableType, které nikdy nejsou součástí podstromu Objects. Stejně tak argumenty metod neexistují jako uzly třídy Variable, nýbrž jsou modelovány přímo. Datový typ obsahuje cizí klíč na nadtyp. Uzel zase na rodiče. Pokud se jedná o základní datový typ OPC UA, ev. kořenový uzel, cizí klíče obsahuje null hodnotu.

V diagramech 5.3 a 5.4 nejsou vyznačeny IS-A hierarchie. Pro úplnost: jestliže je datový typ struktura, lze k němu přiřadit pouze *t_struct_field*, podobně pro výčtový typ. U uzlů platí, že uzel je třídy buď Object, nebo Variable či Method.

Server (t_server)

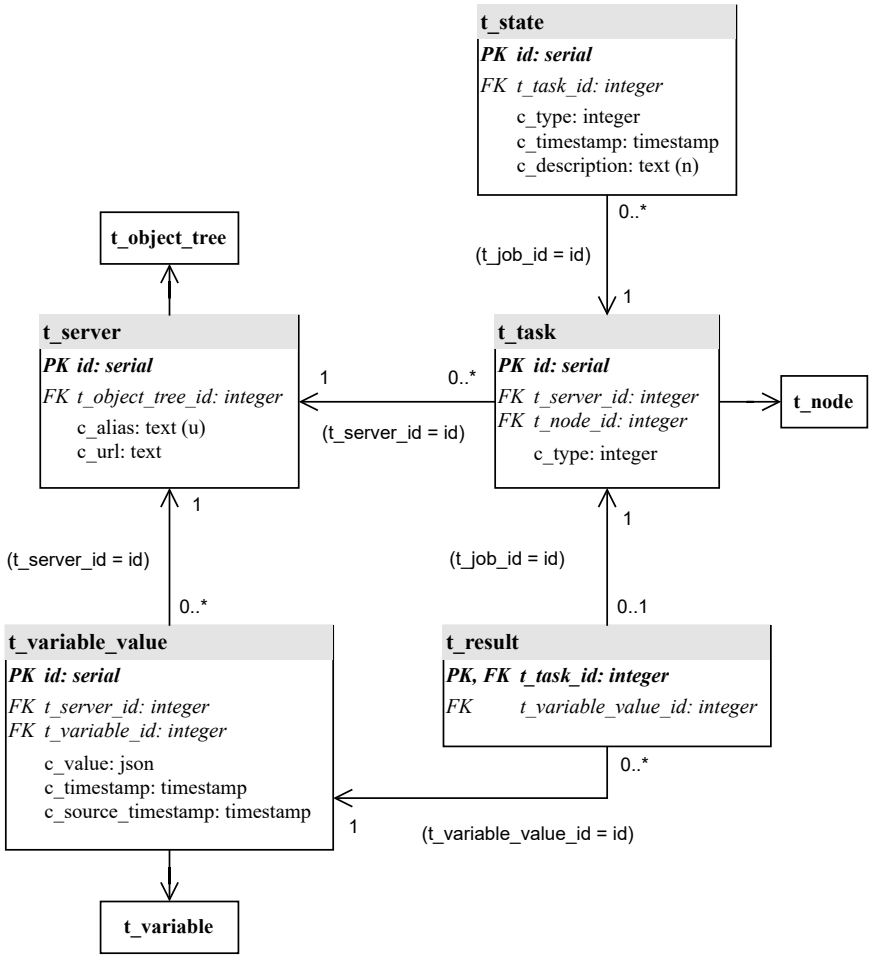
Aby nedocházelo k duplikaci dat, stromy objektů s datovými typy jsou nezávislé na serverech. Více serverů může mít stejný informační model. Liší se pouze hodnoty proměnných. Hodnoty se evidují v tabulce *t_variable_value*, která rozlišuje dva časy: *c_timestamp* představující čas uložení do databáze a *c_source_timestamp*, jenž obsahuje čas vzniku hodnoty na serveru. Hodnoty jsou ukládány jako řetězec obsahující hodnotu v daném kódování.



Obrázek 5.5: Detail na server (UML)

Úkol (t_task)

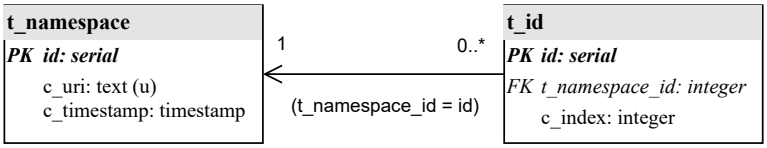
V samostatném schéma *s_task* se evidují úkoly, jejich historie stavů a potažmo výsledek. Na daný detail schéma se zaměřuje obrázek 5.6. Pro přímé spojení tabulek *t_server* a *t_task* tabulka *t_task* obsahuje cizí klíč na server. Navíc jedná se o nutnost pro úkoly, jejichž výsledkem není vyčtená hodnota. V případě úkolů, u nich vznikne záznam v *t_variable_value* a eviduje se vazba mezi úkolem, během kterého došlo k získání hodnoty, a hodnotou, dochází ke vzniku dvou směrů, jak spojit tabulky za účelem získání záznamu daného serveru. Nicméně, jak je již vidět na obrázku 5.6, nejedná se cyklus.



Obrázek 5.6: Detail na úkol (UML)

Identifikátor (t_id)

Návrh předpokládá použití NodeId typu Numeric. Dále klade podmínku na unikátnost NamespaceUri, tj. URI je přirozeným identifikátorem Namespace. V pojetí OPC UA musí být URI unikátní pouze v rámci serveru. Pro možnost využití přirozené identifikace je URI unikátní v rámci domény. Numerický identifikátor uzlu v Namespace se označuje jako index.



Obrázek 5.7: Detail na identifikaci uzlů (UML)

5.1.3 Integritní omezení

Kontrol integritních omezení na úrovni hodnot sloupců je ponechána aplikacím. Dále poškodí integritu, dojde-li na nerespektování IS-A hierarchií. Krom toho existují dvě místa, která by mohla poškodit integritu dat:

- výsledek úkolu – jestliže je výsledek hodnota proměnné, musí shodovat server, ke kterému patří hodnota, se serverem, na kterém byl úkol proveden.
- hodnota proměnné – musí se shodovat objektový strom, kam patří proměnná, s objektovým stromem, který se nachází na serveru.

5.2 Implementace

Vzhledem k možnému dalšímu zpracování dat v oblasti diagnostiky je zvoleno za databázový stroj PostgreSQL. Přestože PostgreSQL je objektově-relační databáze, objektová podpora nespadá mezi hlavní důvody, proč byl Postgres zvolen. Pod objektovou podporou si lze například představit dědičnost tabulek. [12] Výhodami Postgresu pro tento projekt jsou:

- Podpora rozsahu dat od řádu megabytů po terabyty.
- Podpora formátu JSON jakožto hodnoty sloupce. Stejně tak vkládání záznamů tabulek ve formátu XML.
- Lze nasadit jak na osobním počítači, tak na serveru. Což umožňuje lokální vývoj a testování.
- Podpora matematických nástrojů, např. MATLAB, a jazyků, např. R.
- Podpora rozšiřitelnosti funkcionalit na straně databázového serveru pomocí tvorby procedur, materializovaných pohledů a psaní skriptů/procedur i v jiných jazycích než SQL.
- Komunita a dostupnost materiálů, dokumentace.

5.3 Testování

Současná implementace neobsahuje spouštěče (*Triggers*), procedury (*Procedures*) ani jiné prvky aktivní databáze (*Active Database*) [17], tudíž není nutné testování. Je provedena pouze validace databázového schéma pomocí nástroje PgModeler a volání skriptu nad databází.

Kapitola 6

Scalable OPC UA

Kapitola popisuje dílčí projekt Systému pro sběr dat. Jedná se o knihovnu částečně implementující OPC UA. Projekt se soustředí na klientskou část a práci s daty. Jako zkrácený název se používá výraz Scopcua.

6.1 Návrh

V této části jsou popsány základní myšlenky a motivace návrhu. Tj. jak jsou data modelována, jaké je uspořádání balíčků tříd a které funkcionality knihovna podporuje.

6.1.1 Motivace

Lze nalézt vícero knihoven implementující OPC UA. Pro demonstraci je vybráno pět projektů, které mají nejvyšší množství hvězd na GitHub. [18]

projekt	**jazyk**	**počet hvězd**
open62541	C	1 719
OPC Foundation	C#	1 304
node-opcua	JavaScript	1 131
Free OPC-UA Library	C++, Python	1 164
Eclipse Milo	Java	726

Tabulka 6.1: Výběr OPC UA knihoven

Datové typy

Nastane-li ale situace, kdy je nutné modelovat vlastní datové typy, pouze knihovna node-opcua nepotřebuje vygenerovat definice. Důvodem je využití prototypové dědičnosti v jazyce JavaScript. Na druhou stranu node-opcua není aktualizovaná, tudíž nepodporuje OPC UA ve verzi 1.04. [19]

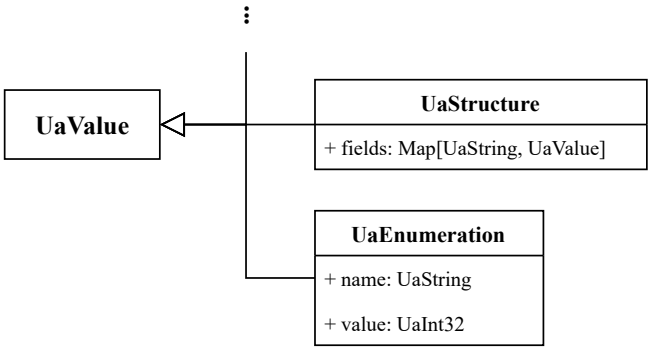
Projekty v jazycích C a C++ nejprve generují kód se samotnými definicemi a s třídami/strukturami datové typy popisující. Knihovny ho interně využívají ke znalosti, jak přepsat paměť podle zvolené operace, či jak velké místo de/alokovat. [20] [14] Oproti tomu knihovny v jazycích Java a C# se spoléhají převážně na reflexi. Opět je nutné vygenerovat kód s definicemi a s popisem datových typů. [21] [13] Při tom během vývoje dochází k úpravám informačních modelů. To sebou nese opětovné generování kódu. Navíc je-li nutné implementovat standard, např. IEC 61580-7-3 či IEC 61580-7-4, datových typů může být více než sto. V případě generátoru knihovny open62541 proces generování zabírá čas v řádech jednotek až desítek minut (v závislosti na parametrech počítače, kde je proces spuštěn).

API

Další motivací k tvorbě vlastní knihovny je vytvoření API, jenž snižuje uživatelovu požadovanou úroveň znalosti OPC UA. Navíc žádná z výše uvedených knihoven neimplementuje celý standard. [20] [13] [19] [21] [14] OPC UA svou univerzalitou tvoří rozsáhlý protokol. Proto se Scalable OPC UA zaměřuje na práci s daty a klientskou část. V této oblasti zmíněné projekty poskytují základní rozhraní (zavolat metodu, vyčíst hodnotu atd.). Do budoucna by bylo možné rozšířit tuto knihovnu i o Domain-Specific Language.

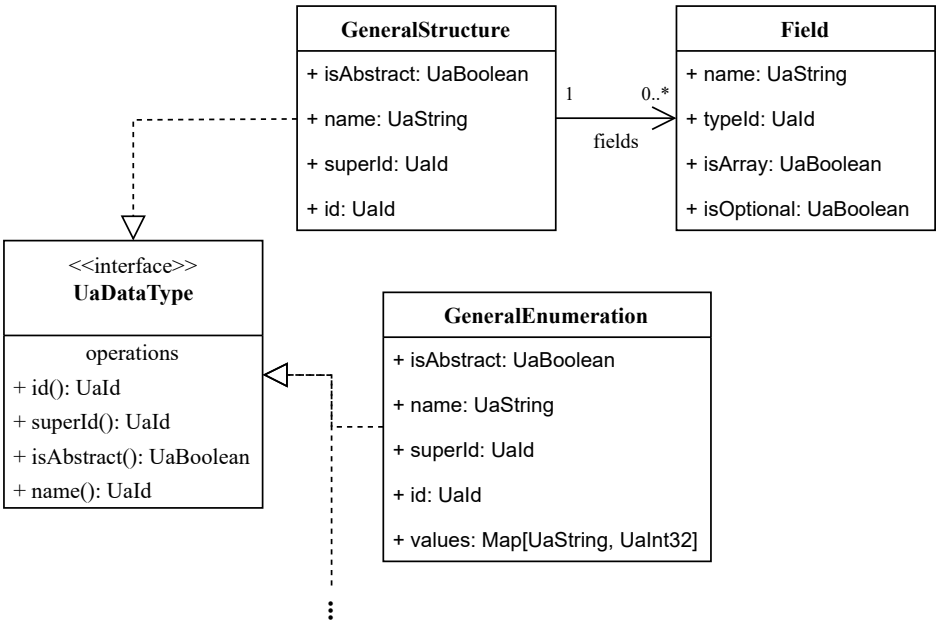
6.1.2 Model datových tříd

Odstranění závislosti na generovaném kódu s definicemi lze provést skrze datový model, který dovoluje obecné struktury a výčtové typy. Tj. struktura implementovaná jako třída s mapováním mezi názvem atributu a jeho hodnotou. A výčtový typ modelovaný jako třída s atributy popisující textovou a číselnou reprezentaci. Obrázky 6.1, 6.3, 6.2 zobrazující UML diagramy tříd zaměřené na dané detaily modelu datových tříd.



Obrázek 6.1: Hodnoty uživatelem definovaných datových typů (UML)

Podle diagramu 6.2 lze zpracovat uživatelsky definované struktury a výčty. Vedle nich zůstávají vestavěné datové typy. Vzhledem k jejich častému použití mají pro sebe deklarovány konkrétní třídy. V řešení se používají dvě abstraktní třídy *UaValue* a *UaDataType* s hierarchiemi podtříd. *UaValue* představuje hodnotu/instanci odpovídajícího datového typu. Zatímco *UaDataType* reprezentuje popis datového typu. Uživatel může přes *UaDataType.GeneralEnumeration* a *UaDataType.GeneralStructure* rozšiřovat datové typy o vlastní struktury a výčty.



Obrázek 6.2: Uživatelem definované datové typy (UML)

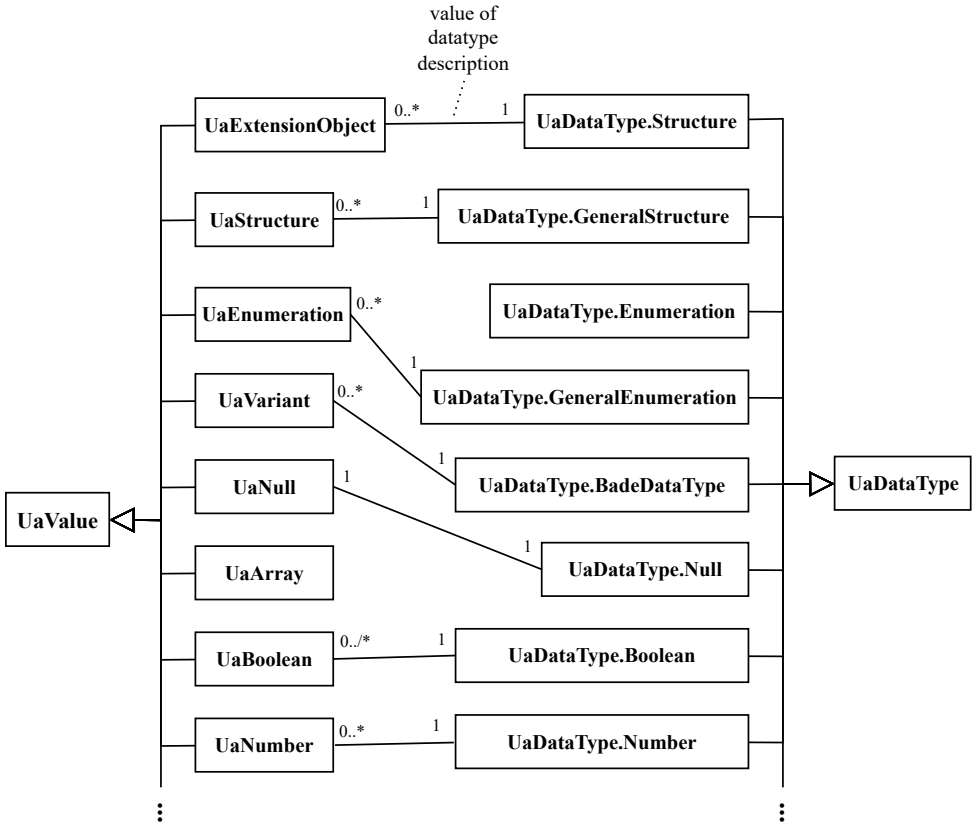
V současném řešení nelze přidávat vlastní jednoduché datové typy. Na druhou stranu vlastní jednoduché datové typy jsou pouze přejmenováním vestavěných jednoduchých datových typů. Mezi ně patří bezznaménkové 16-ti bitové celé číslo, číslo s plovoucí desetinnou čárkou s dvojitou přesností atd. Ostatní hodnoty kopírují systém vestavěných typů OPC UA (dále originální hierarchie). Shoduje se s knihovní hierarchií. Výjimku tvoří:

- *UaNull* – třída představující prázdnou hodnotu. Není v originální hierarchii. Nicméně standard Null hodnotu zná.
- *UaArray* – třída představující pole vybrané hodnoty. Není v originální hierarchii. Nicméně standard reprezentaci pole implementuje.
- *UaExtensionObject* – třída odpovídající abstraktní třídě *Structure* z originální hierarchie. Ponecháno kvůli převodům mezi datovými formáty/re-

prezentacemi. OPC UA využívá *ExtensionObject* pro přenos uživatelem definovaných struktur. Nicméně knihovna podporuje i přímý převod

- *UaVariant* – třída odpovídající abstraktní třídě *BaseDataType* z originální hierarchie. Ponecháno kvůli převodům mezi datovými formáty/reprezentacemi. OPC UA využívá *Variant* jako strukturu obsahující libovolný z vestavěných datových typů. Nicméně knihovna podporuje i přímý převod do *UaValue*.

Diagram 6.3 znázorňuje vazbu mezi modelem hodnoty a modelem odpovídajícího datového typu. Pro zjednodušení v diagramu nejsou zobrazeny všechny datové typy z důvodu vysokého počtu. Nicméně zachycuje všechny, jejichž vazba není triviální.

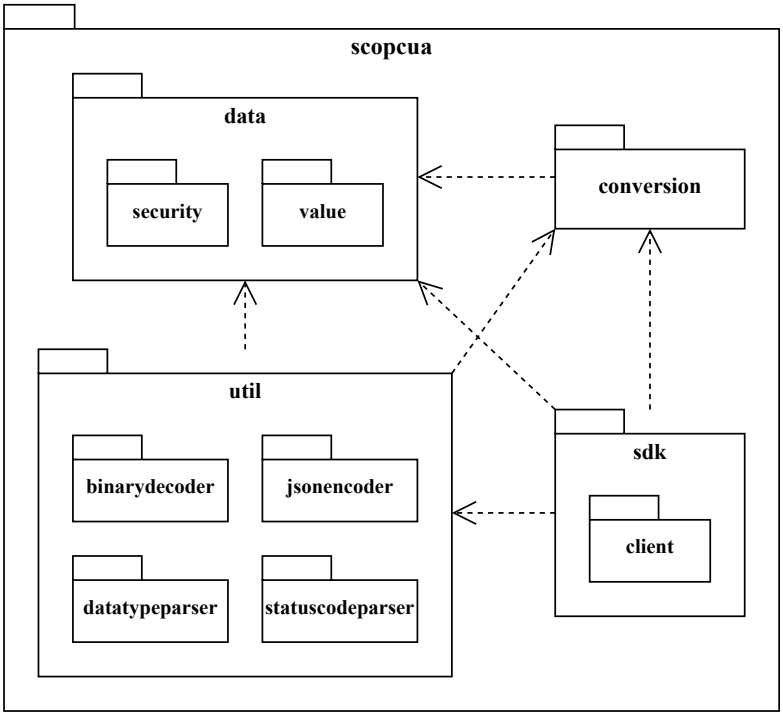


Obrázek 6.3: Vazba mezi vybranými hodnotami a jejich datovými typy (UML)

6.1.3 Struktura projektu

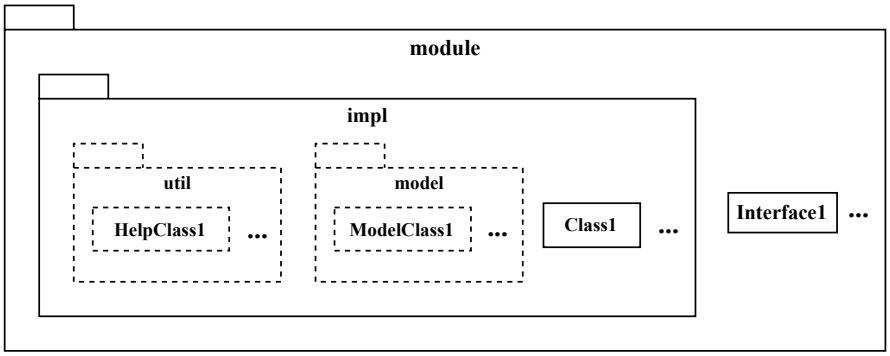
Zdrojové kódy jsou rozděleny do čtyř hlavních balíčků:

- *data* – obsahuje datový model. Skládá se z dalších balíčků podle významu nebo obsahuje přímo jednotlivé třídy.
- *conversion* – obsahuje třídy umožňující konverzi mezi modely.
- *util* – obsahuje moduly dodávající funkcionality do knihovny.
- *sdk* – obsahuje moduly dodávající vícero funkcionalit při využití ostatních balíčků.



Obrázek 6.4: Struktura balíčků knihovny (UML)

Balíčky *util* a *sdk* se skládají z modulů, které poskytují pro okolní svět rozhraní. Na obrázku 6.5 je načrtnuta jejich struktura. Vedle rozhraní je balíček *impl*, jenž obsahuje konkrétní implementace. Případné pomocné třídy se vkládají do balíčku *impl::util*. Stejně tak pomocné datové třídy do *impl::model*. Ty již nemusí obsahovat rozhraní. Pokud by řešení vyžadovalo vnořené moduly, vytvořily by se v balíčku *impl::util* moduly se stejnou strukturou jako je 6.5.



Obrázek 6.5: Struktura balíčků modulu (UML)

6.1.4 Datové formáty

Standard OPC UA spolu s IEC 61580 obsahuje okolo sta datových typů. [22][23] Hodnoty je nutné ukládat, ev. posílat po síti. OPC UA kódování jsou rozsáhlé, obsahují výjimky, což přináší zvýšení chybovosti a náročnosti implementace. Tudíž je navrhnut formát, který umožní implementaci bez množství podmínek.

Simplified OPC UA JSON

V OPC UA existují ke každému kódování hodnot datových typů dva způsoby: reverzibilní a ireverzibilní. První z nich je formát, který lze použít pro komunikaci se serverem. Druhý naopak ztrácí informace pro komunikaci, ale je informačně obsáhlejší. [24] Protože dochází ke sběru dat z vícero serverů, pro reverzibilní formát to znamená mapování *NamespaceIndex* a *NamespaceUri*, *ServerIndex* a *ServerUri* atp., během zpracování. Univerzálním řešením pro všechny servery je ireverzibilní formát. Tzn. místo *NamespaceIndex* je použit *NamespaceUri*, stejně tak *ServerUri* místo *ServerIndex*.

Aby se dosáhlo zkrácení definice a snížení počtu podmínek, výjimek u datové reprezentace, je vytvořen formát Simplified OPC UA JSON. Což je ireverzibilní OPC UA JSON kódování s těmito rozdíly:

- *UInt64, Int64* – 64-bitová celá čísla jsou dle OPC UA kódovány jako JSON String. Nicméně pro unifikaci formátu čísel jsou v Simplified reprezentovány jako JSON Number.
- *DateTime* – v OPC UA je čas „0001-01-01T00:00:00Z“ převáděn do JSON Null. V Simplified je ponechán.
- *ExtensionObject* – na rozdíl od OPC UA vzniklý JSON Object obsahuje pouze atribut body a *enType* (hodnota 0 pro UaByteString, 1 pro UaXmlElement).

- *prázdné atributy* – v OPC UA, pokud je hodnota Null, pak atribut se nenachází ve výsledném JSON Object. V případě Simplified je ponechána hodnota JSON Null.

- *konkrétní URI* – na místo *Namespace* s indexem „0“ pro OPC UA Foundation *Namespace* je uveden plný název. Stejně tak pro „1“ reprezentující *Namespace* aplikace.

6.1.5 Funkcionality

Knihovna poskytuje funkcionality:

- *UaBinaryDecoder* – modul poskytující dekódování binární reprezentace dat do datového modelu. V tuto chvíli lze nalézt implementaci standardní OPC UA binární kódování.

- *UaJsonEncoder* –modul poskytující kódování z datového modelu do formátu JSON. Knihovna momentálně implementuje převod do Simplified OPC UA JSON.

- *UaStatusCodeParser* – modul poskytující vyčtení *UaStatusCode* z textu. Do této verze lze dohledat implementaci vyčtení z formátu CSV.

- *UaDataTypeParser* – modul poskytující vyčtení popisu datového typu z textu. Knihovna implementuje vyčtení z XML souboru *NodeSet2* s OPC UA verzí 1.04.

Dále implementuje synchronního klienta. Jeho rozhraní obsahuje metody na připojení, odpojení, vyčtení hodnoty a vyčtení *NamespaceArray*. Podporuje připojení dvou druhů:

- *anonymní* – klient nevyužívá žádné zabezpečení. V OPC UA se zpravidla takovýto klient používá k prohledávání informačního modelu, zjišťování dostupných služeb a způsobů bezpečného připojení. [1]

- *zabezpečené* na úrovni jméno, heslo a šifrování skrze Public-Key Infrastructure – klient splňující požadavky na připojení k produkčním serverům.

6.2 Implementace

Vzhledem k požadavku na technologii běžící na JVM je zvolen jazyk Scala. Oproti Javě umožňuje přesnější definování datových tříd, např. rozdíl mezi třídou a objektem, a využití pattern matching. [25] V případě další práci na projektu by poskytla například čitelnější zápis DSL.

6.3 Testování

Používají se jednotkové a integrační testy. Jednotkové se týkají především metod pomocných tříd, integrační testují proti rozhraní modulů. Testování OPC UA klienta je vyloženě skrze integrační testy s běžícím serverem, aby se dosáhlo i případného odhalení chyb v komunikaci.

Pro testování klienta jsou vybrány dvě implementace serveru. První z nich je Eclipse Milo OPC UA Demo Server a druhou vlastní testovací server vytvořený pomocí knihovny open62541. Demo Server má již připravenou tvorbu certifikátu, uživatelské role, přístupové body, vlastní datové typy a informační model. [26] Což urychluje testování v počáteční fázi vývoje. Nicméně produkční servery využívají projekt open62541. Obě knihovny jsou v aktivním vývoji a lze očekávat určitou chybovost. Proto je nutné testovat klienta i proti serveru od open62541. Implementace se nachází jako vedlejší projekt v jazyce C++.

Také se testuje použití datových typů z informačních modelů IEC 61580-7-3 a IEC 61580-7-4, jelikož knihovna musí tyto standardy podporovat. Převážně se tyto typy používají při testování modulů, např. dekódování z binárního formátu.

Aby se dala konfigurovat automatizace testů v závislosti na prostředí, např. GitLab CI/CD vs. lokální počítač, a spouštění pomocných serverů, jsou testy potřebující běžící testovací server označeny štítkem *NeedsRunningServer*. Třídy s testy využívají knihovnu ScalaTest.

Kapitola 7

Aplikace

Kapitola popisuje návrh, implementaci a testování aplikace. Jedná se o dílčí projekt Systému pro sběr dat. Využívá projekty Databáze a Scalable OPC UA.

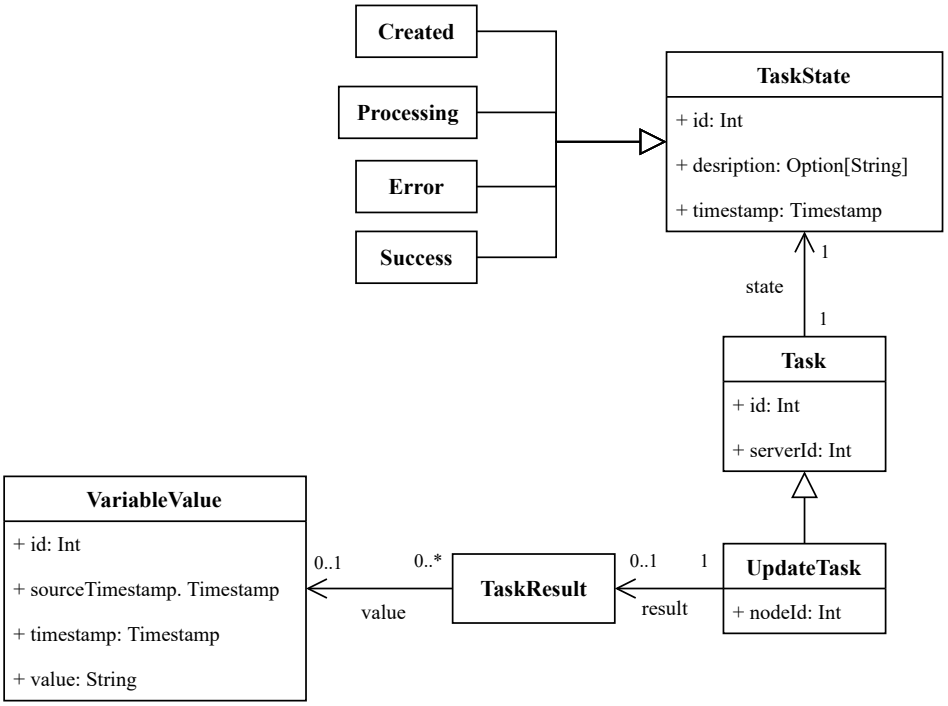
7.1 Návrh

V návrhu jsou využity znalosti domény a požadavků. Část doménového modelu je transformována do modelu datových tříd. Dále je popsána zvolená architektura. Cílem návrhu je vytvořit aplikaci umožňující snadné přidávání dalších funkcionalit či změnu stávajících. Současný návrh počítá s dvěmi a to vyčtení informace o úkolu (*Task*) a provedení *UpdateTask*. Zmíněné úkoly lze chápat jako procesy. Nicméně se jedná o jednoduché příkazy, tudíž nejsou podrobně modelovány, například pomocí UML diagramu aktivit.

7.1.1 Model datových tříd

Obrázek 7.1 ukazuje UML diagram tříd. Datové třídy jsou třídy, které slouží pouze jako nosiče, neimplementují žádnou nebo minimální logiku. Někdy jsou označovány jako *Value Object*. [27] Návrh je koncipován tak, aby při vyčítání informací z databáze docházelo k získání nezbytně nutných záznamů. Tudíž třídy obsahují aktuální hodnoty, nikoliv i historii. Třída *Task* vlastní současný stav, *TaskState*, přestože by bylo možné, aby obsahovala všechny stavy, které kdy měla, například v nějaké kolekci.

Návrh odpovídá současným požadavkům. Nicméně vzhledem k očekávanému rozšiřování je již naznačen koncept výsledků pro další podtypy třídy *Task*, a tím je třída *TaskResult*. Kdyby k rozšíření dojít nemělo, např. u třídy *VariableValue* by stačilo pouze, aby byla navázána přímo na *UpdateTask*. Na druhou stranu návrh nepočítá i s úkoly mimo informační modely. Není specifikováno v požadavcích, zda mohou v budoucnu vzniknout úkoly, které nebudou vázány na uzly, např. přidání serveru.

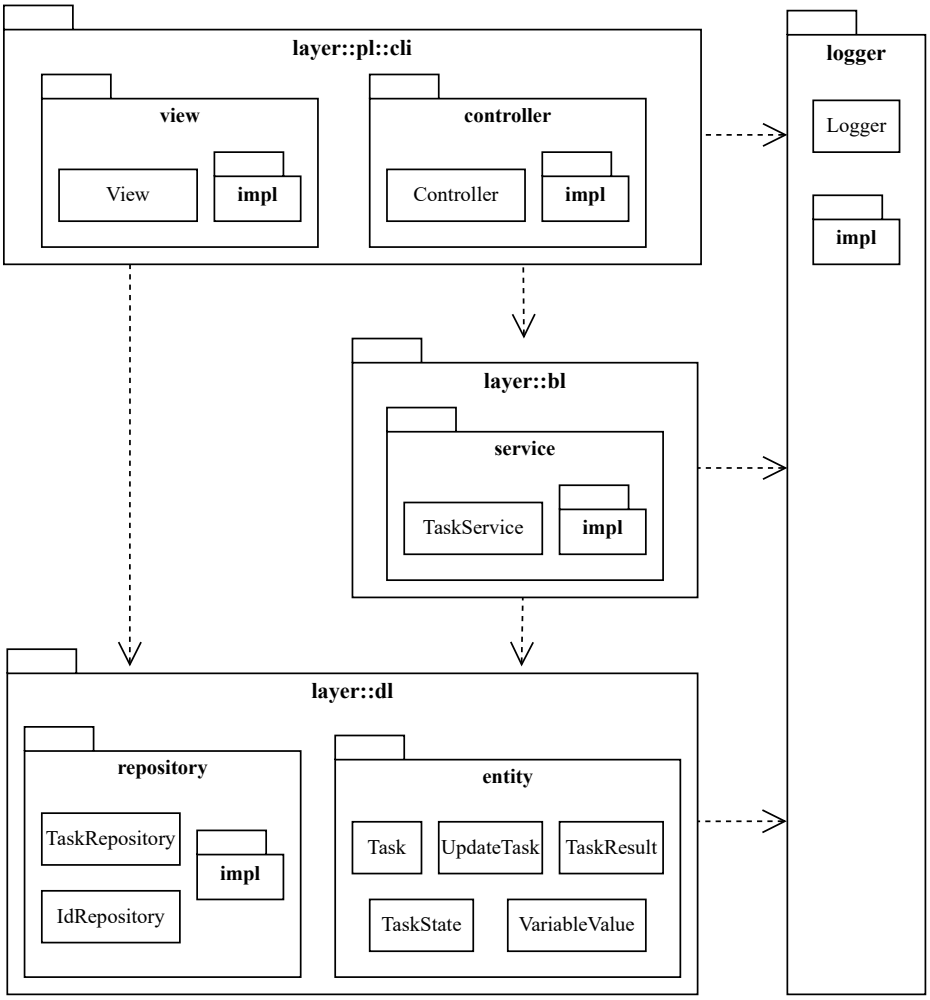


Obrázek 7.1: Model datových tříd (UML)

7.1.2 Architektura

Přestože existuje návrh na produkční databázi, musí architektura aplikace počítat s možnou změnou v uložení dat. Dále ve se očekává uživatelské rozhraní přes CLI. Nicméně v dalších verzích dojde k přidání rozhraní, které umožní komunikaci pro ostatní aplikace. Zadávání požadavků se stane automatizovaným. Požadavkům tedy vyhovuje třívrstvá architektura. Ta se skládá z vrstev datové, byznys logiky a prezentační. [27] Protože aplikace musí umět podporovat i zobrazení stávajících úkolů, jedná se o architekturu třívrstvou relaxovanou. K pouhému zobrazení nemusí být nutná aplikační logika jakožto mezivrstva mezi daty a prezentací.

Obrázek 7.2 zobrazuje UML diagram balíčků, kde je namodelována závislost mezi vrstvami a tzv. cross-cutting concern. Pojmenování je zkratkami: *dl* pro datovou vrstvu, *bl* pro byznys vrstvu a *pl* pro prezentační vrstvu. Z hlediska zdrojových kódů aplikační vrstvy se nacházejí v balíčku *layer*. Na stejné úrovni jako *layer* jsou balíčky *logger*, *clock* nebo *ioc*, který obsahuje třídy umožňující načtení konfigurace a inicializaci tříd z *layer*. Například se jedná o vytvoření klientů OPC UA, jelikož je nutné načíst datové typy, případně i certifikáty.



Obrázek 7.2: Architektura aplikace (UML)

Datová vrstva

Smyslem datové vrstvy je zapouzdřit užití konkrétní technologie pro uložení dat. Poskytuje rozhraní. V tomto případě rozhraní repozitářů. Návrhový vzor Repository podporuje krom přístupu k datům skrze identifikátory i jednoduchou logiku. Tou mohou být různá vyhledávací specifika či stránkování. Dále tato vrstva obsahuje datové třídy, jejichž instance poskytuje skrze zmíněné repozitáře. Pro instance podporuje CRUD operace. Repozitáři jsou:

- *IdRepository* – poskytuje převod mezi přirozenou identifikací uzlu v informačním modelu OPC UA, tj. kombinace index v rámci *Namespace* a *NamespaceUri*, a umělou identifikací, tj. sloupec *id* tabulky *t_node* v databázi.

- *TaskRepository* – poskytuje CRUD operace pro abstraktní třídu *Task*, včetně operací pro podtypy a update stavu implementace třídy *Task*.

Vrstva byznys logiky

Jak název napovídá, v této vrstvě se skrývá aplikační logika. Zde se vykonávají operace a požadavky zadané uživatelem či jinou aplikací. Pro zisk/uložení informací využívá třídy z datové vrstvy. Funkcionality poskytuje skrze rozhraní. Implementace a rozhraní se pojmenovávají jako *Services*. V současné chvíli existuje jedna služba a to *TaskService*. Ta se stará o vykonávání úkolů zadané uživatelem/jinou aplikací. Současná verze podporuje pouze *UpdateTask*, který vyčte hodnotu ze zvoleného serveru a uzlu a uloží do databáze. Následně vrací výsledek této operace.

Prezentační vrstva

Prezentační vrstva obsahuje balíček *cli* představující implementaci CLI. Návrh je rozdělen do dvou komponent: *View* a *Controller*. Kdy *View* se stará o parsování vstupu a zobrazování výstupu, zatímco *Controller* poskytuje volání služeb či repozitářů nebo odchytává výjimky a dále předává jejich obsah *View*. *View* je tedy závislé na *Controller*. Spuštěním programu se uživatel dostane do smyčky, kdy je čten jeho vstup a následně zpracován. Aplikace podporuje příkazy:

- „help“ – zobrazí nápovědu.
- „stop“ – zastaví program.
- „read [id]“ – zobrazí informace o *Task* se zvoleným identifikátorem.
- „create update [data]“ – vytvoří *UpdateTask* pro zvolený server a uzel na základě dat ve formátu JSON. Ten obsahuje JSON Object s atributy nodeId a serverId. Nakonec zobrazí informace o vytvořeném *Task*.

Zobrazené úkoly ve *View* jsou ve formátu JSON.

7.2 Implementace

K implementaci je vybrán jazyk Scala. Jakožto built-tool je zvolen projekt sbt. Čímž se plní nefunkční požadavek na technologii podporující JVM. Zároveň bude možné použít knihovnu Scalable OPC UA, která je implementována ve Scale.

7.2.1 Vrstva byznys logiky

Dalším důvodem pro Scalu je využití projektu Akka pro asynchronní zpracování úkolů. Současná implementace *TaskService*, tedy *SyncTaskService*, pracuje synchronně. Což bude nutné změnit v okamžiku, kdy se budou implementovat úkoly trvající v řádech hodin či dnů. Pak bude namístě asynchronní chování.

7.2.2 Datová vrstva

Implementace datové vrstvy využívá knihovnu Slick. To přináší výhody a nevýhody oproti obligátnímu použití ORM.

Výhodou je především možnost zvolit vlastní implementaci vyčtení/uložení dat. V případě aplikace dochází k přirozené optimalizaci, kdy jsou vyčteny pouze nezbytně nutné záznamy, nikoliv entita se všemi závislostmi a informacemi. Další výhodou je široká komunita a dostupnost materiálů oproti ostatním scalovským knihovnám pro práci s databází. Částečně dáno faktem, že knihovnu tvoří autoři jazyka Scala. [6]

Nevýhodou je potřeba něco implementovat sám, tzn. riziko chybovosti a nutnost vytvořený kód otestovat. Další slabinou jsou transakce. Knihovna je sice poskytuje, ale jsou tvořeny knihovními akcemi. [6] Což svádí k zanášení implementace datové vrstvy do ostatních vrstev (repozitář by musel vracet generikum DBIO).

7.2.3 Prezentační vrstva

Prezentační vrstva je implementována ve Scala za použití knihovny Lift-JSON pro parsování JSON formátu. K implementaci by se dala využít i knihovna poskytnutá projektem sbt. Obsahuje několik funkcí pro programování CLI. Nicméně v tuto chvíli není implementace natolik složitá, aby vyžadovala přidání dalších technologií.

7.2.4 Konfigurace aplikace

Aplikace obsahuje konfigurační soubor splňující formát projektu Typesafe Config. Navíc obsahuje informační modely ve formátu OPC UA NodeSet2 XML, které jsou nutné pro získání datových typů pro korektní inicializaci OPC UA klientů. Jelikož v případě, že klient navazuje zabezpečenou komunikaci, tj. jméno, heslo, privátní klíč a certifikát, jakožto veřejný klíč serveru, musí obsahovat korektní složku pro PKI. Ostatní instance, resp. jednotlivé komponenty aplikace, jsou alokovány voláním daných konstruktorů/factory methods.

7.3 Testování

Testy jsou vytvořeny pomocí knihovny ScalaTest a ScalaMock. V závislosti, zda se jedná o pomocnou třídu či implementaci rozhraní, jsou užity integrační a jednotkové testy. Integrační testy se používají zpravidla pro testování vůči rozhraní.

Dále je použit štítek *NeedsRunningDb* pro označení testů, které vyžadují běžící databázi. Typicky se jedná o testy datové vrstvy. Ostatní vrstvy závislé na datové používají princip tzv. mocking či testovací implementaci repozitářů. Třídy závislé na OPC UA klientovi rovněž používají mocking či testovací implementaci. Zabraňuje to psaní zbytečných, resp. duplicitních testů. Klient byl již otestován v rámci svého projektu Scalable OPC UA.

Kapitola 8

Možná vylepšení

Kapitola se věnuje částem návrhu, které do budoucna mohou být vylepšeny. Jednotlivé části pochází ze všech tří dílčích projektů nebo existující řešení doplňují o další nápady na aplikace či knihovny.

8.1 Kontinuální integrace/vývoj

Verzování v průběhu vývoje probíhalo na platformě GitLab. Vzhledem k tomu, že současný firemní GitLab nepodporuje CI/CD, nebylo v práci řešeno automatické sestavení a spouštění testů na vzdáleném repozitáři. Nicméně automatické testy jsou implementovány s využitím knihovny ScalaTest. Tudíž možným vylepšením je doplnění CI/CD do projektu.

8.2 Zadávání příkazů aplikaci

Ačkoliv v zadání bakalářské práce je uvedeno, že zadávání příkazů aplikaci probíhá přes databázi, v projektu bylo vytvořeno rozhraní přes příkazovou řádku. Rozhraní přes databázi nakonec nebylo specifikováno v požadavcích. Rozhraní je momentálně určeno pro ruční testování a eventuální představení funkčnosti. Příkazová řádka se tak jeví jako lepší volba – databáze by vyžadovala například klienta navíc.

8.3 Modul vykonávající úkoly (TaskService)

Požadavky kladené na tuto práci byly pouze na implementaci úkolu UpdateTask. Jedná se o vyčtení hodnoty se serveru a uložení do databáze. Očekává se, že budou dále implementovány úkoly využívající OPC UA služby: odběr, historické čtení, volání metody, zápis hodnoty. Podstatným rozdílem mezi subskripcí a ostatními je v délce trvání. Současné zpracování úkolů probíhá synchronně. Což se stane špatným návrhem v případě subskripce, jelikož ta může

trvat i v řádech hodin/dní. Cestou ven se jeví asynchronní zpracování. K tomu lze využít například projekt Akka, který implementuje návrhový vzor Actor Model. [28]

8.4 Inicializace databáze

Projekt s databází poskytuje návrh, schéma v aplikaci pgmodeler a tzv. SQL create script, jehož spuštěním se vytvoří daná schémata v databázi. Chybí však naplnění databáze daty, resp. informačními modely. Možným dalším krokem je vytvoření kódu využívající knihovnu Scalable OPC UA, který modely z OPC UA NodeSet2 souboru přečte a vloží do databáze. K parsování je možné využít kód cizích knihoven nebo rozšířit stávající knihovnu Scalable OPC UA. Ta momentálně z informačního modelu získává pouze datové typy, nikoliv uzly.

8.5 Inicializace aplikace

Současné řešení nevyužívá cizí implementaci DI, jelikož obsahuje řádově jednotky komponent. Při navýšení počtu komponent by se měl užít framework či knihovna implementující DI. Ruční tvorba komponent by se tak stala nepřehledná a komplikovaná. Nicméně v obou situacích je nutné OPC UA klienty inicializovat – v tomto případě skrze OPC UA NodeSet2 soubory s informačními modely a konfigurační soubory aplikace.

Závěr

Cílem bakalářské práce byl vývoj Systému pro sběr dat ze zařízení, které komunikují pomocí standardu OPC UA. Práce byla zadána společností ModemTec. V současné době je využívána a dále rozšiřována. Cíl byl rozdělen na čtyři části.

Prvním z dílčích cílů bylo představení OPC UA. Došlo tedy na popis standardu. Text se zejména soustředil na aspekty spjaté s vyčítáním dat, tj. informační model, služby poskytnuté serverem a bezpečnost prvky komunikace klient-server. Vývojáři tento text využívají pro prvotní seznámení s protokolem. Krom OPC UA byly stručně představeny další technologie, které byly zapotřebí v průběhu vypracování práce.

Ke splnění dílčích cílů dva a tři byla nutná analýzu domény a požadavků kladených na Systém. Byl představen zadavatel a následně byly rozlišeny jeho požadavky do dvou skupin: funkční a nefunkční. Taktéž vznikl konceptuální model v notaci OntoUML, kde byly zobrazeny data a jejich vztahy. Z analýzy dále vyplynulo potřeba rozdělit práci do tří dílčích projektů: databáze, knihovna pro OPC UA a aplikace.

Druhým dílčím cílem byla samotná tvorba databáze. Zde se na doménu dívalo z pohledu uložení dat. Což poskytlo půdu pro rozhodnutí využít relační databázové systémy, a tudíž vytvořit relační datový model v notaci UML. Následně byl vybrán PostgreSQL jakožto databázový stroj pro implementaci databázového schéma. Nakonec byla provedena jeho validace.

Třetím dílčím cílem byla tvorba aplikace. Během vývoje vyplynula potřeba oddělit od aplikace knihovnu věnovanou OPC UA (Scalable OPC UA). Tudíž návrh, implementace a testování OPC UA klienta a funkcionalit bylo řešeno samostatně. Zvoleným jazykem se stala Scala. Na základě analýzy požadavků a domény, byl zhotoven návrh aplikace, který se skládal z modelu datových tříd v UML a popisu třívrstvé architektury. Současná verze byla navrhnuta s rozhraním přes příkazovou řádku. Projekt aplikace využíval všechny předchozí dílčí projekty. Implementace byla zhotovena s využitím technologií Scala jako programovací jazyk, PureConfig pro načítání konfigurace, Slick podporující

přístup k databázi a Lift-JSON poskytující funkce pro práci s formátem JSON. Nakonec proběhlo otestování.

Čtvrtým a závěrečným dílčím cílem bylo shrnutí výsledků v závěru bakalářské práce a nastínění možných vylepšení do budoucna. Byla zmíněna současná implementace a nápady na změny.

Literatura

[1] MAHNKE, W.; LEITNER, S.-H.; DAMM, M.: *OPC Unified Architecture*. Berlin: Springer-Verlag Berlin Heidelberg, 2009, ISBN 978-3-540-68898-3.

[2] OPC Foundation: *OPC UA Online Reference, Part 3: Address Space Model, Summery of Attributes of the NodeClasses*. [online], 2021, [cit. 2021-05-06]. Dostupné z: `https://reference.opcfoundation.org/v104/Core/docs/Part3/5.9/`

[3] OPC Foundation: *OPC UA Online Reference, Part 4: Services*. [online], 2021, [cit. 2021-05-06]. Dostupné z: `https://reference.opcfoundation.org/Core/docs/Part4/`

[4] OPC Foundation: *OPC UA Online Reference: Online versions of OPC UA specifications and information models*. [online], 2021, [cit. 2021-05-06]. Dostupné z: `https://reference.opcfoundation.org/`

[5] BAELDUNG: *Difference Between JVM, JRE, and JDK*. [online], 2019, [cit. 2022-03-14]. Dostupné z: `https://www.baeldung.com/jvm-vs-jre-vs-jdk`

[6] LIGHTBEND INC.: *Slick*. [online], 2022, [cit. 2022-03-15]. Dostupné z: `https://scala-slick.org/`

[7] ARTIMA: *ScalaTest*. [online], 2022, [cit. 2022-03-17]. Dostupné z: `https://www.scalatest.org/`

[8] LIGHTBEND INC.: *Typesafe Config*. [online], 2022, [cit. 2022-03-31]. Dostupné z: `https://lightbend.github.io/config/`

[9] 47 DEGREES: *PureConfig*. [online], 2019, [cit. 2022-03-17]. Dostupné z: `https://pureconfig.github.io/`

[10] LIGHTBEND INC.: *sbt*. [online], 2022, [cit. 2022-03-20]. Dostupné z: `https://www.scala-sbt.org/`

[11] LIFT TEAM: *Lift-JSON*. [online], 2011, [cit. 2022-03-20]. Dostupné z: https://github.com/lift/lift/tree/master/framework/lift-base/lift-json

[12] POSTGRESQL: *PostgreSQL: About*. [online], 2022, [cit. 2022-03-19]. Dostupné z: https://www.postgresql.org/about/

[13] HERRON, Kevin aj.: *Eclipse Milo*. [online], 2022, [cit. 2022-03-10]. Dostupné z: https://github.com/eclipse/milo

[14] OPEN62541: *open62541*. [online], 2022, [cit. 2022-03-10]. Dostupné z: https://open62541.org/

[15] MODEMTEC S.R.O: *ModemTec*. [online], 2022, [cit. 2022-03-30]. Dostupné z: https://modemtec.cz/cs/

[16] ALTEXSOFT: *Functional and Nonfunctional Requirements: Specification and Types*. [online], 2021, [cit. 2022-04-10]. Dostupné z: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/

[17] ELMASRI, R.; NAVATHE, S.: *Fundamentals of database systems*. Boston: Pearson, čtvrté vydání, 2004, ISBN 0-321-12226-7.

[18] GITHUB INC.: Repository Results. [online], 2022, [cit. 2022-03-30]. Dostupné z: https://github.com/search?o=desc&q=opc+ua&s=stars&type=Repositories

[19] NODEOPCUA: *NodeOPCUA*. [online], 2022, [cit. 2022-03-27]. Dostupné z: https://node-opcua.github.io/

[20] FREE OPC-UA LIBRARY: *FreeOpcUa*. [online], 2017, [cit. 2022-03-27]. Dostupné z: https://github.com/FreeOpcUa/freeopcua

[21] OPC UA FOUNDATION: *Official OPC UA .NET Standard Stack from the OPC Foundation*. [online], 2022, [cit. 2022-03-27]. Dostupné z: https://github.com/OPCFoundation/UA-.NETStandard

[22] OPC UA FOUNDATION: *OPC UA Online Reference, IEC61850-7-3*. [online], 2022, [cit. 2022-03-29]. Dostupné z: https://reference.opcfoundation.org/IEC61850-7-3/

[23] OPC UA FOUNDATION: *OPC UA Online Reference, IEC61850-7-4*. [online], 2022, [cit. 2022-03-29]. Dostupné z: https://reference.opcfoundation.org/IEC61850-7-4/

[24] OPC UA FOUNDATION: *OPC UA Online Reference, Part 6: Mappings*. [online], 2022, [cit. 2022-03-29]. Dostupné z: https://reference.opcfoundation.org/v104/Core/docs/Part6/

[25] ÉCOLE POPLYTECHNIQUE FEDERALE LAUSSANE: *The Scala Programming Language*. [online], 2022, [cit. 2022-03-27]. Dostupné z: `https://www.scala-lang.org/`

[26] HERRON, Kevin aj.: *Eclipse Milo OPC UA Demo Server*. [online], 2022, [cit. 2022-03-10]. Dostupné z: `https://github.com/digitalpetri/opc-ua-demo-server`

[27] FOWLER, M.: *Patterns of Enterprise Application Architecture*. Boston: Addison Wesley, první vydání, 2002, ISBN 0-231-12742-0.

[28] LIGHTBEND INC.: *Akka*. [online], 2022, [cit. 2022-03-31]. Dostupné z: `https://akka.io/`

Příloha A

Seznam použitých zkratek

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

AST Abstract Syntax Tree

CA Certificate Authority

CD Continuous Deployment

CI Continuous Integration

CLI Command-Line Interface

COM Component Object Model

CRUD Create, Read, Update, Delete

DCOM Distributed Component Object Model

DI Dependency Injection

DSL Domain-Specific Language

GPL General Public License

GUID Globally Unique Identifier

HOCON Human-Optimized Config Object Notation

HTTPS Hypertext Transfer Protocol Secure

IDE Integrated Development Environment

IEC International Electrotechnical Commission

JSON JavaScript Object Notation

JVM Java Virtual Machine

NN nízké napětí

OOP Object-Oriented Programming

OPC UA Open Platform Communications Unified Architecture

ORM Object–Relational Mapping

PKI Public-Key Infrastructure

RDBMS Relational Database Management System

REPL Read–Eval–Print Loop

SBT Simple Build Tool

SQL Structured Query Language

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

VV vysoké napětí

XML Extensible Markup Language

Příloha **B**

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `text` ..... písemná část práce
  - `BP_Codl_Dominik_2022.pdf` ..... text práce ve formátu PDF
  - `latex_zdroj` ..... zdrojové soubory písemné části práce
  - `obrazky_zdroj` ..... zdrojové soubory obrázků
- `dcs` ..... nepísemná část práce – Systém pro sběr dat
  - `database` ..... projekt databáze
  - `scalable` ..... projekt knihovna OPC UA
  - `application` ..... projekt aplikace
  - `test_server` ..... projekt testovací OPC UA server