Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

BAKALÁŘSKÁ PRÁCE

OPC UA servery a jejich použití pro přenos dat řídicích systémů

Plzeň 2015 Tomáš Ausberger

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne ..... Podpis autora .....

Poděkování

Děkuji panu Ing. Pavlu Baldovi, PhD. za velmi zajímavé téma práce, Ing. Milanu Štětinovi za pomoc při práci s REXem a diagnostickým protokolem, Ing. Romanu Pišlovi za užitečné nápady při navrhování serveru a Martinu Úblovi za pomoc se záludnostmi programovacího jazyka C++.

Abstrakt

V této práci jsou popisovány základní principy technologie OPC UA, výhody této technologie oproti klasickému OPC a její využití v průmyslové automatizaci. Práce obsahuje srovnání nástrojů vhodných k vytváření OPC UA aplikací v jazyce C++. V poslední části práce je zdokumentován OPC UA server, který kopíruje strukturu bloků spuštěné exekutivy řídícího systému REX do svého adresního prostoru a umožňuje OPC UA klientům monitorování, čtení a zápis parametrů bloků exekutivy.

Klíčová slova: OPC, OPC Unified Automation, REX, C++, OPC UA toolkit, průmyslová automatizace

Abstract

This thesis describes basic principles of the OPC UA technology, advantages of the OPC UA against the OPC Classic and its utilization in industrial automation. The thesis contains a comparison of tools for building OPC UA aplications in C++. The OPC UA server, which copies the structure of block from a REX executive into its address space and allows OPC UA clients to monitor, read and write values of the executive blocks parameters, is documented in the last part of the thesis.

Keywords: OPC, OPC Unified Automation, REX, C++, OPC UA toolkit, industrial automation

Obsah

1 Úvod **4**
1.1 Motivace . . . 4
1.2 Postup řešení . . . 4

2 Řídicí systém REX **6**
2.1 Diagnostický protokol . . . 6

3 Technologie OPC UA **8**
3.1 Základní principy . . . 8
3.2 Profily . . . 9
3.3 Komunikace a zabezpečení . . . 9
3.3.1 Transportní vrstva . . . 9
3.3.2 Komunikační vrstva . . . 9
3.3.3 Aplikační vrstva . . . 10
3.4 Adresní prostor a uzly . . . 10
3.4.1 Atributy a třídy uzlů . . . 11
3.4.2 Hodnoty . . . 11
3.5 Služby . . . 12
3.5.1 Průzkumné služby (Discovery) . . . 12
3.5.2 Zabezpečený kanál (SecuredChannel) . . . 13
3.5.3 Relace (Session) . . . 13
3.5.4 Správa uzlů (NodeManagement) . . . 13
3.5.5 Procházení (Browse) . . . 13
3.5.6 Pohled (View) . . . 13
3.5.7 Dotazování (Query) . . . 13
3.5.8 Atributy (Attribute) . . . 14
3.5.9 Metody (Method) . . . 14
3.5.10 Monitorování (Monitor) . . . 14
3.5.11 Odběr (Subscription) . . . 14
3.6 Informační model . . . 14
3.7 Mapování . . . 15
3.7.1 Převod a zabezpečení dat . . . 15
3.7.2 Binární kódovaní . . . 16
3.7.3 XML kódování . . . 16
3.7.4 OPC TCP . . . 16
3.7.5 SOAP (Simple Object Access Protocol) . . . 17
3.8 Rozšířené proměnné . . . 17
3.9 Programy . . . 17
3.10 Historický přístup k datům . . . 17

4 Nástroje pro vývoj OPC UA serverů **19**
4.1 Nástroje k vytvoření serveru v C++ . . . 19
4.1.1 UA Communication Stack . . . 19
4.1.2 Softing – OPC UA C++ Development Toolkits . . . 19
4.1.3 Unified Automation – ANSI C based OPC UA Server SDK 20

4.1.4 Unified Automation – C++ based OPC UA Server SDK . 20
4.1.5 MatrikonOPC – OPC UA Embedded Server SDK 21
4.2 Výběr nástroje 21
4.2.1 Zvolený nástroj 21
4.3 Doplňující nástroje 22
4.3.1 UaExpert 22
4.3.2 UaDemoServer 22
4.3.3 Softing OPC Client 22
4.3.4 Wireshark 24

5 Nástroje pro OPC UA firmy Softing Industrial Automation GmbH 25
5.1 Životnost objektů 25
5.2 Inicializace aplikace 25
5.3 Creator 26
5.4 Informační model 26

6 Vývoj OPC UA serveru pro řídicí systém REX 28
6.1 Využití příkladu z tutoriálu toolkitu 28
6.1.1 Hlavní funkce 28
6.1.2 Creator 28
6.1.3 Vytvoření adresního prostoru 29
6.2 Struktura dat v REXu 29
6.2.1 Exekutiva 29
6.2.2 Bloky 29
6.2.3 Subsystém 29
6.2.4 Úloha (Task) 30
6.3 Princip synchronizace 30
6.3.1 Synchronizační vlákno 31
6.3.2 Metody Creatoru 32
6.4 Adresní prostor 32
6.5 Struktura dat serveru 33
6.5.1 Bloky 33
6.5.2 Parametry a vnořené bloky 34
6.5.3 Identifikátor pro diagnostický protokol 34
6.5.4 Proměnné 35
6.6 Propojení s diagnostickým systémem REXu 35
6.6.1 Prohledání prostoru 35
6.6.2 Zápis a čtení 36
6.6.3 Přehrání exekutivy 36

7 Příklady použití 37
7.1 Testování aplikace 37
7.2 Praktické použití 37
7.3 Budoucí vývoj 40
7.3.1 Zabezpečení 40
7.3.2 Pole 41
7.3.3 Historický přístup k datům 41

8 Závěr 42

Seznam použité literatury **43**

Seznam použitých zkratek **45**

1. Úvod

Tato práce popisuje technologii OPC UA (Unified Automation) a dokumentuje OPC UA server vytvořený pro řídicí systém REX. V textu může být namísto zkratky OPC UA použita pouze zkratka UA.

1.1 Motivace

OPC UA je specifikace pro sdílení dat mezi různými výpočetními jednotkami od různých výrobců. Byla navržena jako multiplatformní nástupce klasického OPC (viz kapitola 3). UA v sobě obsahuje i další rozšíření jako například vytváření a zachytávání událostí, historický přístup k datům, vzdálené spouštění funkcí a stavový automat. Na rozdíl od klasického OPC může klient v UA i přidávat další objekty, nejen upravovat již existující. Obecně UA reaguje na směr vývoje komunikace v posledních letech a nabízí modernější a obecnější způsob ukládání a předávání dat.

Díky svým vlastnostem a širokému využití klasického OPC je OPC UA velmi populární. Technologii OPC UA využívá většina velkých firem zabývajících se řízením a komunikací, například ABB Automation GmbH [1] nebo Siemens AG[2]. Další výhodou je možnost využití již vytvořených nástrojů vhodných k budování nové aplikace a použití veřejně dostupných klientů ke komunikaci se serverem.

Nejčastějším použitím OPC UA je získávání dat z řídicích jednotek určitého systému. Data z řídicích jednotek pak mohou být k dispozici ostatním řídicím jednotkám, jiným serverům nebo klientům. Lze vytvářet i složité vícevrstvé sítě.

Popularita OPC UA je hlavním důvodem pro vytvoření UA serveru pro REX. REX je řídicí systém, který poskytuje podporu klasického OPC. Implementace UA serveru v REXu umožní další manipulaci s běžícím systémem s pomocí jakéhokoliv UA klienta.

1.2 Postup řešení

Cílem této práce je nastudování technologie OPC UA a vytvoření OPC UA serveru pro řídicí systém REX.

Nejprve bylo zapotřebí nastudovat velmi podrobnou a rozsáhlou specifikaci OPC UA verze 1.01 [3], která byla aktuální v době zahájení práce a obsahovala 11 hlavních částí. Kromě specifikace samotné bylo vhodné prostudovat i dokument OPC UA/PLC open information model [4], který specifikuje použití OPC UA spolu s PLCopen. PLCopen je specifikace od stejnojmenné nezávislého sdružení, které zajišťuje efektivitu v průmyslové automatizaci. Vzhledem k rozsahu specifikace UA nebylo prakticky možné vytvořit celý UA server od začátku. Důležitým krokem pro návrh UA serveru bylo nalezení vhodných nástrojů, které by umožnily vytvoření UA serveru v kratší době. Po nalezení a zakoupení vhodných nástrojů se délka vývoje razantně zkrátila a UA server bylo možné vytvořit během několika týdnů.

Na závěr bylo třeba navrhnout strukturu dat a použití OPC UA v systému REX. Úkolem nebylo vytvořit naprosto dokonalý UA server, ale spíše funkční

prototyp, který bude možné dále rozvíjet podle dalších požadavků.

2. Řídicí systém REX

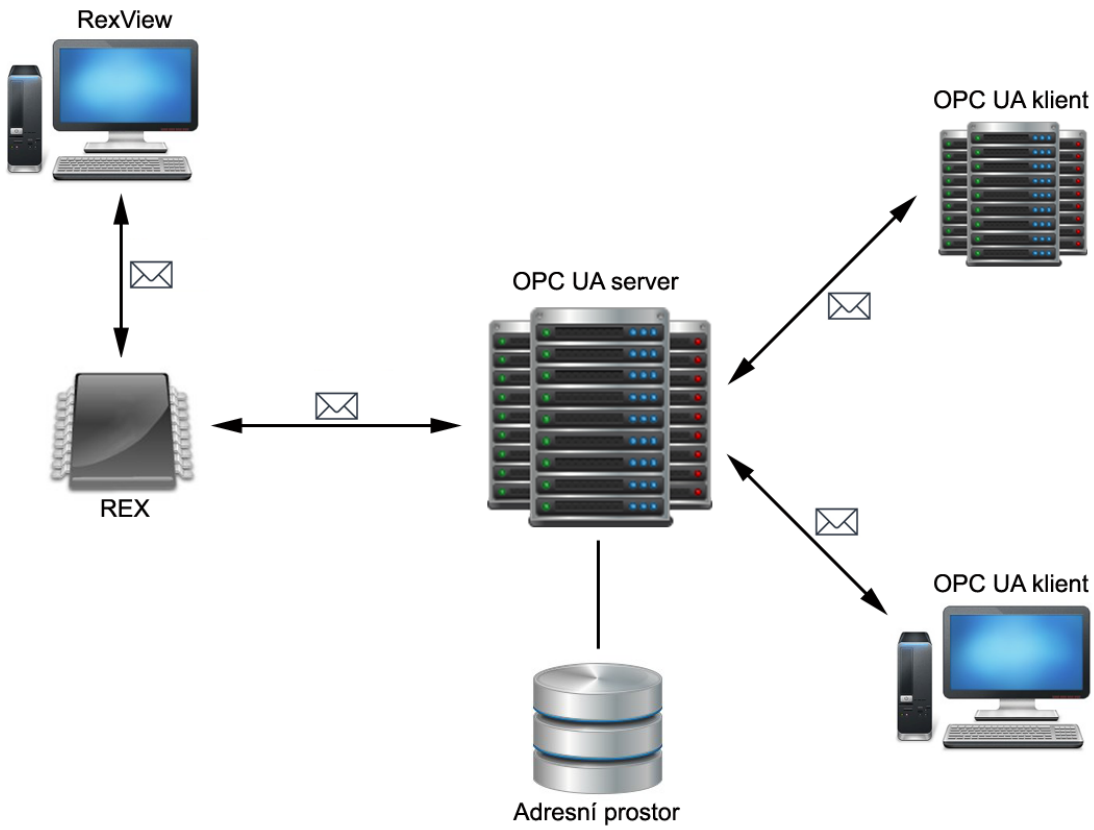
„Řídicí systém REX je vyspělý nástroj pro návrh a realizaci komplexních algoritmů automatického řízení. Knihovny funkčních bloků, z nichž je výsledný algoritmus sestaven, pokrývají nejen všechny běžné oblasti automatizace a regulace, ale nabízejí navíc i řadu prvků pro tvorbu pokročilých algoritmů řízení. Pro speciální aplikace je dále možné zařadit zcela vlastní funkce využitím volně programovatelného bloku. V případě, že je k dispozici licence systému Matlab®-Simulink®, je díky vzájemné kompatibilitě možné využít všechny jeho výhody pro simulaci a otestování řídicích algoritmů před jejich skutečným nasazením [5].“

Tento systém je možné provozovat na platformách Windows, Linux, Windows CE a FreeRTOS. REX podporuje komunikaci pomocí OPC Data Access 2.0 a 3.0. K vytváření schémat lze využít program RexDraw, k diagnóze, zobrazení a ovládání stavu běžící exekutivy se používá program RexView [5].

2.1 Diagnostický protokol

Řídicí systém REX podporuje vzdálené ovládání a diagnostiku serveru pomocí svého diagnostického protokolu napsaného v jazyce C++. Pomocí diagnostického protokolu je možné se připojit k běžící exekutivě REXu a poté s ní jednoduchými příkazy manipulovat a získávat data o jejím stavu. Tento protokol využívají další nástroje, které komunikují s exekutivou REXu.

OPC UA server pro REX bude pomocí diagnostického protokolu získávat data z REXu a ukládat je do svého adresního prostoru. Na data v adresním prostoru se poté budou dotazovat OPC UA klienti (viz obrázek 2.1).



Obrázek 2.1: RexView i OPC UA klient mají stejný přístup k datům v REXu. Pro OPC UA klienta je zprostředkovává OPC UA server, který si je ukládá do svého adresního prostoru.

3. Technologie OPC UA

Specifikace OPC UA je nástupcem původního OPC, nyní známého jako OPC Classic, z velké části se však od něj svými principy liší.

Původní význam zkratky OPC je OLE for Process Control, kde OLE znamená Object Linking and Embeding. Celý název lze přeložit jako Spojování a vkládání objektů pro řízení procesů. Nyní zkratka OPC znamená Open Platform Communication. Účelem specifikací OPC Classic a OPC UA je zprostředkovat předávání dat mezi různými aplikacemi v předem dané podobě a pomocí předem známých principů.

OPC UA specifikace nemá za úkol definovat vnitřní stavbu aplikace, pouze specifikovat její komunikaci a ukládání dat. Tato data jsou přístupná jiným systémům využívajícím OPC UA bez dalších znalostí. Každý klient a server musí tuto specifikaci splňovat, jinak je vlastní programování ponecháno na vývojářích, kteří musí použít prvků systému, pro který je aplikace programována.

První verze OPC Classic byla zveřejněna v roce 1996, od té doby se dočkala rozšíření v podobě specifikací obsahujících historický přístup k datům, alarmy a události. Samotný klasický přístup k datům prošel dalšími dvěma úpravami, které jsou zpětně kompatibilní ze strany serveru.

Tato technologie byla navržena pro využití řízení procesů a je často využívána v automatickém řízení, kde umožňuje předávání dat mezi kontroléry, které řídí systém, nebo mezi kontrolérem a klientem, který data zobrazoval a umožňoval s nimi operovat na dálku.

Výhodou využití OPC je jistota kompatibility komunikace s jinými aplikacemi používající OPC a úspora času a nákladů na návrh a vývoj nové společné komunikace mezi aplikacemi.

Hlavní nevýhodou OPC Classic je založení této myšlenky na technologii Microsoftu COM/DCOM, čímž se eliminovalo využití OPC na jiných platformách. To bylo hlavním důvodem pro vytvoření OPC UA. Dalšími důvody byly například specifikování zabezpečení, podpora komunikace pomocí SOAP nebo potřeba využití komplexních datových struktur.

První verze OPC UA byla vydána v roce 2006, od té doby vývoj dále pokračuje. Aktuální verze je 1.02.

3.1 Základní principy

Specifikace OPC UA je velmi rozsáhlá a podrobná. Její nejdůležitější myšlenky se dají rozdělit do několika částí.

První částí je fyzická komunikace a předávání dat mezi klientem a serverem. Do této části patří mapování, kde je specifikován způsob navazování spojení, zabezpečení komunikace a struktura poslaných dat daná použitým protokolem.

OPC UA je specifikována jako služebně orientovaná architektura (SOA – Service Oriented Architecture), což znamená, že jsou definované služby, na které se klient může dotazovat a server na každý dotaz reaguje příslušnou odpovědí. Druhou částí komunikace jsou služby, které server poskytuje. Služby vytváří abstraktní komunikační model. Po navázání fyzického spojení se pomocí služeb vytváří a udržuje zabezpečený kanál (SecuredChannel) a relace (Session). Zabez-

pečený kanál je nutné mít aktivní pro veškerou komunikaci, relaci je nutné mít vytvořenou pro dotazování klienta na služby serveru.

Obě tyto části komunikace jsou často dohromady označovány jako komunikační zásobník (Stack), který bývá programován jako samostatná knihovna použitelná dalšími aplikacemi při vytváření aplikací, které využívají OPC UA.

Třetí částí je popis adresního prostoru pomocí uzlů, popis využití uzlů k vytvoření informačního modelu a popis přístupu k datům včetně monitorování.

Ačkoliv lze tyto části považovat za základní myšlenky, nejde o nutné minimum potřebné k provozování a používání aplikací využívajících OPC UA.

3.2 Profily

Pro to, aby byla aplikace OPC UA funkční, není důležité implementovat veškeré specifikované funkce, stačí splnit nutné minimum pro provoz a další funkčnost přidávat podle potřeby. Aby se dalo zjistit, které funkce aplikace podporuje, existují profily se seznamem vlastností, které aplikace musí splňovat. Aplikace poskytuje informace o tom, které profily podporuje, a tím umožní dalším aplikacím zjistit, které části specifikace využívá. Profily mohou obsahovat sady služeb, použité kódování, zabezpečení a další volitelné části specifikace OPC UA.

3.3 Komunikace a zabezpečení

Komunikaci v OPC UA obstarává komunikační zásobník, který je rozdělen do tří vrstev – transportní, komunikační a aplikační. Zabezpečení je navrženo tak, aby odolalo moderním hrozbám napadení. V druhé části UA specifikace jsou popsány hrozby a postupy, kterými se lze proti těmto hrozbám bránit a které UA implementuje svým návrhem nebo mohou být do implementace zavedeny vývojáři. Zabezpečení je vnitřní a vnější. Vnějším zabezpečením může být například šifrování a podepisování zpráv nebo využití zabezpečeného kanálu. Mezi vnitřní patří zavírání přebytečných a neužitečných spojení nebo auditing (zaznamenávání aktivity na serveru).

3.3.1 Transportní vrstva

Transportní vrstva obstarává samotné odesílání zpráv. Vytváří a obsluhuje komunikační protokol. Používá šifrovací a ověřovací mechanismy k zajištění bezpečnosti zprávy poslané po síti, aby nemohla být zpráva přečtena ani modifikována třetí stranou. Při navazování spojení je transportní vrstva vytvořena ihned po připojení.

3.3.2 Komunikační vrstva

Komunikační vrstvu představuje zabezpečený kanál (SecuredChannel), který musí být vytvořen ihned po navázání komunikace. Způsob vytvoření kanálu záleží na použitém komunikačním protokolu.

Komunikační kanál zajišťuje bezpečnost z pohledu důvěrnosti a celistvosti komunikace, identifikuje aplikaci. Každá komunikace mezi serverem a klientem

probíhá na zabezpečeném kanále. Pro více připojení je třeba použít více kanálů, na každém kanále lze identifikovat, kterému partneru patří.

Po vytvoření zabezpečeného kanálu je vytvořen identifikátor kanálu (SecureChannelId) a bezpečnostní známka (SecurityToken), kterými se dále kanál prokazuje. Zatímco identifikátor je trvalý, známka má omezenou životnost a je třeba ji v pravidelných intervalech obměňovat. Po vypršení životnosti poslední známky se spojení ukončí.

Při poruše transportní vrstvy komunikační vrstva existuje dál, dokud nevyprší platnost známky. Pokud je do této doby obnovena transportní vrstva, může dále využívat původní zabezpečený kanál.

3.3.3 Aplikační vrstva

Aplikační vrstvu představuje relace (Session), ve které probíhá veškeré volání a zpracování služeb. Relace se používá k vnitřní identifikaci komunikace a může poskytovat i autorizaci.

Při vytváření relace předá klient serveru přihlašovací údaje a server po ověření údajů označí klienta jako specifického uživatele, který může mít práva na provedení některých akcí. Specifikace OPC UA nenavrhuje, v jaké formě mají existovat uživatelé, navrhuje pouze jak předat přihlašovací údaje. Každý zabezpečený kanál může mít nejvýše jednu relaci.

Relaci je třeba vytvořit a poté ještě aktivovat. Aktivací se relace sváže s aktivním zabezpečeným kanálem. Relaci je možné aktivovat v jiném kanále, tím se s novým kanálem sváže. Je to ale možné pouze pokud jde o stejného klienta. Relace takto není ovlivněna při poruše zabezpečeného kanálu, stačí vytvořit nový kanál a relaci znovu aktivovat. V relaci po aktivaci jiným kanálem zůstanou všechny údaje a nastavení.

Relace se uzavírá po předem určené době nečinnosti. Pokud klient neposílá žádné požadavky, musí pravidelně poslat dotaz, kterým relaci udrží aktivní.

3.4 Adresní prostor a uzly

Adresní prostor (AddressSpace) tvoří sdílená data. Adresní prostor se vytváří z jednoduchých objektů, uzlů (Node), které mezi sebou mají různé vazby. Uzly jsou základní jednotkou sdílených dat.

Vazby mezi uzly se nazývají reference. Pomocí různých referencí se vytvářejí složitější objekty. Mezi reference patří například *hasComponent*, která další uzel označuje jako komponentu zdrojového uzlu, *hasType*, která přiřazuje každému uzlu jeho typ, *hasVariable*, která přiřazuje cílový uzel jako proměnou zdrojového uzlu, a mnohé další.

Každý uzel má vlastní identifikátor (NodeId), který se skládá s z jmenného prostoru (Namespace) a unikátní části, která může být číselná, textová nebo může být využito GUID (Global Unique Identifier). Jmenný prostor je URI, který označuje kontext uzlu. Pokud je na jiném serveru použito stejného identifikátoru, včetně jmenného prostoru, jedná se o tentýž uzel.

Dalšími povinnými parametry uzlu jsou zobrazované jméno (DisplayedName), název uzlu, který nemá žádný další význam, a prohledávané jméno (BrowseName). S pomocí prohledávaného jména lze hledat uzly z výchozího uzlu pomocí re-

ferencí. Prohledávané jméno nemusí být unikátní, může vyjadřovat vlastnost a umístění uzlu.

UA definuje vlastní jmenný prostor *http://opcfoundation.org/UA/*, který je všeobecně známý a obsahuje předdefinované uzly potřebné pro provoz serveru. Mezi těmito uzly jsou například typy, reference, mapování a dotazy i odpovědi služeb. V adresním prostoru jsou tedy uloženy i nástroje potřebné k definování adresního prostoru.

3.4.1 Atributy a třídy uzlů

Atributy jsou parametry konkrétního uzlu, které jsou jiné pro každou třídu uzlů. Mezi atributy patří například identifikátor uzlu, prohledávané jméno, zobrazovací jméno, právo k zápisu, hodnota nebo třída uzlu.

Uzly se organizují do několika základních tříd, které definují jejich použité atributy. Dělí se na proměnné (Variable), typy proměnných (VariableType), objekty (Object), typy objektů (ObjectType), typy referencí (ReferenceType), typy dat (DataType), metody (Method) a pohledy (View).

- **Proměnné** v sobě mohou obsahovat data nebo mohou vytvářet komplexnější úložné struktury tím, že obsahují další uzly jako své vlastní proměnné. Proměnné by měly svou strukturou odpovídat svému **typu**.
- **Objekty** slouží k organizování svých komponent a proměnných. Objekt by měl odpovídat svou konstrukcí svému **typu**.
- **Typy referencí** se používají k vyjádření vztahů mezi různými uzly. Každý uzel musí být připojen alespoň jednou referencí k jinému, již připojenému uzlu adresního prostoru, jinak se v adresním prostoru nenachází.
- **Typy dat** definují u proměnných datový typ uložené hodnoty. Proměnné nemají určený datový typ své hodnoty, ten může být jakýkoliv. Informace o datovém typu je v proměnné uložena jako atribut.
- **Metody** jsou uzly, které umožňují klientovi spustit metodu serveru s určitými argumenty. Výsledek metody se po zpracování vrací zpět klientovi.
- **Pohledy** určují, které z uzlů má klient možnost zobrazit. Je možné vytvořit několik pohledů, jimiž lze poté různě filtrovat data zobrazená klientovi.

Objekty a proměnné by měly konstrukcí odpovídat svému typu, z pohledu objektového programování by objekty a proměnné měly být instancí svých typů. V rámci definice typu je možné nastavit, které dceřiné uzly jsou povinné. Pokud je typ složený z více uzlů, daný objekt nebo proměnná musí obsahovat uzly, které jsou označeny jako povinné (mandatory).

3.4.2 Hodnoty

Hodnoty proměnných jsou komplexní struktury, které obsahují data, datový typ, čas serveru, čas zdroje a stavový kód, který obsahuje informaci o tom, zda server může danou hodnotu poskytnout.

Čas serveru je čas, kdy byla hodnota zapsána do uzlu, čas zdroje může označovat pořízení hodnoty externím zařízením. Čas serveru může být totožný s časem zdroje. Při dotazování na data může klient požadovat data s maximálním stářím, pokud jsou data starší, server vrátí v odpovědi chybu.

Hodnoty mohou být skaláry i pole s více dimenzemi a jejich hodnotou mohou být různé typy, od jednoduchých čísel a řetězců až po složitější struktury.

3.5 Služby

Komunikace mezi klientem a serverem probíhá výhradně pomocí volání a zpracovávání služeb. Tyto služby se podle své činnosti organizují do služebních sad (ServiceSet), které se zabývají ovládáním jednotlivých části funkcí serveru OPC UA.

Jednotlivé konstrukce dotazů a odpovědí jsou definovány ve čtvrté části specifikace OPC UA. Dotazy i odpovědi mají své společné hlavičky. Klient má například u všech dotazů možnost nastavit požadované informace, které má server vrátit. Server ve svých odpovědích nastavuje stavový kód vykonání požadavku, který oznamuje, zda bylo vykonání služby úspěšné. Některá volání služby mohou najedou vykonat více akcí, pak se kromě stavového kódu celé služby vracejí i částečné stavové kódy jednotlivých akcí.

Všechny stavové kódy definuje OPC UA. Dělí se na dobré (Good), které značí správně provedenou službu, nejisté (Uncertain) a špatné (Bad), které značí selhání volání služby. Každá služba má definováno, které stavové kódy může vracet. Chyby se například mohou týkat přístupových práv, stavu serveru, stavu cíle, špatně provedené akce. Nejistý výsledek může nastat, pokud alespoň jedna z částečných akcí skončila chybou.

Kromě služeb týkajících se relací a zabezpečeného kanálu lze většinu služeb, namísto opakovaných dotazů, používat hromadně.

3.5.1 Průzkumné služby (Discovery)

Průzkumná sada služeb umožňuje zjišťovat údaje o samotném serveru a možnostech jeho připojení. Připojit se lze pouze k přípojnému bodu (Endpoint), který je otevřen ke komunikaci. Přípojné body mohou a nemusí být předem známy, pomocí této sady je možné najít seznam přípojných bodů.

První možností získání informací o serveru je připojení k veřejnému přípojnému bodu serveru, který podporuje průzkumné služby. Následně klient požádá o seznam jeho dalších přípojných bodů. Poté se může díky získaným informacím připojit na jiný přípojný bod a využívat ostatních funkcí serveru.

Druhou možností je vytvoření samostatného serveru s dobře známou adresou přípojného bodu. Tento server poté skladuje adresy přípojných bodů všech ostatních serverů.

Pokud server poskytuje přípojný bod pouze pro průzkumné služby, bývá tento přípojný bod nezabezpečen, spolu se seznamem přípojných bodů pak klient u každého získá i informace o zabezpečení, které je k připojení k danému bodu nezbytné.

3.5.2 Zabezpečený kanál (SecuredChannel)

Sada služeb zabezpečeného kanálu umožňuje kanál vytvořit a ukončit. Při vytváření je nutné použít certifikát aplikace. Stejný certifikát je poté nutné použít i při vytváření relace.

Při otevírání kanálu klient zároveň nastaví používané zabezpečení, server při každém volání vrací novou bezpečnostní známku. Pro vytvoření nové známky je tedy třeba opětovně volat službu na otevření zabezpečeného kanálu.

3.5.3 Relace (Session)

Sada služeb relací umožňuje vytvořit a zrušit relaci, aktivovat existující relaci a zrušit všechny probíhající akce. Relace není po vytvoření aktivní, klient ji před voláním dalších služeb musí aktivovat, aby se spojila s jeho aktuálním zabezpečeným kanálem. Klient může zrušit všechny probíhající služby, které zavolal v aktuální relaci, všechny odpovědi budou obsahovat chybový kód označující přerušení akce. Tato funkce může být využita k uvolnění zdrojů nebo přerušení příliš dlouho trvajících akcí.

3.5.4 Správa uzlů (NodeManagement)

Tato sada služeb umožňuje klientovi vytvářet a mazat uzly a vytvářet a odstraňovat reference mezi různými uzly. Na rozdíl od specifikace OPC Classic může OPC UA server dovolit klientovi vytváření a mazání uzlů v adresním prostoru a správy referencí mezi nimi. Klient tak může na serveru vytvářet vlastní objekty.

3.5.5 Procházení (Browse)

Pomocí služeb procházení (Browse) a pokračování procházení (BrowseNext) lze procházet a načítat adresní prostor. Procházen může být určitý pohled (View). Lze nastavit, v kterém směru budou které reference prohledávány a která data má služba vrátit.

3.5.6 Pohled (View)

Sada služeb týkajících se pohledů obsahuje služby na postupné procházení adresního prostoru. Klient může zobrazit reference daného uzlu na další uzly, tím lze postupně projít celou strukturu adresního prostoru nebo daného pohledu.

Další službou této sady je služba, která pomocí relativní cesty a počátečního uzlu najde požadovaný unikátní identifikátor uzlu. Relativní cesta je složena z procházejících jmen a referencí.

Poslední dvě služby umožňují zaregistrování a zrušení registrace uzlu. Při zaregistrování klient označí uzel, který bude často používat, server má poté možnost co nejvíce zefektivnit přístup k tomuto uzlu.

3.5.7 Dotazování (Query)

Pomocí těchto služeb se klient může dotazovat na data, která splňují parametry specifikované v dotazu. Při dotazování nad pohledem server vrací pouze data

obsažená v daném pohledu.

Dotazy mohou být konstruovány od velmi jednoduchých až po velice sofistikované. Lze nastavit jak požadované parametry uzlu, tak i složité podmínky pomocí operandů. Při příliš velkém počtu výsledků se vrátí pouze část, další části se získávají voláním jednodušší služby, které předávají bod, od kterého má seznam pokračovat.

3.5.8 Atributy (Attribute)

Tyto služby umožňují číst a zapisovat do atributů uzlu a číst jeho historické hodnoty. Při čtení hodnoty lze navíc mimo jiné zvolit, jak stará tato hodnota může být, aby byla považována za přijatelnou.

3.5.9 Metody (Method)

Tato sada obsahuje pouze jednu službu, a to službu, která danou metodu zavolá, předá jí argumenty a vrátí zpět hodnoty vstupních i výstupních parametrů. Metodu v adresním prostoru představuje specializovaný uzel.

3.5.10 Monitorování (Monitor)

Sada služeb určená k monitorování umožňuje vytvoření, úpravu a mazání monitorovaných položek (MonitoredItem), nastavení monitorování a vytváření spouštěčů událostí. Monitorované položky periodicky snímají hodnotu monitorovaného uzlu a předávají je do fronty odběru (Subscription). Stejně tak se do fronty odběru přidávají spuštěné události.

3.5.11 Odběr (Subscription)

Odběry jsou vytvářeny, upravovány a mazány pomocí této sady služeb. Jejich účelem je předávat klientovi informace o změnách stavu monitorovaného uzlu.

Odběr obsahuje frontu, do které jsou přidávána data. Při zavolání publikující služby se obsah fronty odešle klientovi. Mezi publikováním musí uběhnout předem určené intervaly, při vícenásobném zavolání publikující služby se služba vykoná až po uplynutí požadovaného intervalu.

3.6 Informační model

Pátá část specifikace [3] obsahuje popis informačního modelu OPC UA. Jsou zde popsány atributy patřící k jednotlivým třídám uzlů, možnosti datových typů, výpis standardních objektů, proměnných, referencí a datových typů. V této části specifikace lze zjistit nástroje určené k budování vlastního adresního prostoru.

Datový typ obsahuje údaj o tom, zda je jeho hodnotou skalár, pole nebo obojí. Přesně definované pole obsahuje údaj o své dimenzi. Pole vždy obsahuje jen hodnoty jednoho typu. Datové typy jsou čísla, řetězec a řetězec bytů, GUID, struktury, obrázky, identifikátor uzlu, čas, XML element, lokalizovaný název, výčtový typ (Enumerator) a další.

UA definuje základní uzel objektů (ObjectType), od kterého by měly dědit všechny ostatní typy objektů. Pro server jsou definovány speciální uzly, které umožňují předávat údaje o serveru samotném, například typ Server nebo různé typy uzlů pro události serveru.

Proměnné se v základu dělí na klasické proměnné a vlastnosti. Vlastnosti jsou speciálním, dále nerozšiřitelným typem. Zatímco atributy uzlu jsou společné pro všechny uzly, proměnné jsou uzly, které specifikují vlastnosti rodičovského uzlu. Příkladem může být rozsah nebo údaje o inženýrské jednotce příslušné k rodičovské proměnné.

Reference umožňují vyjadřovat vztahy mezi jednotlivými uzly. Vycházejí ze základní reference, dělí se, dále specifikují svůj význam a vytváří tak stromovou strukturu. Hlavní rozdělení referencí je mezi hierarchické a nehierarchické, kde nehierarchické definují typ uzlu (*hasTypeDefinition*) a generují události (*GeneratesEvent*), zatímco hierarchické organizují (*Organizes*) nebo agregují další uzly. Agregační reference označují dceřiný uzel za komponentu uzlu (*hasComponent*), proměnnou uzlu (*hasVariable*) nebo podtyp jiného typu (*hasSubtype*).

Kromě typů objektů UA definuje i objekty. Základním uzlem adresního prostoru je kořenový uzel (Root), ke kterému ovšem nelze přímo připojit další uzly. K připojení dalších uzlů slouží jiné předdefinované uzly, které jsou přímo nebo nepřímo připojeny ke kořenovému uzlu. Připojením nových uzlů je vytvářen adresní prostor. Přípojnými uzly jsou uzel Objects, ke kterému lze připojit všechny objekty, uzel Views, ke kterému lze přidávat další pohledy, a uzly VariableTypes, ObjectTypes, ReferenceTypes, EventTypes a DataTypes, které lze rozšiřovat pouze vytvářením podtypů.

3.7 Mapování

Část specifikace zabývající se mapováním obsahuje informace o správném použití komunikačního protokolu a konverzi dat serveru na data přenášená po síti a zpět. OPC UA specifikuje předávání dat v předem známé podobě mezi různými platformami, které stejná data interpretují jiným způsobem, například čas na různých platformách, paměť pro celočíselný typ v 32 a 64 bitových systémech nebo pořadí bytů.

3.7.1 Převod a zabezpečení dat

Komunikaci v aplikaci OPC UA obstarává komunikační zásobník, který lze podle přístupu práce s daty rozdělit na vrstvu serializace, vrstvu zabezpečeného kanálu a transportní vrstvu.

Data odesílaná aplikací jsou ve vrstvě serializace převedena z objektů na pole bytů nebo XML podle použitého kódování (binární, XML).

UA specifikuje také způsob navázání komunikace, kde musí proběhnout bezpečnostní ověření (Security Handshake). Při spojení musí být poslána žádost na otevření zabezpečeného kanálu, která je šifrována a podepisována asymetrickými metodami, a přijata odpověď, která je zabezpečena stejným způsobem. Po přijetí zpracované odpovědi klient a server začnou používat k podpisu a šifrování symetrické metody, na kterých se shodli při otevírání zabezpečeného kanálu.

UA specifikuje strukturu certifikátu, který si aplikace vyměňují i postupy použité ke splnění nutného zabezpečení a bezpečnostního ověření pro oba komunikační protokoly.

3.7.2 Binární kódovaní

Binární mapování je navrženo k minimalizování použité paměti nutné k přenosu dat. UA specifikuje velmi podrobně počet a význam bytů pro každý základní datový typ a způsob přenášení složitějších struktur a objektů. Díky přesnému popisu pořadí bytů mizí problém s přenosem dat mezi platformami, které stejný datový typ zpracovávají různými způsoby, obě platformy musí používat správný převod mezi binárním mapováním a svými datovými typy.

Předem známé složité struktury, například zprávy mezi aplikacemi, se převádějí do pole bytů tím, že je na jejich začátku identifikátor objektu následovaný obsahem objektu pomocí seřazeného obsahu jeho parametrů. Při skládání objektu se objekt naplňuje daty do daných parametrů. Tím, že je přesně známa posloupnost parametrů pro většinu objektů, není třeba dalších metadat, která by zvětšovala délku odeslané zprávy. Další metadata se ukládají pouze na místech, kde označují počet prvků, délku řetězce, typ následujícího objektu nebo značí, které části objektu jsou přítomny. Pokud objekt obsahuje další objekt o předem známé délce, není pro vnořený objekt zapotřebí dalších metadat. Pokud je počet vnořených objektů proměnlivý, před objekty je přidáno číslo s jejich počtem. Skládání a rozkládání těchto objektů probíhá bez dalších metadat.

Zajímavé struktury jsou například řetězec, řetězec bytů a XML, kde se před vlastním obsahem vyskytuje číslo značící počet bytů využívaných zprávou, identifikátor uzlu, který ve své hlavičce obsahuje přepínač zobrazující, které části jsou ve zprávě přítomny, rozšiřitelný objekt určený k identifikování přenášených objektů, variant použitý k přenášení hodnot nepředdefinovaného typu nebo čas vyjádřený počtem desetin milisekund od 1. 1. 1601.

3.7.3 XML kódování

XML kódování určuje převod mezi daty aplikace OPC UA a XML, které se posílá mezi klientem a serverem. UA definuje struktury použité k vyjádření datových typů i složitých objektů. Toto kódování může být pro některé aplikace jednodušší ke zpracování, používá se hlavně při použití komunikačního protokolu SOAP.

3.7.4 OPC TCP

OPC TCP je protokol vhodný pro udržování spojení pomocí TCP, ke kterému specifikuje způsob připojení, strukturu a zabezpečení odesílaných zpráv. Na rozdíl od SOAP je možné přijímat odpovědi serveru v jakémkoliv pořadí a v případě chyby komunikace i na jiném segmentu (socket).

TCP dělí zprávy na balíčky, které postupně posílá posluchači po síti. OPC TCP dělí ještě před odesláním zprávu na menší kousky (chunk), aby je mohla přijímající aplikace zpracovat. První částí kousku je vždy hlavička zprávy, která daný kousek identifikuje a zároveň je díky němu možné poznat, že zpráva patří k určitému zabezpečenému kanálu. Podle nastaveného zabezpečení se data zašifrují,

podepíšou a pro obě akce se přidají odpovídající hlavičky. Při přijetí je kousek identifikován, zkontrolován, dešifrován a z obsahu se poté skládá celá zpráva pomocí určeného kódování.

OPC TCP při navázání komunikace ještě před otevřením zabezpečeného kanálu odešle a přijme zprávy, které jsou nutné k udržení komunikace. Třetí zprávou je žádost o otevření zabezpečeného kanálu, která bez tohoto kanálu nemůže být odeslána klasickým způsobem.

UA definuje i postup při chybě připojení, který je téměř totožný s postupem k navázání spojení. Jediným rozdílem je odeslání identifikátoru komunikace ve zprávě otevírající zabezpečený kanál.

3.7.5 SOAP (Simple Object Access Protocol)

UA při použití SOAP definuje URL, pomocí níž je možné se dotazovat na určité služby nebo posílat binární data. Klient odesílá data v POST části dotazu, server vrací data pomocí HTTP odpovědi. UA nedefinuje žádné přídavné hlavičky, ale hlavičky mohou být přidány při využití zabezpečení zpráv.

3.8 Rozšířené proměnné

Tato část specifikace přidává další vestavěné typy proměnných. Tyto typy jsou navržené tak, aby lépe odpovídaly potřebě zobrazování dat z fyzického systému za pomoci UA serveru.

Analogová proměnná (AnalogItem) je komplexní proměnná obsahující kromě své hodnoty rozsah měřícího nástroje, rozsah měřené hodnoty a inženýrskou jednotku (EngeneeringUnit), která doplňuje například fyzikální jednotky hodnoty proměnné.

Diskrétní proměnné jsou dvoustavové a vícestavové. Dvoustavová proměnná obsahuje dva stavy zobrazené ve formě textu, mezi kterými může přepínat, a odpovídá tak Booleovské logické proměnné. Vícestavová proměnná obsahuje více stavů, z nichž právě jeden je vždy aktivní.

Tyto proměnné by měly samy pravidelně snímat reálné hodnoty fyzického systému, aktualizovat své hodnoty a poskytovat údaje o intervalu snímání dat.

3.9 Programy

Programy jsou složité funkce týkajících se nějakého procesu. Server umožňuje klientovi získávat informace o stavu programu, změnách, výsledných datech a také program ovlivňovat voláním funkcí. Příkladem je automat s konečným počtem stavů, kde program mění stavy kvůli svým vnitřním pochodům nebo při volání funkcí klientem.

3.10 Historický přístup k datům

Tato část specifikuje způsob vytváření a interpretace historických dat. Jde o rozšíření původního historického čtení o nové reference, atributy, události a oddělení historických dat do speciálních uzlů, které jsou závislé na monitorovaných

uzlech. Tím jsou poskytovány sofistikovanější nástroje ke sběru a publikování různých historických dat.

4. Nástroje pro vývoj OPC UA serverů

Hlavním úkolem před vlastním vývojem OPC UA serveru bylo najít nástroje, které by tento vývoj mohly usnadnit. S použitím správných nástrojů by se vývoj aplikace mohl zkrátit z několika let na několik týdnů.

Systém REX je psán v jazyce C++. Výhodou aplikací napsaných v jazyce C/C++ je jejich rychlost a malá spotřeba paměti. Bylo rozhodnuto, že OPC UA server pro program REX bude napsán v jazyce C/C++. Bylo tedy nutné najít nástroje, které umožňují vytvoření serveru právě v tomto jazyce.

4.1 Nástroje k vytvoření serveru v C++

Při hledání a srovnávání různých nástrojů byl použit oficiální katalog [6] otestovaných serverů veřejně dostupný na stránkách OPC Foundation, katalog certifikovaných produktů [7] tamtéž. Dále se jako užitečné ukázalo srovnání různých SDK (Software Development Kit) pro vývoj OPC UA serveru na stránkách opcconnect.com [8]. Pro studium různých nástrojů poté bylo použito oficiálních stránek produktu.

4.1.1 UA Communication Stack

Tato knihovna je distribuována přímo OPC Foundation [9], bohužel k ní mají přístup jen korporátní členové. Přímo tuto knihovnu použít nemůžeme, nicméně jí využívají další knihovny a nástroje určené k jednoduché implementaci OPC UA.

Tato knihovna je napsána v ANSI C, je navržena tak, aby měla malou spotřebu paměti, byla bezpečná a plně implementovala komunikační vrstvu OPC UA.

4.1.2 Softing – OPC UA C++ Development Toolkits

Tento nástroj je nabízen firmou Softing Industrial [10]. Jedná se o nástroj k jednoduchému vytváření OPC UA serverů a klientů.

Toolkit je možné použít v operačních systémech Windows, Linux a WxWorks, pokrývá tím všechny potřebné platformy.

K volnému použití je dána demo verze toho nástroje, která se po uplynutí jedné hodiny od spuštění zastaví. Pro použití plné verze nástroje je třeba zakoupit klíče, které se vloží na patřičné místo do kódu. Je třeba zvlášť koupit klíč pro klienta a pro server. Klíče jsou navíc závislé na platformě, je tedy třeba koupit klíč pro každou kombinaci klient/server – platforma.

Součástí toolkitu je i dokumentace s tutoriálem, kde je popsáno, jak vytvořit jednoduché servery, ale i jak přepsat některé funkce serveru, aby lépe odpovídaly požadavkům vývoje. V demo verzi jsou obsaženy i příklady použití, komplexní testy, simulační nástroje a jednoduchý certifikát vhodný k okamžitému vývoji.

S tímto nástrojem se dá velmi rychle vytvořit jednoduchý UA server, ale lze i jednoduše modifikovat některé funkce serveru pouhým zděděním třídy Creator a přepsání požadovaných funkcí k vlastní potřebě.

Tutoriál je přehledný a rozdělený do několika kroků, kde jsou všechny potřebné informace ukázány nebo je nastíněno doporučené řešení problému.

Toolkit se skládá z nepřeloženého zdrojového kódu, který je překládán spolu s aplikací. Další částí jsou již přeložené knihovny obsahující další nástroje potřebné k běhu OPC UA, například výše zmíněná knihovna UA Communication Stack.

Toolkit zajišťuje širokou škálu funkcí včetně rozšířené bezpečnosti a profilů přístupu k datům, historického přístupu k datům, komplexních dat a událostí. Tento toolkit je certifikovaný sdružením OPC Foundation.

4.1.3 Unified Automation – ANSI C based OPC UA Server SDK

Tento SDK je vyvíjen společností Unified Automation [11]. Je napsán v jazyce ANSI C, což pro použití v C++ aplikaci nepředstavuje problém. Je distribuován v podobě zdrojového kódu, který je poté nutno přeložit spolu s aplikací. Součástí jsou i hotové příklady.

ANSI C SDK implementuje UA Stack jinak, než OPC Foundation, odděluje vrstvu nezávislou na platformě a vrstvu s kódem závislým na platformě. Implementuje společné funkce potřebné ve všech UA aplikacích, zabezpečení a pomocné funkce.

Tento SDK se prodává ve dvou verzích, základní a profesionální. Základní verze obsahuje profily podporující vestavěný UA server, běžný UA server a základní přístup k datům. Profesionální verze podporuje navíc profily komplexních typů, metody, monitorování a události, alarmy a prostředí (Alarms and Conditions) a historický přístup. ANSI C SDK je certifikován sdružením OPC Foundation.

Ačkoliv by pro tento projekt a vývoj prototypu mohla stačit základní verze SDK, pro další vývoj by bylo třeba zakoupit profesionální verzi, která poskytuje žádané rozšíření.

4.1.4 Unified Automation – C++ based OPC UA Server SDK

Tato C++ knihovna je také vyvíjena společností Unified Automation [12] a umožňuje psaní přenositelných UA serverů. Součástí jsou i hotové příklady pro Windows i Linux.

C++ SDK se skládá ze dvou SDK, jednoho pro server a druhého pro klienta, pro oba je však společná UA Base knihovna, která zajišťuje obalení ANSI C datových typů tak, jak jsou definovány ve specifikaci UA v části týkající se komunikace.

SDK zjednodušuje používání UA komunikace, implementuje společnou funkčnost UA serverů, zabezpečení a pomocné funkce. SDK se snaží zjednodušit vývoj OPC UA implementací společných funkcí OPC UA jako referencí, poskytováním základních a pomocných tříd pro často používané komponenty. Poskytuje obalové třídy pro veškerou funkcionalitu všech OPC UA struktur.

Nástroj je opět rozdělen do dvou verzí. Základní verze podporuje standardní UA server a práci s daty. Profesionální verze oproti základní navíc podporuje

profily správy uzlů, redundance klientů a viditelných serverů, komplexních datových typů, událostí, metod, alarmů a prostředí (Alarms and Conditions), kontroly pomocí auditů a historický přístup k datům. SDK je k dispozici ve formě zdrojových kódů nebo přeložených knihoven, pro každou verzi existuje jiná licence. C++ SDK je certifikován sdružením OPC Foundation.

Opět by pro tento projekt bylo lépe využít profesionální verzi, která by umožnila další vývoj UA serveru v budoucnosti.

4.1.5 MatrikonOPC – OPC UA Embedded Server SDK

Tento SDK je vyvíjen firmou MatrikonOPC [13]. Jedná se o nástroj, který umožňuje rychle přidat UA server do existující aplikace. Nejedná se o knihovnu, která by šla použít při programování C++ aplikace, ale o server, který se snadno nakonfiguruje.

SDK je zaměřen na vývoj nenáročných OPC UA serverů, které lze poté umístit i na mikrokontroléry s malou operační pamětí. Sám zajišťuje bezpečnost serveru i omezením počtu připojení a monitorovných položek. SDK je naprogramován v jazyce ANSI C. Tento výrobek není certifikován sdružením OPC Foundation.

4.2 Výběr nástroje

SDK společnosti MatrikonOPC není určen k programování vlastního OPC UA serveru, ale k rychlému zprovoznění UA serveru na již běžící aplikaci. A ačkoliv je velmi modifikovatelný, nevyhovuje potřebě programování serveru v C++.

UA Communication Stack je přístupný pouze korporátním členům sdružení OPC Foundation, poskytuje API implementující UA specifikaci komunikace. Další vývoj by byl při použití tohoto nástroje snazší, nicméně se UA Communication Stack nevyrovná dalším nástrojům, které zajišťují jednoduchou implementaci celého serveru. Nutností by bylo zaplacení členských poplatků.

ANSI C SDK společnosti Unified Automation by bylo možné využít, ale lepší variantou jsou zbývající dva nástroje psané přímo v jazyce C++. Bylo by možné implementovat ANSI C kód v C++ aplikaci, ale pokud existují knihovny pro jazyk C++, dá se od nich očekávat objektový přístup, který může použití nástroje velmi usnadnit.

C++ SDK společnosti Unified Automation i C++ Development Toolkit společnosti Softing vyhovují požadavkům na nástroj použitý k vývoji OPC UA serveru. Oba nástroje jsou certifikované společností OPC Foundation, implementují požadované funkce a umožňují snadné a rychlé vytvoření přenosného OPC UA serveru, nezávislého na platformě, na které běží.

Oba nástroje obsahují přehlednou dokumentaci a připravené příklady. Toolkit od firmy Softing má navíc velmi přehlednou dokumentaci, testovací nástroje a užitečný tutoriál.

Tyto dva poslední nástroje byly navrženy k použití.

4.2.1 Zvolený nástroj

Ve výsledku byl vybrán OPC UA C++ Development Toolkit od firmy Softing z důvodu dřívějších dobrých zkušeností s touto firmou.

K vývoji byla použita demoverze API, kterou lze na plnohodnotnou verzi povýšit pouhým zadání klíčů. Demoverze obsahuje nepřeložený kód, knihovny, velké množství příkladů s řešením nejčastějších úkolů, tutoriál pro výuku práce s Toolkitem, přehlednou dokumentaci, nástroje na vytvoření certifikátů potřebných pro komunikaci.

Od firmy Softing byly zakoupeny všechny klíče a zdrojový kód k Toolkitu pro vývoj OPC UA serveru.

4.3 Doplňující nástroje

Kromě základních nástrojů využívaných přímo k programování UA serverů byly použity i další nástroje, které k vývoji OPC UA serveru přispěly nepřímo. Některé nástroje byly použity ke studiu chodu OPC UA klienta a serveru, jiné nástroje byly použity k testování vyvíjeného serveru.

4.3.1 UaExpert

Tento nástroj je nabízen firmou Unified Automation [14] volně ke stažení. Jedná se o velmi sofistikovaného klienta s příjemným uživatelským prostředím.

UaExpert zobrazuje strukturu dat v přehledné stromové struktuře, manipuluje s daty, monitoruje uzly, zobrazuje historický trend dat, alarmy a prostředí (Alarms and Conditions) a umožňuje sledovat výkon určitých částí UA serveru (viz obrázek 4.1).

UaExpert je nejen plně funkční UA klient, ale splňuje i další kvality nástroje na testování UA serveru. Aplikace UaExpert je použitelná na operačním systému Windows i Linux.

4.3.2 UaDemoServer

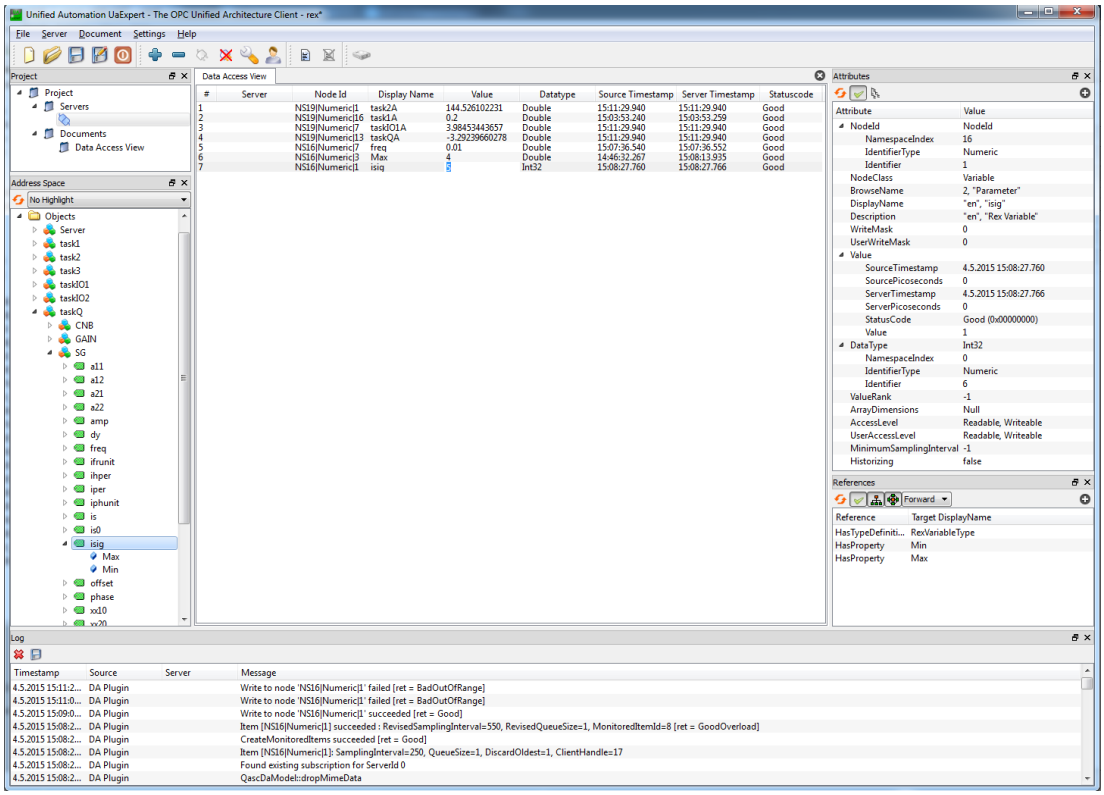
UaDemoServer je demonstrační UA server od firmy Unified Automation [15] nabízený volně ke stažení. Tento server byl použit k učení práce s UA serverem a ke zkoumání komunikace s klientem.

4.3.3 Softing OPC Client

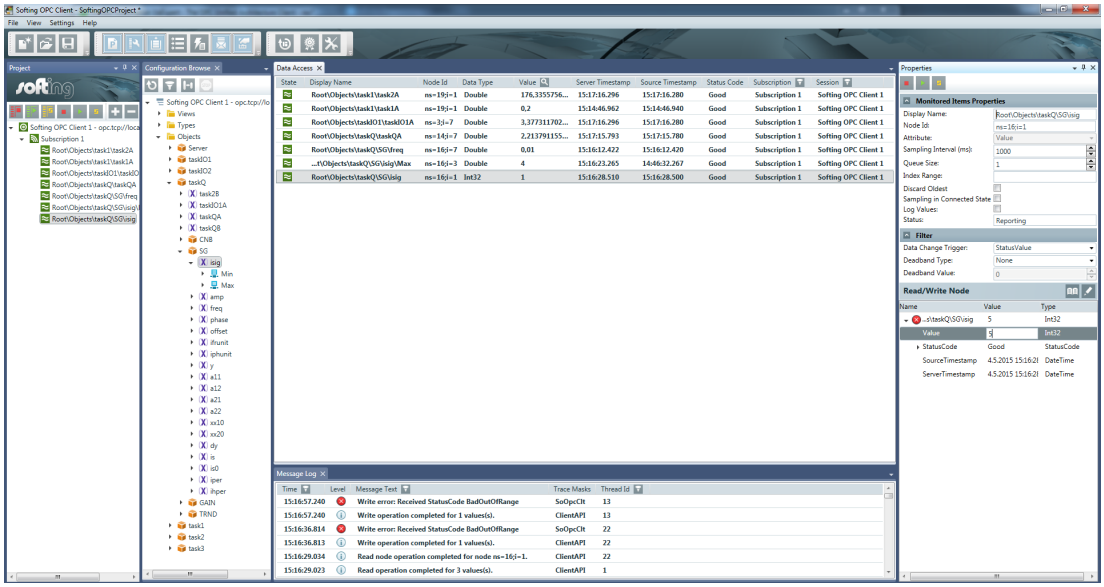
Tento OPC UA klient je vytvořen firmou Softing [16] a je dán volně ke stažení.

Softing OPC Client má podobné grafické prostředí jako UaExpert a stejně manipuluje s daty, ale neobsahuje další funkce, které jsou v UaExpertu navíc. I tak jde ale o klienta, kterého lze jednoduše používat k testování serveru (viz obrázek 4.2).

Druhý klient od jiné firmy je použit kvůli možnosti připojení více klientů zároveň a také kvůli možnosti jiného chování. Cílem je, aby server fungoval na všech certifikovaných klientech, testováním serveru jiným klientem se zmenšuje šance opomenutí rizik, přehlédnutí chyb a spoléhání na vlastnosti aplikace UaExpert.



Obrázek 4.1: Program UaExpert



Obrázek 4.2: Softing OPC Client

```
Filter: opcua                                   Expression...  Clear  Apply  Save
No.  Time        Source      Destination   Protocol  Length  Info
 48  40.519318   127.0.0.1   127.0.0.1     OpcUa      176    OpenSecureChannel message: OpenSecureChannelResponse
 51  40.520318   127.0.0.1   127.0.0.1     OpcUa       78    UA Secure Conversation Message: CreateSessionRequest
 59  40.521318   127.0.0.1   127.0.0.1     OpcUa     1490    UA Secure Conversation Message: CreateSessionResponse
 61  40.521318   127.0.0.1   127.0.0.1     OpcUa      154    UA Secure Conversation Message: ActivateSessionRequest
 63  40.521318   127.0.0.1   127.0.0.1     OpcUa      104    UA Secure Conversation Message: ActivateSessionResponse
 65  40.522318   127.0.0.1   127.0.0.1     OpcUa      136    UA Secure Conversation Message: ReadRequest
 67  40.522318   127.0.0.1   127.0.0.1     OpcUa      348    UA Secure Conversation Message: ReadResponse
 69  40.531319   127.0.0.1   127.0.0.1     OpcUa      141    UA Secure Conversation Message: BrowseRequest
 71  40.532319   127.0.0.1   127.0.0.1     OpcUa      205    UA Secure Conversation Message: BrowseResponse
 73  40.532319   127.0.0.1   127.0.0.1     OpcUa      171    UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsRequest
 75  40.532319   127.0.0.1   127.0.0.1     OpcUa      150    UA Secure Conversation Message: TranslateBrowsePathsToNodeIdsResponse
 77  40.533319   127.0.0.1   127.0.0.1     OpcUa      231    UA Secure Conversation Message: ReadRequest
 81  40.533319   127.0.0.1   127.0.0.1     OpcUa      440    UA Secure Conversation Message: ReadResponse
 83  40.534319   127.0.0.1   127.0.0.1     OpcUa      141    UA Secure Conversation Message: BrowseRequest

⊞ Frame 69: 141 bytes on wire (1128 bits), 141 bytes captured (1128 bits)
⊞ Raw packet data
⊞ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ Transmission Control Protocol, Src Port: 50800 (50800), Dst Port: 48010 (48010), Seq: 1898, Ack: 9287, Len: 101
⊟ OpcUa Binary Protocol
    Message Type: MSG
    Chunk Type: F
    Message Size: 101
    SecureChannelId: 6864310
    Security Token Id: 1
    Security Sequence Number: 55
    Security RequestId: 5
  ⊟ OpcUa Service : Encodeable Object
    ⊞ TypeId : ExpandedNodeId
    ⊟ BrowseRequest
      ⊞ RequestHeader: RequestHeader
      ⊞ View: ViewDescription
        RequestedMaxReferencesPerNode: 65536
      ⊟ NodesToBrowse: Array of BrowseDescription
          ArraySize: 1
        ⊟ [0]: BrowseDescription
          ⊞ NodeId: NodeId
            BrowseDirection: Forward (0x00000000)
          ⊞ ReferenceTypeId: NodeId
            IncludeSubtypes: True
            NodeClassMask: 0
            ResultMask: 63

0000  45 00 00 8d 20 9e 40 00  80 06 00 00 7f 00 00 01   E... .@. ........
0010  7f 00 00 01 c6 70 bb 8a  19 2f c1 aa 6c c2 6b f4   .....p.. ./..l.k.
0020  50 18 00 1e 29 79 00 00  4d 53 47 46 65 00 00 00   P...)y.. MSGFe...
0030  b6 bd 68 00 01 00 00 00  37 00 00 00 05 00 00 00   ..h..... 7.......
0040  01 00 0f 02 02 00 00 29  bd 68 00 15 91 ab 56 97   .......) .h....V.
0050  d6 cf 01 44 42 0f 00 00  00 00 00 ff ff ff ff 10   ...DB... ........
0060  27 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   '....... ........
0070  00 00 00 00 00 00 01 00  01 00 00 00 00 5d 00 00   ........ .....]..
0080  00 00 00 2f 01 00 00 00  00 3f 00 00 00            ...../.... .?...
```

Obrázek 4.3: OPC UA mapování v programu Wireshark

4.3.4 Wireshark

Program Wireshark [17] umožňuje zachytávat komunikaci mezi UA serverem a klientem, překládat ji zpět do OPC UA zpráv a zobrazovat, v jaké formě jsou tato data posílána (viz obrázek 4.3).

Tento program byl užitečný při studiu OPC UA specifikace, a to hlavně části týkající se komunikace, služeb a mapování.

5. Nástroje pro OPC UA firmy Softing Industrial Automation GmbH

K vývoji OPC UA serveru byl využit nástroj OPC UA C++ toolkit, dále označovaný jako toolkit. Součástí toolkitu jsou hotové příklady, přeložené i nepřeložené části kódu, hotové příklady a nástroje určené k testování aplikace.

Dokumentace obsahuje srozumitelné shrnutí znalostí o OPC UA potřebných k práci s toolkitem a vytváření vlastních aplikací, popis základního zacházení s toolkitem, tutoriál s příklady vhodný pro pochopení principů toolkitu k vytváření UA aplikací, příručku programátora popisující řešení častých problémů a referenční příručku.

Toolkit lze použít pro vytváření klientů i serverů. Obsahuje společné části, část pro stavbu serveru a část pro stavbu klienta. Tutoriál i programátorská příručka jsou rozděleny na vývoj obou aplikací.

K prvnímu spuštění aplikací stačí přeložit kterékoliv z hotových řešení a vygenerovat certifikát. K vytvoření dalšího projektu je třeba připojit přeložené knihovny a hlavičky ze složek *api* a *core*.

5.1 Životnost objektů

Toolkit používá k odkazování na objekty chytré ukazatele (smart pointers), které udržují informaci o počtu referencí na objekt. Jakmile počet referencí klesne na nulu, objekt je automaticky smazán. Tímto způsobem se zefektivňuje a automatizuje hospodaření s pamětí. Pokud navíc při skončení aplikace ve vývojovém režimu zůstává některý objekt nesmazán, překladač zahlásí chybu a programátor zjistí, že je nutné najít objekty, které paměť blokují.

Toolkit namísto přímého vytváření objektů obalujících uzly adresního prostoru doporučuje používat statické metody, které vrací chytrý ukazatel na tento objekt. Tyto reference počítá i mezi uzly adresního prostoru. Pokud jsou tedy uzly navzájem závislé na jednom rodičovském uzlu a pokud se reference na tyto uzly neukládají, po smazání rodičovského uzlu se rekurzivně smaže celý strom uzlů, který z něj vychází.

Toolkit ve většině případů předává metodám ukazatel na objekt namísto objektu samého z důvodu šetření času a paměti. Výjimkou v tomto ohledu je třída *DataValue* implementující univerzální hodnotu OPC UA.

5.2 Inicializace aplikace

Hlavní funkce aplikace nejdříve načte toolkit metodou *loadToolbox*. Pokud je třeba využít vlastní třídy *Creator*, zadá se této metodě jako parametr. Opačnou funkcí je funkce *unloadToolbox*, která toolkit uvolní. Tuto funkci je nutné zavolat před ukončením aplikace.

Statickou metodou *instance* třídy *Application* lze kdekoliv v programu získat ukazatel na vlastní UA aplikaci. Aplikaci se nastaví popis, například URI a jméno aplikace nebo její typ, který označuje, zda je aplikace server, klient, obojí nebo průzkumný server (Discovery Server). Dále se aplikaci musí nastavit certifikáty a zavolat metodu *start*. Opět existuje funkce *stop*, která aplikaci zastaví a která musí být zavolána před uvolněním toolkitu a ukončením programu.

Po nastartování je potřeba v aplikaci vytvořit přípojný bod, na který se klient může připojit. Přípojný bod obsahuje URL, na které poslouchá, nastavení zabezpečení a uživatelskou politiku, která určuje, s jakým uživatelem se může klient připojit. Přípojný bod je třeba dále přidat do aplikace a otevřít. Po přidání přípojných bodů se obvykle vytváří vlastní adresní prostor.

Je třeba mít na paměti, že aplikace po nastartování běží v jiném vlákně. Server z příkladu v tutoriálu čeká na příkazy uživatele z příkazové řádky, díky nimž lze manipulovat se serverem přímo, bez použití klienta. Tento přístup také zajišťuje, že hlavní vlákno programu nedoběhne do svého konce a neskončí tím celá aplikace.

5.3 Creator

Za pomoci toolkitu lze během krátké doby vytvořit funkčního klienta nebo server, aniž by bylo potřeba zasahovat do vnitřních metod serveru. Toolkit ale umožňuje programátorům přepsat některé z rutinních metod tak, aby odpovídaly návrhu programu. K tomuto účelu slouží třída *Creator*, kterou lze zdědit, přepsat požadované funkce a nový *Creator* při nahrávání toolkitu předat. Toolkit poté bude využívat přepsané funkce namísto původních.

Za pomoci *Creatoru* lze změnit funkce spouštěné na serveru při přijetí požadavku na čtení či psaní, manipulací s monitorovanými položkami, vytváření relací a mnoha dalších vnitřních funkcí aplikace.

Příkladem může být situace, kdy uzel získává hodnotu z jiného zdroje. Jednou možností je přepsání metody uzlu umožňující čtení. Pokaždé, když bude požadována hodnota uzlu, vykoná se nová metoda namísto původní. Toto řešení nevyžaduje přepsání metody *Creatoru*.

Druhou možností je přepsání metody *Creatoru* zachytávající dotazy na čtení. V tomto případě lze vyhodnotit všechny dotazované uzly naráz. Je však nutné vyfiltrovat ty uzly, které se mají číst nestandardně.

Dalším příkladem může být omezování monitorovací frekvence u monitorovaných položek. Při dotazu na vytvoření nebo změnu monitorované položky lze nastavit vlastní maximální frekvenci a v odpovědi na požadavek vrátit příslušný stavový kód operace.

5.4 Informační model

Manipulování s uzly probíhá pomocí obalových tříd. Po zavolání statické metody *create* je vytvořen uzel a jeho chytrý ukazatel je vrácen. Uzlu se v konstruktoru nastaví jeho parametry a zavolá se jedna z funkcí, která ho vloží do adresního prostoru.

- Metoda *init* vytvoří objekt z daného uzlu jako typu a automaticky vygeneruje všechny ostatní povinné uzly, které ještě nebyly vytvořeny.

- Metoda *insert* vloží do adresního prostoru pouze aktuální uzel.
- Metoda *insertTree* vloží do adresního prostoru všechny uzly, včetně aktuálního, které jsou navázány na aktuální uzel a nejsou dosud přítomny v adresním prostoru.

Pokud je vytvářen uzel s typem, který nebyl definován, je třeba tento typ nejprve vytvořit. Typ se vytvoří jako instance uzlu základního typu, které se pak nastaví atributy a navážou se na ní dceřiné uzly, u kterých se označí, zda jsou povinné nebo nepovinné. Uzel se sváže referencí *hasSubtype* s typem, který rozšiřuje, a vloží se do adresního prostoru pomocí metody *insert*.

Uzly je možné vyhledávat podle identifikátoru. Lze je procházet pomocí prohledávaných jmen, dotazovat se na typ, identifikátor a prohledávané jméno, nastavovat a získávat hodnoty jejich atributů a procházet proměnné. U proměnných lze získávat jejich hodnotu a hodnoty jejich vlastností.

Při vytváření vlastních uzlů je vhodné vytvořit i zděděnou obalovou třídu pro tento typ uzlů, která může tento uzel vytvořit automaticky a umožní snazší manipulaci s uzly. Základní obalová třída umožňuje uložení ukazatele na obalový objekt a jeho opětovné vrácení na jiném místě, což může na požadovaném místě usnadnit práci s tímto uzlem.

Doporučený postup je vytvořit typ uzlu před vytvořením prvního objektu daného typu, což může opět řešit obalová třída, nebo vytvořit všechny typy najednou před vytvořením prvních instancí. Tyto typy by před ukončením aplikace měly být opět odstraněny.

Toolkit zjednodušuje zápis a čtení některých atributů, jako identifikátoru, prohledávaného jména, práva na zápis a čtení, datový typ hodnoty uzlu typu proměnné a jiné.

Hodnotu implementuje třída *DataValue*, které lze zadat čas zdroje, čas serveru, kód hodnoty a její obsah. Obsah objektu *DataValue* obstarává objekt *Value*, kterému lze zadat jakýkoliv datový typ a poté se na tento typ zeptat. Tím je hodnota proměnné nezávislá na svém datovém typu. Je ovšem nutné datový typ proměnné znát ke správné interpretaci hodnoty.

Toolkit umožňuje vytvoření uzlů metod vytvořením třídy zděděné od třídy *Method* a přepsání její funkce *handleExecuteRequest*. Poté stačí vytvořit instanci této třídy, nastavit její atributy, nastavit její vstupní a výstupní proměnné a připojit ji do adresního prostoru. Poté může klient tuto funkci zavolat se svými hodnotami proměnných.

Další funkcí je ruční vytvoření notifikací upozorňujících na proběhlé události. Těmto notifikacím se nastaví typ události, závažnost, čas a případně zdroj nebo hodnota. Na závěr se notifikace odešle klientům přijímajících notifikace událostí pomocí metody *report*, které se jako parametr nastaví zdrojový uzel. Jinou možností je povolení objektu k registraci klientem k notifikování o událostech. Předdefinovaný uzel Server má tuto možnost v základu povolenou.

6. Vývoj OPC UA serveru pro řídicí systém REX

Při vytváření OPC UA serveru pro řídicí systém REX bylo využito předpřipraveného příkladu dodaného spolu s toolkitem. Prvním krokem byl návrh funkce serveru a synchronizace jeho dat s exekutivou REXu. Následoval vývoj serveru podle návrhu, který ovšem namísto použití diagnostického protokolu pouze vypisoval data na obrazovku. Nejvíce času si vyžádal návrh adresního prostoru, který byl mnohokrát přetvořen. Posledním krokem bylo napojení diagnostického protokolu REXu na připravených místech.

6.1 Využití příkladu z tutoriálu toolkitu

Příklad na vytvoření OPC UA serveru byl složen tak, jak je popsáno v kapitole 5.2 Inicializace aplikace a 5.4 Informační model. Příklad představoval ovládání mrazáku s čidly, stavy a metodami.

Adresní prostor byl vytvořen tak, že při prvním volání metody *create* objektu *Refrigerator* se nejdříve vytvořil typ pro tento objekt, na který se uložil ukazatel do statické proměnné *s_typeDeclarationNode* třídy *Refrigerator*. Až poté byl vytvořen samotný objekt s pomocí této statické proměnné. Jak typ, tak objekt samotný využily statické metody, která vytvořila dceřiné uzly s předem známými identifikátory. Toolkit tedy při zavolání metody *init* nevygeneroval uzly samovolně. Každý nový objekt *Refrigerator* měl svůj definovaný jmenný prostor, identifikátory tak mohly být konstantní pro stejné uzly proměnných v objektu.

Každý objekt *Refrigerator* se ukládal do seznamu a před ukončením programu byl smazán. Stejně tak byla před ukončením programu zavolána funkce *uninitialize*, která manuálně smazala typ pro objekt *Refrigerator*.

6.1.1 Hlavní funkce

Z původní stavby popsané v kapitole 5.2 Inicializace aplikace byla ponechána většina funkcí. Hlavní změnou je vytvoření třídy *AddressSpace* s metodami *load*, *reload* a *clean*, do kterých se přesunulo vlastní vytváření a mazání adresního prostoru.

Celý server je vytvořen jako konzolová aplikace. Při chybě se chyba vypíše na obrazovku a v případě kritické chyby se program po stisknutí klávesy ukončí. Pokud načtení adresního prostoru proběhne v pořádku, zobrazí se uživateli nápověda, kde je popsáno další ovládání aplikace pomocí standardního vstupu. Aplikaci lze zavřít nebo znovu načíst adresní prostor.

6.1.2 Creator

Pro další vývoj aplikace byla vytvořena třída *RexCreator*. Tato třída umožnila změnu vnitřních pochodů UA serveru. V první verzi pouze filtrovala typ uzlu a případně zobrazovala jeho hodnoty a zda se zapisuje či čte. Instance třídy *RexCre-*

ator byla vytvořena v hlavní funkci a předána jako parametr metodě *loadToolbox*, server tak začal používat pro čtení a zápis přepsané metody.

6.1.3 Vytvoření adresního prostoru

Třída *AddressSpace* obsahuje statickou proměnnou *s_instance*, ve které je uložena její instance. Výsledkem je jediná instance třídy *AddressSpace*, kterou je možné volat z kteréhokoliv místa v programu. Obsahuje všechny metody a data týkající se vytváření, udržování a mazání adresního prostoru, včetně typů objektů, a další prvky určené k vlastní synchronizaci s exekutivou REXu. Tato instance se vytvoří po zavolání metody *activate* s názvem stroje jako parametrem. Název stroje je pak použit jako část jmenného prostoru, tím je možné mít pod kontrolou duplicitu jmenných prostorů. Stačí použít pro každou exekutivu jiný název stroje.

6.2 Struktura dat v REXu

Pro stavbu serveru, který zobrazuje data z běžící exekutivy, je důležité pochopit strukturu těchto dat. Struktura dat je podobná zobrazení programu RexView, kde jsou bloky uspořádány do stromové struktury a u každého bloku je seznam parametrů s hodnotami, typem a omezením minimem a maximem.

Účelem serveru je zobrazení všech úloh (Task), jejich bloků a parametrů v podobné struktuře, jako je zobrazuje program RexView (viz obrázek 6.1). Jiné části exekutivy zobrazeny být nemají [18].

6.2.1 Exekutiva

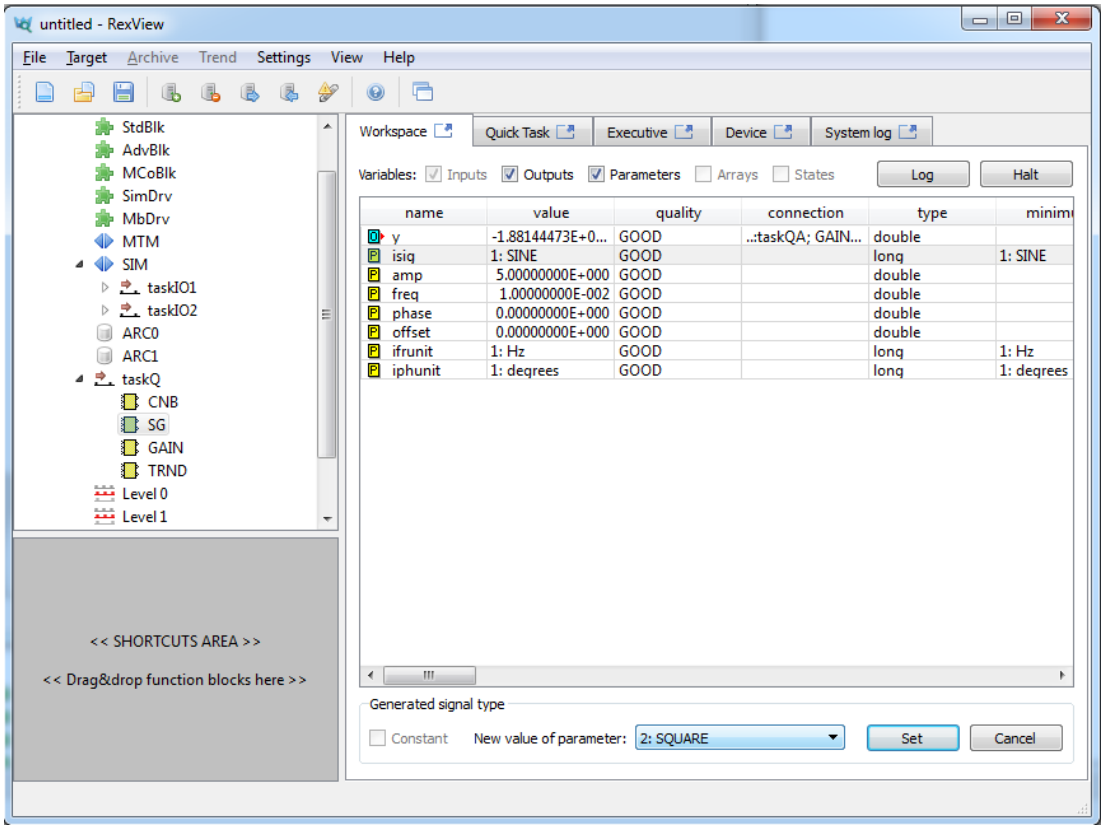
Exekutiva je zobrazena jako blok, ke kterému se připojují jako další bloky knihovny, ovladače a úlohy. Úlohy mohou být napojené v různých úrovních podle priority přímo na blok exekutivy nebo na některý ovladač.

6.2.2 Bloky

Základní zobrazovanou jednotkou je blok. Blok v sobě obsahuje své vstupy, výstupy, parametry, stavy a pole. Pole byla v prvním prototypu toho programu vynechána. V programu RexView se vybraný blok zobrazuje jako seznam jeho proměnných, které jsou seřazené podle svého účelu – vstupy, parametry, stavy, výstupy. U těchto proměnných je zobrazena hodnota, omezení rozsahu, typ a je zde možnost hodnotu u některých proměnných nastavit.

6.2.3 Subsystém

Subsystém je v programu RexView zobrazen jako klasický blok, který ale ve stromové struktuře jej lze rozbalit a zobrazit tak bloky, které do něj náleží. Z pohledu programu RexView není mezi subsystémem a blokem žádný další rozdíl.



Obrázek 6.1: Ukázka programu RexView se stromem bloků na levé straně a parametry bloku *SG* na pravé straně

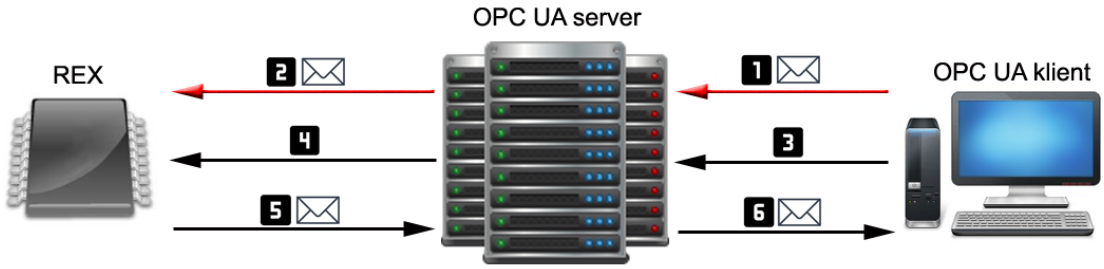
6.2.4 Úloha (Task)

Úlohu v programu RexView lze považovat za rozšíření subsystému. Hlavním rozdílem je, že úloha je vždy navázána na exekutivu nebo na ovladač a lze ji považovat za kořen podstromu obsahující další subsystémy a bloky. Existence úlohy v úloze není možná, stejně jako existence bloku nebo subsystému bez úlohy jako předka.

6.3 Princip synchronizace

K zobrazení správných dat z exekutivy je třeba UA server s exekutivou synchronizovat. Kvůli možnosti dlouhé doby odezvy exekutivy a kvůli minimalizování přenosu dat je lepší využít hromadnou synchronizaci, kdy jsou požadovaná data synchronizována v jednom dotazu. Synchronizace probíhá asynchronně, aby se neblokovaly funkce serveru. Není možné synchronizovat všechny bloky. UA server je využíván jako vyrovnávací paměť. Synchronizována jsou jen klientem požadovaná data, která jsou moc stará, nebo data, která jsou pravidelně monitorována (viz obrázek 6.2).

Při vytvoření adresního prostoru se načte celá struktura exekutivy, podle které se vytvoří stromová struktura objektů v serveru včetně parametrů a jejich vlastností, ovšem bez úvodní hodnoty. Synchronizují se pouze hodnoty parametrů bloků, subsystémů a úloh, ostatní uzly mají zakázané právo na zápis a jsou brány jako konstantní, dokud není třeba přehrát celý adresní prostor serveru.



Obrázek 6.2: Při zápisu hodnoty do serveru (1) se hodnota propíše do REXu (2). Při čtení hodnoty klient zašle serveru požadavek (3). Pokud server nedisponuje dostatečně aktuální hodnotou, zašle požadavek REXu (4), který požadovanou hodnotu vrátí (5). Na závěr pošle server klientu požadovaná data (6).

K synchronizování slouží třída *SynchronizationStack*, jejíž instanci obsahuje třída *AddressSpace*. Třída *SynchronizationStack* obsahuje seznamy uzlů, které je třeba synchronizovat, seznam monitorovaných uzlů s počtem monitorovaných položek, žádosti ke čtení, které budou muset být po synchronizaci zpracovány znovu, a mutex, zámek určený k zamezení přístupu k datům v kritické sekci.

Pokud chce klient přečíst synchronizovanou hodnotu, která obsahuje příliš stará data, zařadí se ukazatel na uzel do synchronizačního seznamu ke čtení a ukazatel na UA požadavek do seznamu synchronizovaných požadavků. Po synchronizaci se požadavek zpracuje znovu a klientovi se odešle žádaná hodnota.

Pokud je vytvořena monitorovaná položka na synchronizovaný uzel, přidá se uzel do seznamu pravidelně synchronizovaných a před synchronizací se sloučí se seznamem ke čtení. Seznam pravidelně synchronizovaných uzlů se po synchronizaci nevyprazdňuje. Uchovává si počet monitorovaných položek daného uzlu. Pokud je smazána poslední monitorovaná položka, smaže se i záznam ze seznamu.

Pokud klient zapíše do synchronizované proměnné, uloží se ukazatel na uzel do synchronizačního seznamu na psaní a provede se zápis do hodnoty uzlu.

Během synchronizace se nejprve hodnoty uzlů určených k zapsání zapíší do exekutivy. Zapisuje se aktuální hodnota na uzlu. Po zapsání se z exekutivy přečtou hodnoty, které jsou označeny ke čtení nebo jsou monitorovány. Přečtené hodnoty se zapíší do příslušných uzlů. Uložené UA požadavky čtení (ReadRequest) jsou znovu zpracovány, přičemž se využijí aktuální hodnoty uzlu a odpověď se odešle klientovi (viz obrázek 6.2). Všechny čtené, monitorované a zapisované hodnoty jsou čteny nebo zapsány pouze jednou. Čtení, zápis a vytváření monitorovaných položek se týká pouze synchronizovaných uzlů, ostatní uzly se zpracují klasickým způsobem.

6.3.1 Synchronizační vlákno

Kvůli potřebě asynchronního zpracování byla vytvořena třída *SynchronizationThread* s metodou *run*, která vytváří a obaluje nové vlákno, ve kterém běží synchronizace. Tuto třídu obsahuje třída *AddressSpace*, která vlákno spouští po vytvoření adresního prostoru a zastavuje před smazáním.

Tato funkce periodicky kontroluje, zda je třeba něco synchronizovat. Po testu a případné synchronizaci se vlákno uspí na dobu zadanou v konstantě. Není zde žádné probuzení na signál, kde by vlákno bylo probuzeno až při samotném čtení,

kvůli monitorovaným položkám.

Před samotnou synchronizací funkce se zamkne mutex, seznam monitorovaných uzlů se sloučí se seznamem uzlů ke čtení do nového seznamu, seznam uzlů k zapsání se pouze zkopíruje. Nejde o zbytečně využitou paměť, po zkopírování do nového seznamu se staré seznamy smažou a odemkne se mutex. Tímto způsobem může server po zkopírování zpracovávat další dotazy, aniž by musel čekat na odpověď REXu.

6.3.2 Metody Creatoru

RexCreator přepisuje metody na zpracování požadavků na čtení a psaní. Pokud klient čte synchronizovanou hodnotu, která není dostatečně aktuální, umístí server dotazovaný uzel do seznamu k synchronizaci a nechá dotaz nesplněný. Pokud klient zapisuje do synchronizované proměnné, data se zapíší, uzel se zařadí do seznamu k synchronizaci, dotaz je ukončen a odešlou se výsledky.

Při vytváření monitorované položky se inkrementuje počet synchronizovaných jednotek v seznamu synchronizovaných uzlů. Pokud uzel v seznamu není, přidá se a je mu nastaven počet monitorovaných položek na jednu. Při mazání se počet monitorovaných položek dekrementuje a při dosažení nuly je uzel ze seznamu odebrán.

Při vytváření i úpravě monitorovacích uzlů je kontrolována perioda vzorkování. Pokud je perioda příliš malá, než aby mohla být v každém intervalu získána hodnota uzlu z REXu, je zvýšena na minimální povolenou velikost. Minimální povolená velikost intervalu je přímo úměrná velikosti synchronizačního intervalu. Maximální frekvence vzorkování je menší než synchronizační frekvence.

V metodě pro zápis do uzlu je navíc vytvořena kontrola pro rozsah hodnoty. V případě, že daná synchronizovaná proměnná obsahuje vlastnosti specifikující omezení hodnoty a zároveň je její hodnota číselná, se daná hodnota zapíše pouze, pokud spadá do povoleného intervalu.

6.4 Adresní prostor

Adresní prostor spravuje třída *AddressSpace*. Při vytváření adresního prostoru instance této třídy nejprve vytvoří všechny potřebné typy. Poté je načítán vlastní adresní prostor procházením bloků REXu pomocí diagnostického protokolu a zároveň si *AdressSpace* ukládá ukazatele na vytvořené úlohy. Nakonec se spustí synchronizační vlákno a server je připraven k použití.

Při ukončení programu nebo přehrání adresního prostoru je vypnuto synchronizační vlákno, smazány všechny úlohy, s nimiž se rekurzivně smažou ostatní bloky. Pokud jde pouze o přehrání adresního prostoru, znovu se načte adresní prostor pomocí diagnostického protokolu a je opět spuštěno synchronizační vlákno. Pokud jde o ukončení programu, smažou se všechny typy uzlů a program může být ukončen.

Typy jsou definované s předem známými identifikátory. Toho je dosaženo využitím jmenného prostoru *urn:Rex:TypeDeclaration* a konstantních číselných identifikátorů.

Vytvoření globální instance *AddressSpace* se děje pomocí statické metody *activate*, které se v parametru *machineName* předá název stroje. Hodnota tohoto pa-

rametru je poté použita v jmenných prostorech všech bloků. Jmenný prostor bloku obsahuje společný začátek své URI a je rozšířen o svou cestu od jednotlivých úloh přes subsystémy až k sobě samému. URI bloku vnořeného v jednom subsystému má podobu *urn:Rex:<machineName>:<taskName>:<subsystemName>:<blockName>*. Všechny názvy použité v URI jsou totožné s názvy bloků v REXu.

Každý blok má svůj jmenný prostor, který je dán jeho umístěním. Identifikátory uzlů používají číselné hodnoty, kde blok samotný má identifikátor 1 a identifikátor se pro každý prvek bloku inkrementuje. Při správném zacházení je tímto způsobem zamezeno záměně jednoho uzlu za jiný v případě, že jeden klient je připojen na více exekutiv REXu. Správným zacházením je myšlena vhodná volba unikátního *machineName*.

6.5 Struktura dat serveru

K vytvoření vlastního serveru bylo třeba navrhnout vhodnou strukturu. Specifikace UA dovoluje variabilní stavbu adresního prostoru, existovalo tedy velké množství možností. Některé návrhy se ukázaly po čase z nějakého důvodu nevhodné, další byly přetvořeny při pochopení a využití dalších možností toolkitu. Cílem bylo najít optimální strukturu, kde by bylo jednoduché filtrovat synchronizované proměnné, adresní prostor odpovídal co nejvíce vizualizaci programu RexView a zobrazovaná data zůstala přehledná.

6.5.1 Bloky

Bloky byly vytvořeny jako objektové uzly a byla pro ně vytvořena obalová třída *UniversalBlock*. Subsystém je pouze rozšířením obyčejného bloku a úloha je pouze rozšířením subsystému. Pro tyto dva typy vytvořeny obalové třídy *Subsystem* a *Task*. Stejným způsobem, jako od sebe dědí obalové třídy, od sebe dědí i jejich typy v adresním prostoru. Všechny obalové třídy v sobě mají metodu *initialize*, která vytvoří jejich typ v adresním prostoru. Pokud daná třída dědí od nějaké jiné, spustí nejdřív inicializaci rodiče. Tímto způsobem se inicializují všechny požadované typy. Opačným způsobem funguje metoda *uninitialize*, která před smazáním typu nejprve smaže typy, které se od něj dědí. Při inicializaci celého adresního prostoru se inicializují listy tohoto stromu a při ukončení je smazán pouze kořen.

Základní třídou pro všechny uzly vytvořené REXem je třída *IRexComponent*. Tato třída není obalovou třídou a neexistuje k ní žádný typ v adresním prostoru, nelze ji ani přímo vytvářet, je to společná třída pro uzly objektů i uzly proměnných. Jejím účelem je slučovat data společná pro všechny uzly REXu, jako identifikátor uzlu pro diagnostický protokol. Tato třída je nezbytná kvůli faktu, že neexistuje společná obalová třída pro objekty i proměnné. Třídy, které od těchto obalových tříd dědí, využívají *IRexComponent* jako společného předka. Ačkoliv tato třída neobaluje žádný typ uzlů, obsahuje metody *initialize* a *unitialize*, které se účastní závislostního mazání a vytváření typů v adresním prostoru.

Třída *IObject* dědí od třídy *IRexComponent* a od klasické obalové třídy pro objektové uzly. Vytváří v adresním prostoru typ *RexObjectType*, kterou dědí všechny objektové typy, které jsou vytvářeny v UA serveru pro REX. Tímto způsobem lze

podle typu jednoduše vyfiltrovat, zda je objektový uzel vytvořen REXem. Tuto třídu nelze přímo vytvářet.

UniversalBlock je obalová třída pro uzel představující blok, v adresním prostoru se tento typ nazývá BlockType. Název *UniversalBlock* pochází z rozhodnutí, zda využívat jeden univerzální blok, nebo pro každý typ bloku v REXu vytvořit samostatný typ. Ve výsledku bylo použito univerzálního bloku kvůli velkému množství jednotlivých bloků v systému REX. Tento uzel lze vytvořit pouze statickou metodou *create*, které se předají požadované parametry včetně ukazatele na rodičovský subsystém. Tím je zamezeno tomu, aby blok existoval bez rodičovského subsystému nebo úlohy. Tomuto objektu lze pomocí čtyř funkcí přidat instanci proměnné REXu jako vstup, výstup, parametr nebo stav.

Subsystem je obalová třída pro subsystém a dědí od třídy *UniversalBlock*. V adresním prostoru je typ tohoto uzlu nazván SubsystemType. Ukazatel na instanci této třídy lze předat jako parametr metody *create* při vytváření vnořeného bloku nebo subsystému, ty se pak v adresním prostoru připojí k rodičovskému subsystému.

Třída *Task* dědí od třídy *Subsystem* a její typ v adresním prostoru je nazván TaskType. Tuto třídu lze jako jedinou vytvořit metodou *create* bez ukazatele na rodiče. Tím je v adresním prostoru zajištěna struktura bloků v REXu, kde vrcholem jsou právě úlohy.

Task, *Subsystem* i *UniversalBlock* používají procházející jméno (BrowseName) Block. Pomocí tohoto univerzálního jména lze procházet adresní prostor.

6.5.2 Parametry a vnořené bloky

Jedním návrhem bylo vytvoření dceřiných uzlů typu FolderType pro uzel BlockType. Byly by vytvořeny složky Inputs, Outputs, Stats a Params. Tyto uzly by měly organizovat vstupy, výstupy, parametry a stavy jako své parametry. Další uzel by organizoval bloky subsystému do složky Subsystem. Výhodou by bylo rozdělení parametrů uzlu podle účelu. Nevýhodou by byl složitější strom uzlů, kterým by bylo nutné procházet.

Ve výsledném řešení bylo použito navázání bloků, vstupů, výstupů, stavů i parametrů přímo na objekt bloku. Toto řešení se více podobá zobrazení programu RexView. Mezi jednotlivými parametry lze rozlišovat pomocí procházejícího jména, kde jmenný prostor je *urn:Rex:TypeDeclaration* a jednotlivé názvy jsou Input, Output, Parameter a Status. Všechny hodnoty je povoleno číst, ale hodnoty uzlů označující vstupy a parametry lze zapisovat.

6.5.3 Identifikátor pro diagnostický protokol

Aby byla synchronizace co nejrychlejší, každý uzel si pamatuje svůj identifikátor pro diagnostický protokol. Aby byl tento identifikátor přístupný, byl v první verzi uchován v uzlu RexId, který byl definován jako vlastnost. Tuto vlastnost měl každý objekt i proměnná REXu.

V konečném řešení tato vlastnost u uzlů neexistuje. Toolkit dovoluje programátorům do uzlu uložit libovolný ukazatel. V případě UA serveru REXu je tímto ukazatelem ukazatel na obalový objekt. Identifikátor pro diagnostický protokol je uložen do obalového objektu a lze na ní přistupovat tak, že je použit ukaza-

tel na obalový objekt v uzlu. Tímto způsobem bylo možné odstranit přebývající vlastnost RexId.

6.5.4 Proměnné

Při návrhu struktury adresního prostoru byly navrženy dvě možnosti pro zobrazování proměnných bloku REXu. Proměnné bylo třeba označit tak, aby je šlo poté při čtení nebo zápisu vyfiltrovat a synchronizovat.

První možností bylo využití předdefinovaných typů proměnných, kterým by bylo přiřazeno určené procházející jméno. Při čtení a zápisu by se kontrolovalo procházející jméno uzlu a zjišťovalo by se, zda je daný parametr určen k synchronizaci, nebo zda je potomkem uzlu, který má být synchronizován. Výhodou by bylo využití předdefinovaných typů UA, které ostatní klienti znají, například *AnalogItem* a *DiscreteItem*.

Druhou možností bylo vytvoření vlastního typu základní proměnné, který by obsahoval přidané vlastnosti, jako povolená minima a maxima. Při čtení a zápisu by se kontroloval typ uzlu. Výhodou tohoto řešení by byla větší podobnost s vizualizací v programu RexView.

V současné době je využito řešení s vlastním typem proměnné, nicméně v budoucím vývoji se bude diskutovat o výhodách obou řešení a může být rozhodnuto jinak. Důvod k současnému použití vlastní proměnné je ten, že v programu REX neexistují parametry potřebné k vytvoření proměnných *AnalogItem*, *DicreteItem* a jejich subtypů.

6.6 Propojení s diagnostickým systémem REXu

Posledním krokem k vytvoření prototypu UA serveru pro REX bylo připojení serveru k REXu pomocí diagnostického protokolu. Jelikož s touto eventualitou bylo při vývoji serveru počítáno, stačilo na předem připravených místech použít diagnostický protokol.

6.6.1 Prohledání prostoru

Při nahrávání adresního prostoru je navázáno spojení s diagnostickým protokolem. Toto spojení je poté uloženo v objektu *AddressSpace*. Po navázání spojení jsou od kořene exekutivy rekurzivně prohledávány komponenty REXu. Filtr prohledávání je nastaven na návrat úloh, subsystémů, bloků a ovladačů. Ovladače jsou prohledávány kvůli možnému výskytu závislých úloh.

Ke každému vrácenému bloku, subsystému a úloze je vytvořen obalující objekt pro uzel a jsou mu přidány proměnné. Pokud se jedná o úlohu nebo subsystém, funkce poté prohledá i jeho komponenty, kterým je předán ukazatel na uzel rodiče, který je poté použit k vytvoření dceřiných subsystémů nebo bloků.

Přidání proměnných probíhá filtrovanými dotazy na vstupy, výstupy, parametry a stavy, které jsou zpracovány jednotlivě. Z vrácených proměnných se zjistí minimum, maximum, datový typ a hodnota, které se použijí k vytvoření proměnné v adresním prostoru UA, která se naváže na uzel bloku.

6.6.2 Zápis a čtení

Při zápisu do REXu je ze seznamu uzlů k zapsání vytvoří seznam identifikátorů proměnných bloků a seznam jejich hodnot. Tyto seznamy jsou zaslány exekutivě REXu pomocí diagnostického protokolu, exekutiva hodnoty zapíše do požadovaných parametrů. Diagnostický protokol omezuje počet hodnot k zapsání v jednom požadavku. Při příliš velkém seznamu hodnot k zapsání je posláno požadavků několik tak, aby každý z nich vyhovoval limitu protokolu.

Při čtení je vytvořen ze seznamu uzlů ke čtení pomocí identifikátorů vytvořen seznam parametrů a prázdný seznam hodnot, který je poté naplňován postupně, stejně jako při zpracování požadavků k zapsání. Vrácená hodnota je po otestování datového typu nastavena jako hodnota uzlu. U hodnoty uzlu je časem serveru čas nastavení hodnoty, časem zdroje je čas vrácený diagnostickým protokolem.

Diagnostický protokol REXu musí být používán, po dlouhé době nečinnosti se spojení automaticky uzavře. Pokud během určité doby nedojde ke čtení nebo zápisu do REXu, synchronizační vlákno pošle zprávu, která udrží diagnostický protokol otevřený.

6.6.3 Přehrání exekutivy

Existuje možnost, že se exekutiva REXu změní. V tomto případě začnou všechny dotazy na diagnostický protokol vracet chybu. Diagnostický protokol musí být uzavřen, otevřen znovu a musí být nahrán celý adresní prostor, který se po přehrání mohl změnit. Tuto událost lze řešit ručním zadáním příkazu, který znovu nahraje adresní prostor.

V budoucím vývoji se adresní prostor při této události přehraje automaticky a UA klient, který bude monitorovat server nebo některý z bloků, subsystémů, úloh, či jejich parametrů, dostane oznámení o změně adresního prostoru.

7. Příklady použití

V této kapitole je popsáno testování a použití hotového prototypu UA serveru včetně grafických ukázek. V závěrečné části kapitoly je navržen předběžný postup dalšího vývoje.

7.1 Testování aplikace

Po vytvoření prototypu serveru bylo třeba otestovat jeho funkci na různých příkladech. Pro tyto účely byla spuštěna exekutiva REXu a UA server, který se k této exekutivě připojil. UA server poté načetl adresní prostor z bloků exekutivy a umožnil UA klientům manipulovat s jejími daty. Dopady zápisu, přečtené hodnoty a načtená struktura v adresním prostoru byly zkontrolovány pomocí programu RexView, který byl napojen na stejnou exekutivu jako testovaný UA server.

V REXu existují hotová schémata, na kterých je možné otestovat základní funkce REXu. Tato schémata vznikla pro jednotné testování různých verzí REXu a jsou vhodná i pro testování UA serveru. Exekutivu těchto testovacích schémat se podařilo bezchybně načíst a poté bylo možné zapisovat a číst její data.

Dalším schématem k testování bylo uměle vytvořené schéma se všemi druhy bloků, které jsou v REXu k dispozici. Po přeložení a spuštění bylo i toto schéma načteno správně, čímž bylo ověřeno, že UA server umí zpracovat všechny bloky i jejich parametry.

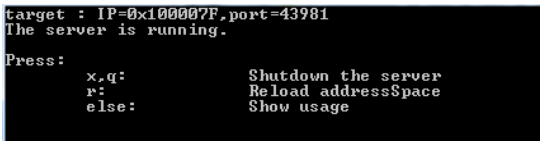
V posledním testu bylo spuštěno schéma s 30 000 bloky, které mělo otestovat možnosti serveru v počtu načtených bloků a dobu nutnou k jejich načtení a smazání. UA server načetl správně i takto rozsáhlou exekutivu. K načtení bylo třeba přibližně 50 sekund, ke smazání 5 sekund. Vzhledem k tomu, že je načtení adresního prostoru jednorázovou akcí, lze považovat tuto dobu za dostatečně krátkou.

7.2 Praktické použití

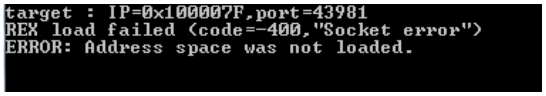
Pro použití stačí spustit UA server, který má nastaveno jméno, cestu k certifikátu a adresu exekutivy, ke které se má připojit. Pokud nahrání adresního prostoru proběhne správně, zobrazí se povolené příkazy k dalšímu ovládání (viz obrázek 7.1). V opačném případě program zobrazí chybu (viz obrázek 7.2). Po načtení adresního prostoru se lze k serveru připojit UA klientem.

Program UaExpert i klient firmy Softing vytvářejí stejnou stromovou strukturu, jako program RexView (viz obrázek 7.3).

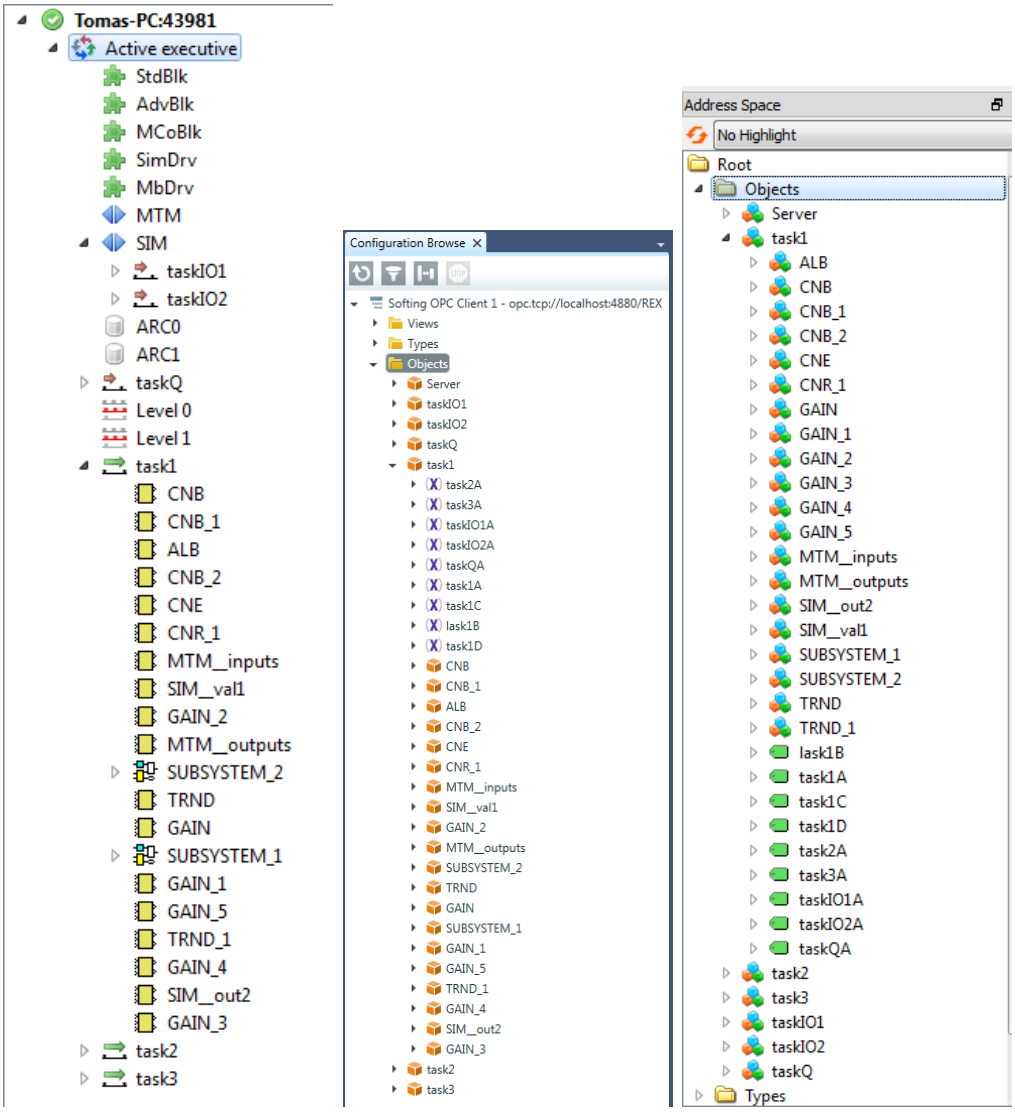
Hodnota, typ a omezení zobrazené klientem odpovídají hodnotě, typu a omezení v programu RexView.



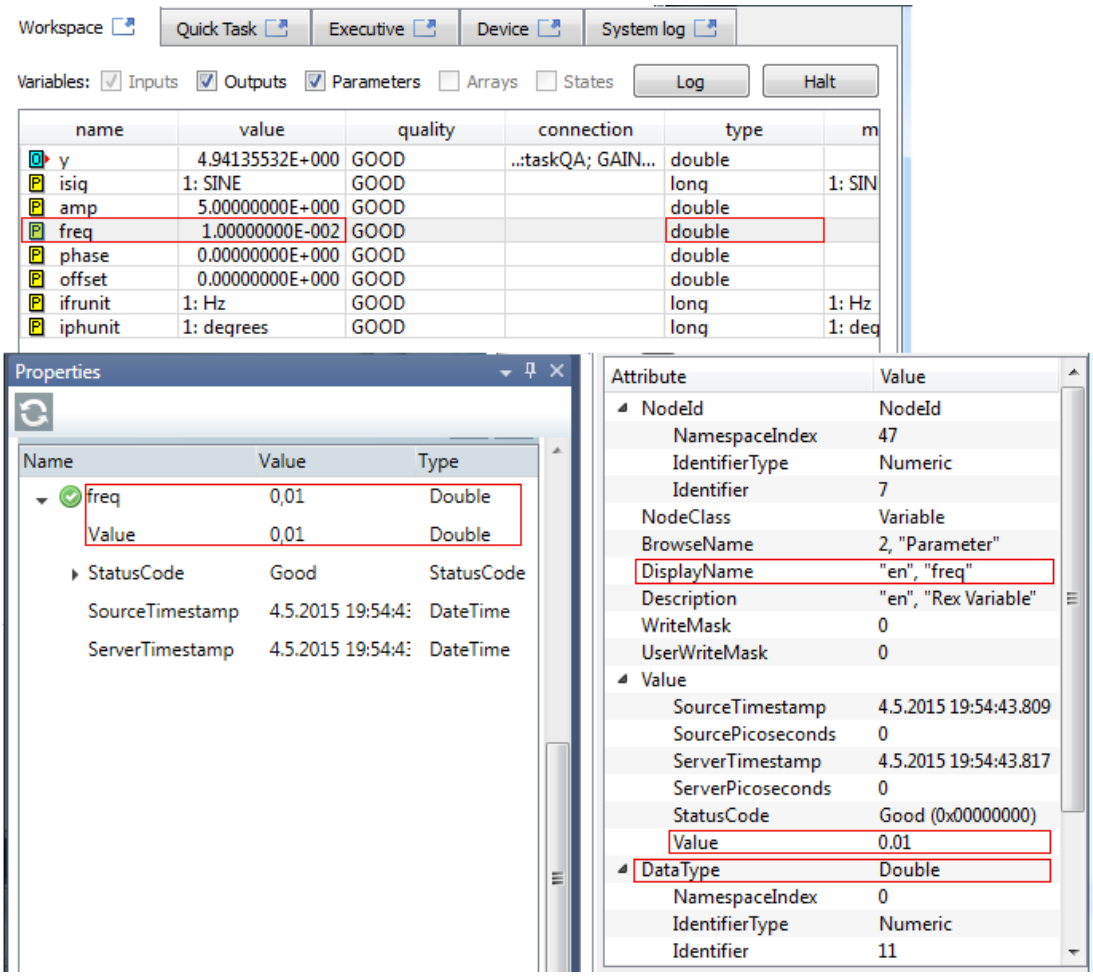
Obrázek 7.1: Úspěšně spuštěný server



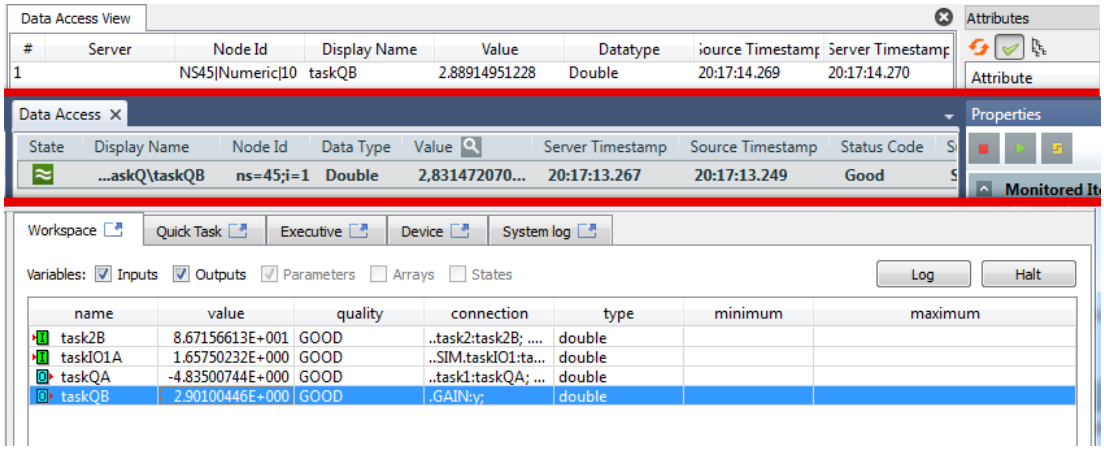
Obrázek 7.2: Chyba serveru



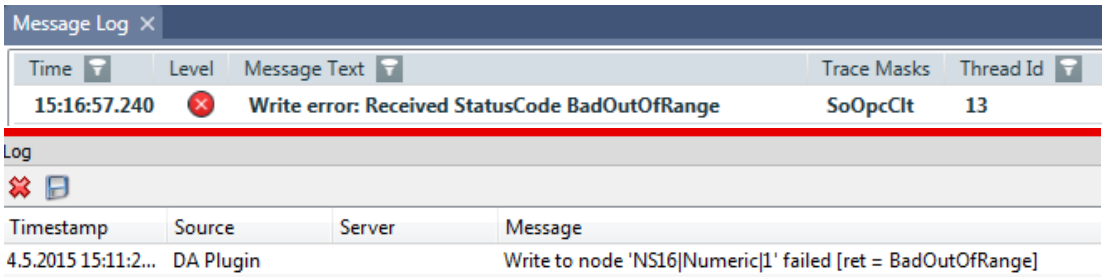
Obrázek 7.3: Stromová struktura bloků, zleva RexView, Softing Client a UaExpert



Obrázek 7.4: Název, typ a hodnota parametru *freq* v programech RexView (nahoře), Softing Client (vlevo dole) a UaExpert (vpravo dole)



Obrázek 7.5: Monitorovaná položka *taskQB* v programech UaExpert (nahoře) a Softing Client (uprostřed) a její hodnota v programu RexView (dole)



Obrázek 7.6: Zápis příliš velké hodnoty, shora Softing Client a UaExpert

Při vytvoření monitorované položky se hodnota uzlu pravidelně synchronizuje podle nastavení monitorovacího intervalu položky a minimálního intervalu odběru (viz obrázek 7.5).

Při zapsání hodnoty se tato hodnota projeví v běžící exekutivě. Při pokusu o zapsání hodnoty, která není v povoleném intervalu, server vrací chybu (viz obrázek 7.6).

7.3 Budoucí vývoj

Popsaný UA server je pouze funkční prototyp. Po diskusi a specifikaci dalších požadavků bude probíhat další vývoj. V dalším rozšíření mohou být použity následující funkce UA serveru.

7.3.1 Zabezpečení

V dalším vývoji bude lépe navrženo zabezpečení serveru. Například bude omezen počet připojení k serveru a počet monitorovaných položek, aby se omezila maximální spotřeba zdrojů systému. Dále by mělo být ošetřeno zpracování chyb diagnostického protokolu a načítání příliš velkého adresního prostoru. V neposlední řadě by měly být nalezeny optimální synchronizační frekvence pro různě rychlá připojení a velikost dat.

7.3.2 Pole

V programu RexView existují u bloků vedle parametrů, vstupů, výstupů a stavů ještě pole hodnot, která v prototypu serveru nebyla implementována. V dalším vývoji tedy mohou být v adresním prostoru serveru zpracována a synchronizována i tato pole.

7.3.3 Historický přístup k datům

OPC UA specifikuje služby, které umožňují historický přístup k datům. Tyto služby jsou při spolupráci s REXem nepoužitelné, protože se mění pouze dotazovaná a monitorovaná data. REX umožňuje svůj vlastní přístup k historickým datům. Přístup k historickým datům může být dalším z budoucích rozšíření prototypu.

Zatím nejpravděpodobnější způsob řešení toho problému spočívá ve vytvořením uzlů funkcí, které by při zavolání vrátily požadovaná historická data přímo od běžící exekutivy REXu. V tomto případě by se žádná data na serveru neukládala, server by sloužil pouze jako prostředník.

8. Závěr

Technologie OPC UA představuje pro řídicí systémy, mezi které patří i REX, vhodné rozšíření, které ulehčuje komunikaci s jinými aplikacemi. Tato technologie je velmi populární a nabízí modernější přístup k předávání a skladování dat než staré OPC.

K dispozici jsou kvalitní nástroje určené k rychlému vývoji a nasazení OPC UA aplikace. OPC UA C++ Development Toolkit se ukázal jako vhodný nástroj k vytvoření vlastního UA serveru. Toolkit obsahuje všechny nástroje potřebné k vytvoření adresního prostoru, umožňuje zachytávání žádostí klienta a následnou synchronizaci údajů s exekutivou REXu. Díky manuálu s názornými ukázkami zkrátil a zpříjemnil vývoj celé aplikace.

Prototyp UA serveru ukládá data z exekutivy REXu tak, že je veřejně dostupný UA klient zobrazuje podobně jako program RexView. Na adekvátních příkladech bylo ověřeno, že do požadovaných proměnných bloků exekutivy lze zapisovat, lze je číst a lze je i monitorovat. Tento prototyp dobře pracuje se zdroji systému, na kterém běží, a minimálně zatěžuje spojení s běžící exekutivou REXu.

V této práci byly popsány základní vlastnosti a funkce technologie OPC UA, porovnány nástroje vhodné k vývoji vlastní UA aplikace a vysvětleny myšlenky, které byly použity při vývoji UA serveru pro řídicí systém REX. Na závěr bylo ukázáno použití toho serveru a byl nastíněn směr jeho dalšího vývoje.

Seznam použité literatury

[1] OPC Foundation *ABB Automation GmbH* [online] [cit. 1.5.2015]. Dostupné z: https://opcfoundation.org/members/view/7

[2] OPC Foundation *Siemens AG* [online] [cit. 1.5.2015]. Dostupné z: https://opcfoundation.org/members/view/353

[3] OPC Foundation *OPC Unified Architecture Specification* [online] Verze 1.01 IEC 62541 [cit. 30.4.2015]. Dostupné z: https://opcfoundation.org/developer-tools/specifications-unified-architecture

[4] OPC Foundation *OPC Unified Architecture / PLCopen Information Model* [online] [cit. 30.4.2015]. Dostupné z: https://www.rexcontrols.cz/media/documents/promotion/Rex_prospekt.pdf

[5] REX Controls s.r.o. *Řídicí systém REX pro pokročilé řízení strojů a procesů* [online] [cit. 30.4.2015]. Dostupné z: https://opcfoundation.org/developer-tools/specifications-unified-architecture/opc-unified-architecture

[6] OPC Foundation *Catalog of tested products using OPC UA* [online] [cit. 30.4.2015]. Dostupné z: https://opcfoundation.org/products

[7] OPC Foundation *Catalog of tested products using OPC UA* [online] [cit. 30.4.2015]. Dostupné z: https://opcfoundation.org/certified-products

[8] OPC programmers' connection *OPC UA SDKs and Toolkits* [online] [cit. 30.4.2015]. Dostupné z: http://www.opcconnect.com/uakit.php

[9] OPC Foundation *ANSI C Stack Source Code* [online] [cit. 30.4.2015]. Dostupné z: https://opcfoundation.org/developer-tools/developer-kits-unified-architecture/ansi-c-stack-source-code/

[10] Softing Industrial GmbH *OPC UA C++ Server and Client Development Toolkits* [online] [cit. 30.4.2015]. Dostupné z: http://industrial.softing.com/en/products/software/opc-development-toolkits/opc-ua-c-development-toolkits-platform-independent/opc-ua-c-server-client-toolkit-for-windows.html

[11] Unified Automation GmbH *ANSI C Based OPC UA Server SDK* [online] [cit. 30.4.2015]. Dostupné z: https://www.unified-automation.com/products/server-sdk/ansi-c-ua-server-sdk.html

[12] Unified Automation GmbH *C++ Based OPC UA Server SDK* [online] [cit. 30.4.2015]. Dostupné z: https://www.unified-automation.com/products/server-sdk/c-ua-server-sdk.html

[13] MatrikonOPC GmbH *OPC UA Embedded Server SDK* [online] [cit. 30.4.2015]. Dostupné z: http://www.matrikonopc.com/opc-ua/embedded/sdk.aspx

[14] Unified Automation GmbH *UaExpert* [online] [cit. 30.4.2015]. Dostupné z: https://www.unified-automation.com/products/development-tools/uaexpert.html

[15] OPC UA C++ Demo Server *OPC UA Embedded Server SDK* [online] [cit. 30.4.2015]. Dostupné z: https://www.unified-automation.com/downloads/opc-ua-servers.html

[16] Softing Industrial GmbH *OPC UA Client* [online] [cit. 30.4.2015]. Dostupné z: http://industrial.softing.com/en/downloads.html

[17] Wireshark Foundation *Wireshark* [online] [cit. 30.4.2015]. Dostupné z: https://www.wireshark.org/

[18] REX Controls s.r.o. *Funkční bloky systému REX, Referenční příručka* [online] Verze 2.10.6 Plzeň, 7.4.2015 [cit. 30.4.2015]. Dostupné z: http://www.rexcontrols.cz/media/DOC/CZECH/BRef_CZ.pdf

Seznam použitých zkratek

OPC UA: OPC Unified Architecture
OPC: Open Platform Communication, dříve OLE for Process Control
OLE: Object Linking and Embeding
OPC Classic - první specifikace OPC
COM: Component Object Model
DCOM: Distributed Component Object Model
SOAP: Simple Object Access Protocol
SOA: Service Oriented Architecture
URI: Uniform Resource Identifier
URL: Uniform Resource Locator
URN: Uniform Resource Name
GUID: Globaly Unique Identifier
XML: Extensible Markup Language
OPC TCP - protokol, kterým se mohou spojit OPC UA aplikace
TCP: Transmission Control Protocol
POST - dotazovací metoda HTTP
HTTP: Hypertext Transfer Protokol
SDK: Software Development Kit
ANSI C - programovací jazyk
C++ - objektový programovací jazyk odvozený od jazyka ANSI C