Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

Bakalářská práce

Vývoj webového OPC klienta

Plzeň 2012 Jiří Faist

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 23. května 2012

Jiří Faist

Abstract

The goal of this work is to implement an *thin client application* that would serve for web clients to communicate and manipulate with *OPC* Servers. In first chapter there are discussed technologies usable for implementation. There are mentioned various approaches to bidirectional communication in web applications with a focus on *WebSocket* technology. In the next chapter there are exposed some implementation principles along with their advantages and disadvantages. In final chapters there are shown some performance test results of demanding components of application along with the performance statistics of reading various numbers of items from the *REX* Control OPC server and briefly described example of use of the application with SVG graphic in web browser.

Cílem práce je implementace aplikace typu tenkého klienta, která umožní webovým klientům komunikaci a manipulaci s *OPC* Servery. V první části práce jsou diskutovány technologie, které jsou použitelné pro vývoj aplikace. Jsou zde také zmíněny různé přístupy k obousměrné komunikaci mezi webovými klienty a servery se zvláštním zaměřením na technologii *WebSocket*. V další kapitole jsou poodhaleny významné principy implementace společně s jejich výhodami a nevýhodami. V závěru práce jsou uvedeny výsledky měření výkonu aplikace se zaměřením na jednotlivé náročné komponenty společně s výsledky testování čtení různého počtu položek z OPC Serveru řídícího systému REX. V závěru práce je stručně popsán příklad použití aplikace s využitím SVG grafiky ve webovém prohlížeči.

Key words: opc, opcbridge, webový klient, tenký klient, websocket, rex control

Obsah

1 Úvod

Základním cílem této práce je vytvořit uživatelsky přátelskou aplikaci, která usnadní přístup k datům na *Opc Serverech*(viz. 2.1). Vzniklá aplikace by měla co nejvíce odpovídat charakteristikám aplikace *tenkého klienta*. Tyto aplikace jsou zpravidla spuštěny na serveru a klient k nim přistupuje pomocí snadno dostupného softwarového prostředku, typicky webového prohlížeče. Tento přístup má mnoho výhod, z nichž nejdůležitější je, že veškerá výpočetní zátěž je prováděna na serveru a klient je zatížen pouze síťovou komunikací. Klient také není nucen instalovat na svém počítači žádný dodatečný software. Tato aplikace by mohla najít využití zvláště pro přístup k datům řídícího systému *REX Controls*, který v sobě má základní implementaci *Opc Serveru*.

Opc Servery mají definované API[1](rozhraní) v *.NET Frameworku*, přes které se k datům přistupuje. Je tedy nasnadě toto rozhraní využít. Z požadavku na tenkost klienta je dalším logickým požadavkem, aby rozhraní vytvořené aplikace bylo přístupné z programovacího jazyce *JavaScript*, který je hlavním programovacím jazykem webových prohlížečů. Aplikace tedy bude mít tvar jakéhosi mostu mezi klientem v *JavaScriptu* a OPC Serverem, ke kterému bude přistupováno z prostředí .NET Framework. Z této úvahy vyplývá druhý požadavek na aplikaci a tím je vytvořit ji natolik obecně, aby mohla být v budoucnu snadno přizpůsobena pro práci i s jinými rozhraními než je rozhraní OPC Serverů.

V první části práce jsou diskutovány použité technologie, které jsou v implementovaném řešení využity. V navazujících kapitolách se již nacházejí popis principů funkce aplikace společně s výsledky testování aplikace. Součástí práce je také jednoduchá ukázka s vizualizací v podobě vektorové SVG grafiky ve webovém prohlížeči, po jejímž prostudování by měl být čtenář schopen aplikaci aktivně využívat.

[1]API - Application Programming Interface

2 Použité technologie

2.1 Opc Servery a Opc .NET API

2.1.1 Standardy OPC

OPC je soubor specifikací, které popisují standardy naplňující specifické potřeby průmyslu. O udržování těchto otevřených standardů se stará nadace OPC Foundation. První standart *OPC Data Access*[1] vznikl na základě spolupráce mnoha světových podniků, zajišťujících automatizaci, za pomoci *Microsoftu*. Tato specifikace je původně postavená na technologiích *Microsoftu* OLE COM[2] a DCOM[3]. Tyto technologie umožňují softwarovým komponentám vzájemnou komunikaci a jsou využívány vývojáři pro vytváření znovupoužitelných komponent, slinkování komponent dohromady a využívání výhod Windows servisů[Mic(2012a)]. Specifikace definuje základní set objektů, rozhraní a metod pro použití při řízení procesů a automatizaci tak, aby zajistila vzájemnou kompatibilitu[OPC(2012)].

OPC aplikace jsou implementovány jako dvojice server/klient. OPC server je program, který konvertuje hardwarový komunikační protokol používaný v PLC[4] na OPC protokol[Cog(1995-2010)]. OPC klient je program, který se potřebuje připojit k hardwarovému zařízení. Používá OPC server pro získání dat ze zařízení nebo pro vyslání příkazu do zařízení (viz. obr. 2.1).

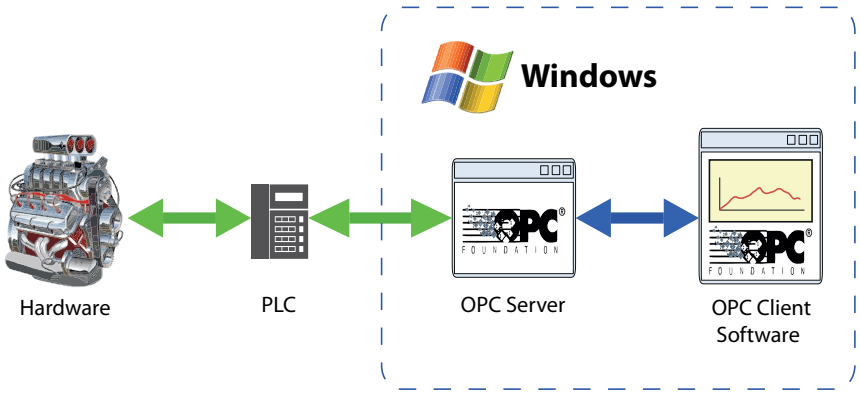
K důležitým vlastnostem OPC patří to, že se jedná o otevřený standard, z čehož vyplývá, že výrobce hardwaru musí poskytnout jediný OPC server pro svá zařízení, aby mohla komunikovat s kterýmkoliv klientem. Z opačného pohledu je patrné, že rovněž vývojářům aplikací stačí zanést do svého produktu vlastnosti OPC klienta a tím budou kompatibilní s mnoha hardwarovými zařízeními.

[1] původně nazývaný *OPC Specification*

[2] *OLE- Object Linking and Embedding; COM - component object model*

[3] *distributed component object model*

[4] *Programmable Logic Controller* - malý průmyslový počítač kontrolující jedno či více hardwarových zařízení



Obrázek 2.1: jednoduché OPC s jedním server/klient připojením

2.1.2 OPC .NET API

OPC .NET(WCF[5])[6] je API pro přístup k OPC serverům z prostředí *Microsoft .NET Frameworku.*

OPC .NET 3.0 je výsledkem spolupráce několika společností, který poskytuje jednoduché .NET rozhraní pro OPC Classic servery. Bylo vyvinuto tak, aby umožnilo klientským aplikacím využívat přednosti .NET prostředí při přistupování k existujícím OPC Classic serverům [OPC(2012)].

Rozhraní OPC .NET 3.0 kombinuje sémantiku základních OPC specifikací OPC *Data Access*, OPC *Alarms and Events* a OPC *Historical Data Access*[7].

Rozhraní také umožňuje automatické znovupřipojení k serveru, jestliže dojde ke ztrátě spojení. Tato vlastnost rovněž zachovává na serveru povědomí o klientu, takže klient nemusí znovu vytvářet své nastavení na serveru. OPC .NET 3.0 API je navrženo tak, aby klientská aplikace byla oddělena od síťového rozhraní. Klientská aplikace už se nemusí vypořádávat s věcmi kolem síťového připojení jako například s výměnami požadavků a odezev(request/response). Tyto úkony obstarává zdrojový kód, který je distribuován s API, a WCF rozhraní.

OPC .NET je distribuováno se třemi generickými klientskými aplikacemi, které byly vytvořeny jako model pro vývoj klientských aplikací a slouží také pro testování nově vyvinutých serverů pro ověření správné funkčnosti.

Na straně serveru je přítomný generický wrapper pro OPC DA, A&E

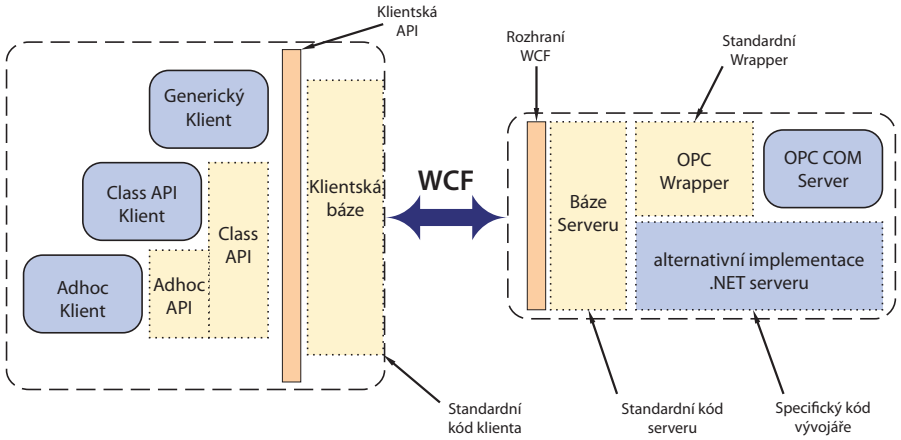
[5]Windows Communication Foundation - součást .NET Framework určená pro tvorbu servisově orientovaných aplikací

[6]dříve nazývané *OPC Express Interface (Xi)*

[7]dále používáno ve zkratkách DA, A&E, HDA

nebo HDA servery. Wrapper je rozdělen do dvou částí, první obstarává WCF rozhraní a druhá se stará o mapování mezi OPC .NET a OPC Classic funkcemi (viz. obr.2.2). Zdrojový kód, který zajišťuje WCF rozhraní serveru je označen jako *báze serveru*. Tato část definuje stejné chování všech OPC .NET serverů a je tedy vždy stejná.

Pro vyhledávání přípustných serverů obsahuje OPC .NET *Discovery server*, který vytváří seznam serverů dostupných pro klientskou aplikaci. K tomu využívá technologii *Microsoft Peer Name Resolution Protocol*(PNRP), která je dostupná na systémech Windows XP a vyšších.



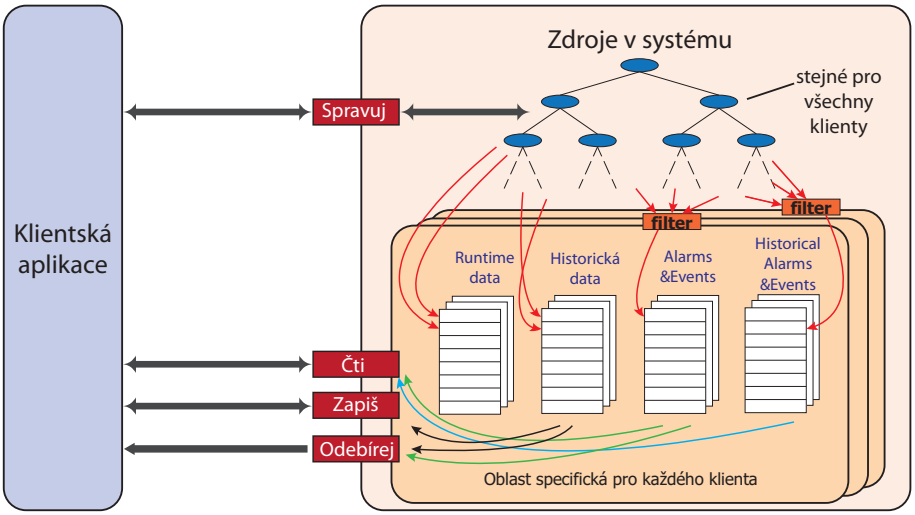
Obrázek 2.2: OPC .NET klient a server

Funkční architektura

Na obrázku 2.3 jsou přiblíženy principy funkce vztahu OPC server/klient. Základem jsou jakési přístupové body serveru definované rozhraními Spravuj(Manage), Čti(Read), Zapiš(Write) a Přihlaš se k odběru(Subscribe). Každý klient si vytváří vlastní seznamy ze zdrojů v systému. Ty jsou pro všechny klienty stejné na rozdíl od vzniklých seznamů, které má každý klient vlastní. Klient poté tyto seznamy může přiřadit k přístupovým bodům. Každý klient může mít více přístupových bodů a ke každému přístupovému bodu může být přiřazeno více seznamů[Neitzel(2009)].

Server tedy disponuje následujícími rozhraními:

- *ServerDiscovery* - používané pro nalezení serverů



Obrázek 2.3: Schéma funkční architektury OPC .NET

- *ResourceManagement* - umožňuje prohledávání zdrojů serveru, vytváření seznamů z těchto zdrojů, vytváření přístupových bodů a přidělování seznamů přístupovým bodům
- *Subscribe* přístupové body mohou být dva typy určené hlavně druhem použitého protokolu:
  - *Callback* - určené pro přístup k datům v rámci jednoho stroje případně místní sítě, protokol TCP
  - *Poll* - určené pro přístup z internetu, webových aplikací atd., protokol HTTP
- *Read* - obsahuje metody pro čtení hodnot ze serveru
- *Write* - obsahuje metody pro změnu stavu serveru a pro zápis hodnot na server

2.2 .NET a Reflexe

2.2.1 .NET Framework

.NET Framework[8] je technologie vyvinutá společností Microsoft. Podporuje řadu programovacích jazyků a snaží se mezi nimi zajistit vzájemnou interoperabilitu. Při kompilaci v .NET se zdrojový kód programu překládá do *Common Intermediate Language*(CIL), což je nejnižší pro člověka čitelný programovací jazyk. Do binárního kódu je program v CIL přeložen až za běhu. CIL je ukládán do assembly, která je buď ve tvaru spustitelného souboru *.exe* nebo ve tvaru *.dll*[9]. Kromě CIL kódu assembly ještě obsahuje svůj Manifest, Metadata a Reference.

2.2.2 Metadata a Reflexe

Každá assembly obsahuje svá Metadata, která jsou tvořena automaticky při překladu. Metadata popisují každý datový typ a člen, který je v modulu nebo v assembly definován nebo je na něj odkazováno a jsou základem pro reflexi v .NET Frameworku.

Reflexe v programování je proces, při kterém je daný program schopný za běhu sledovat a modifikovat svou vlastní strukturu a chování[Hurlbut(1998)]. V .NET zajišťuje reflexi jmenný prostor *System.Reflection*, jehož objekty zaobalují assembly, moduly a datové typy. Tyto objekty mohou být využity k dynamickému vytváření instancí objektů, získávání informace o typu existujícího objektu, vyvolávání jeho metod nebo k přístupu k jeho položkám a vlastnostem[Mic(2012b)]. Jediné co musí uživatel pro použití reflexe znát je jméno datového typu objektu a informace o assembly, ve které je objekt definován. V .NET reflexe funguje na základě parsování výše zmiňovaných Metadat z assembly.

Při vývoji aplikace se nabízeli dva různé způsoby implementace. První možností bylo celé (a nebo značnou část) OPC .NET API přímo zpřístupnit v klientu JavaScriptu. V JavaScriptu by tedy musel být obraz každého objektu a každé metody, které jsou definované v OPC .NET API. Vzhledem

[8]framework - podpůrný prostředek pro programátora obsahující vývojové prostředí, podpůrné knihovny atd.

[9]Dynamic Link Library

k velikosti API se to ale jeví jako značně nevhodné řešení, neboť rozhraní aplikace v JavaScriptu by muselo být rozsáhlé a komplikované.

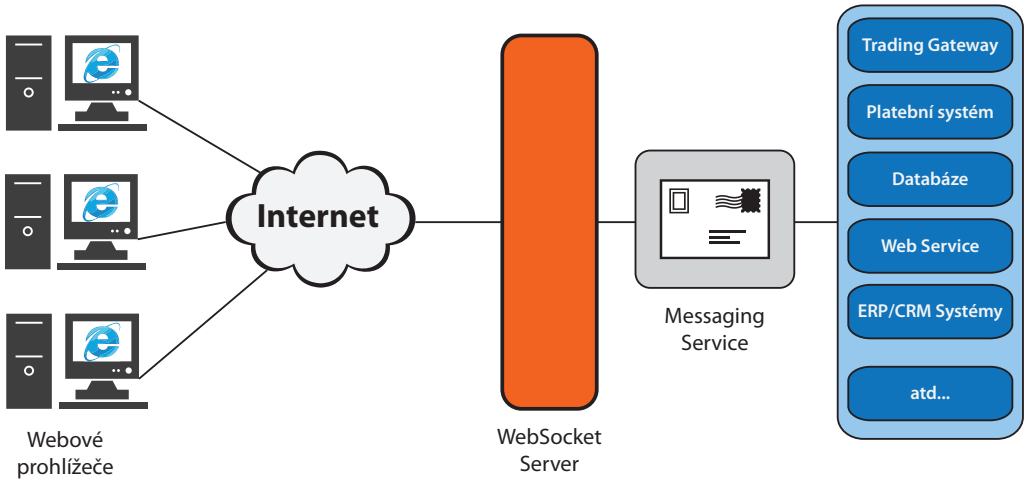
Jako druhé řešení se nabízí vytvořit část aplikace, která v prostředí .NET manipuluje s OPC .NET API za využití reflexe. Hlavní výhodou tohoto řešení je možnost velice jednoduše vyměnit API, ke kterému se klient připojuje. Toto řešení dokonce umožňuje připojovat se k několika API nezávisle na sobě. JavaScriptová část navíc musí v základní verzi obsahovat pouze obecné metody pro vytvoření objektu, zavolání metody nad objektem a nastavení členu objektu na určitou hodnotu. Na druhou stranu zde existují i nevýhody. Nejzávažnější z nich je skutečnost, že přečtení hodnoty nějakého členu objektu přes reflexi nebo vyvolání metody objektu použitím reflexe je mnohem pomalejší než při přímém přístupu (viz. sekce 4.1). Další nevýhodou jsou větší paměťové nároky, protože do paměti se musí načíst množství Metadat o využívaných objektech.

2.3 WebSocket

2.3.1 Úvod do WebSocket technologie

Navrhovanou aplikace si lze představit jako most mezi JavaScriptem a OPC Serverem. K OPC Serveru se přistupuje přes rozhraní definované v .NET. Data mezi prostředími jazyka JavaScriptu a .NET jsou přenášena přes WebSocket server.

WebSocket je protokol, který umožňuje obousměrnou komunikaci mezi klientským skriptem a vzdáleným serverem, který byl v tomto skriptu zvolen [Ian Fette(2011)]. Klientem jsou ve většině případů webové prohlížeče s podporou HTML5. Základní architektura je nastíněna na obrázku 2.4. Protokol se skládá z otevíracího *"handshake"*, po kterém zpravidla následuje výměna informací založená na základě komunikačního protokolu TCP[10]. Cílem této technologie je poskytnout mechanismus pro webové aplikace komunikující obousměrně se servery, který nebude závislý na otevírání velkého počtu HTTP připojení.



Obrázek 2.4: Základní architektura technologie Websocket

[10]Transmission Control Protocol

Alternativní metody obousměrné síťové komunikace WebSocket by měl poskytnout náhradu za doposud používané technologie HTTP *polling, long-polling a streaming*. Všechny tyto metody zahrnují posílání HTTP položek (headers), které obsahují mnoho zbytečných dat, čímž dochází ke zvýšení latence nad minimální úroveň. WebSocket je navržen tak, aby mohl zužitkovat existující infrastrukturu (proxy servery, ověřování, atd.)

polling Při *pollingu* klient v pravidelných intervalech posílá serveru HTTP požadavky a okamžitě přijímá odpověď. Tato technika je používána pro zisk dat z real-time aplikací. Poskytuje dobré řešení, pokud je známý interval, po jehož uplynutí jsou na serveru poskytována nová data. Bohužel tento interval je často nepravidelný a nepředvídatelný. Proto dochází k otevírání zbytečných HTTP připojení, obzvláště pokud nové informace nejsou dostupné příliš často.

long-polling Při *long-pollingu* prohlížeč posílá serveru požadavek a ten ho nechá otevřený tak dlouho, než bude mít pro klienta nějaká data. Pokud se tak ve zvoleném vymezeném čase nestane, server pošle odpověď pro ukončení připojení. Z principu *long-polling* vyplývá, že nepřináší žádné zlepšení vůči *polling* v případě, že se data na serveru často obnovují.

streaming Další z možných přístupů je *streaming*, při kterém prohlížeč pošle serveru požadavek a server mu jako odpověď průběžně posílá data. Nemusí tedy čekat na vygenerování kompletní odpovědi, aby ji mohl poslat najednou. *Streaming* nesnižuje latenci připojení, jeho výhodou ale je, že klient může na příchozí data reagovat už v průběhu přijímání dat. *Streaming* je ale stále založený na technologii HTTP, což může způsobit, že firewally a proxy servery si mohou zvolit, že budou data bufferovat, což způsobí zvýšení latence doručení zprávy. Navíc server je nucen používat několik různých TCP připojení pro každého klienta: jeden pro posílání informací a další pro každou nově příchozí zprávu. WebSocket oproti tomu umožňuje pro komunikaci v obou směrech využít pouze jedno TCP připojení [web(2011)].

2.3.2 Principy komunikace

Handshake Termín *handshake* v komunikačních technologiích označuje proces, při kterém mezi účastníky komunikace dochází k dohodnutí se na hodnotách dynamických parametrů před vlastním započetím komunikace. Možná podoba *handshake* klienta a serveru pro technologii WebSocket je uvedena v Příloze 1.

Handshake v protokolu *WebSocket* má tvar HTTP *Upgrade* požadavku. Díky tomu je tato technologie kompatibilní s HTTP servery a s prostředníky na HTTP bázi. Dalším důsledkem je možnost využití jednoho portu pro HTTP i WebSocket klienty.

Klient ve svém *handshake* specifikuje požadovanou adresu URI[11], ke které se chce připojit. Tento způsob umožňuje, aby z jedné IP adresy mohlo být obslouženo více domén a aby více WebSocket klientů mohlo být obslouženo jediným serverem[Ian Fette(2011)]. Pro URI Websocketu jsou definována dvě schémata. Pro klasické Websocket připojení má URI prefix *ws://* a pro zabezpečenou verzi se používá prefix *wss://*.

Kromě požadované URI klientský *handshake* obsahuje i další položky pro upřesnění způsobu následné komunikace.

- *Sec-WebSocket-Protocol* - určení aplikačního protokolu, který značí formát, v jakém budou data přenášena
- *Sec-WebSocket-Extensions* - seznam žádaných rozšíření protokolu(např. komprese dat atd.)
- *Origin* - položka sloužící k filtrování webových stránek, ze kterých má být server přístupný
- *Sec-WebSocket-Version* - verze protokolu

Po odeslání svého handshake klient musí počkat na odpovídající handshake serveru, dříve než začne posílat další data.

Server si z klientova *handshake* musí vyjmout položku *Sec-WebSocket-Key* v níž se nachází náhodný klientem vybraný řetězec a upraví ho podle stanovených pravidel. Tak získá řetězec, který poté odešle ve svém *handshake* v položce *Sec-WebSocket-Accept*. Tímto způsobem si klient ověřuje, že přijatý *handshake* pochází od jím zvoleného serveru. Jakmile klient přijme *handshake* serveru, mohou si obě strany komunikačního kanálu nezávisle na sobě posílat data.

[11] Uniform Resource Identifier

Data Framing Po úspěšném *handshake* si mohou klient se serverem vyměňovat data. Tato data jsou sdružována do celků označovaných jako *"messages"*, které jsou dále rozdělené do jednotlivých rámců neboli *framů*. V Příloze 2 je zobrazen základní tvar pro jeden rámec. Vysvětlení jednotlivých částí rámce je v Příloze 3.

Hlavní výhoda fragmentování zpráv spočívá v tom, že server může začít posílat fragmenty zprávy ještě před tím, než má celou zprávu k dispozici. Pokud by zpráva nebyla fragmentována, odesílatel by musel počkat až bude mít celou zprávu a spočítat její velikost dříve, než by mohl být odeslán první byte. Díky fragmentaci může server nebo prostředník zvolit rozumnou velikost bufferu a jakmile se buffer naplní rámec odeslat.

Ukončení spojení Po úspěšném otevíracím handshake a přenosu potřebných dat spojení ukončuje uzavírací handshake. Kterýkoliv z koncových bodů může vyvolat uzavírací handshake tím, že odešle svému protějšku uzavírací rámec, jehož tvar lze odvodit z Přílohy 3. Po odeslání tohoto rámce již nesmí pokračovat v odesílání žádných dat a vyčkává na odezvu také v podobně uzavíracího rámce. Po přijetí odpovědi iniciátor ukončení komunikace ukončí spojení.

2.3.3 WebSocket API

HTML5 WebSocket specifikace definuje jednoduché API, které umožňuje webovým stránkám použít WebSocket protokol pro komunikaci se vzdáleným hostitelem. Kompletní podoba rozhraní v sémantice WebIDL[12] lze najít v Příloze 4.

Konstruktor Konstruktor *WebSocket(url, protocols)* má jeden nebo dva argumenty. První z nich, *url*, určuje URL adresu, ke které se má WebSocket připojit. Parametr *protocols* je nepovinný a je typu string nebo pole stringů. Tyto řetězce mají význam subprotokolů a jsou obsaženy v klientském handshake v položce *Sec-WebSocket-Protocol*[Hickson(2012)].

Úkolem konstruktoru je na základě předaných parametrů vytvořit nové připojení k serveru a vrátit nový WebSocket objekt.

[12] *WebIDL* - jazyk zaměřený na popis rozhraní technologií určených pro webové prohlížeče

Atribut *readyState* reprezentuje stav spojení a může nabívat následujících hodnot:

- CONNECTING(číselná hodnota 0) - připojení ještě nebylo vytvořeno
- OPEN(číselná hodnota 1) - připojení je vytvořeno a posílání dat je možné
- CLOSING(číselná hodnota 2) - připojení je ve stavu probíhajícího closing handshake
- CLOSED(číselná hodnota 3) - připojení je uzavřené a nebo nemohlo být otevřeno

Atribut *bufferedAmount* Atribut *bufferedAmount* obsahuje počet bytů, které byly přidány do fronty použitím metody *send* a nebyly ještě odeslány.

Metoda *send(data)* Tato metoda posílá data prostřednictvím vytvořeného připojení.

Metoda *close()* Tato metoda spouští proceduru uzavíracího *handshake* a ukončuje spojení se serverem. Volitelným parametrem je integer s kódem chyby, která způsobila ukončení spojení.

V následující tabulce jsou event handlery[13], které obsahuje WebSocket rozhraní. Tyto handlery jsou zpravidla přepisovány uživatelem, který rozhraní využívá.

Event handler	událost, při které je handler spuštěn
onopen()	úspěšné vytvoření připojení
onmessage()	přijetí nové zprávy
onerror()	příjetí rámce s chybovým hlášením
onclose()	přijetí uzavíracího rámce

[13]metody pro zpracování vzniklých asynchronních událostí

2.4 JSON

V sekci 2.3 je popsána technologie WebSocket využívaná pro přenos dat ve tvaru textových řetězců. Ve směru od prohlížeče tyto řetězce obsahují jen jednoduché příkazy pro OPC .NET API. V opačném směru je ale nutné posílat data reprezentující výsledky vykonání těchto příkazů. Pro reprezentaci těchto dat byl vybrán protokol JSON.

2.4.1 Protokol JSON

„JSON (JavaScript Object Notation) je odlehčený formát pro výměnu dat a serializaci objektů[14]. Je jednoduše čitelný i zapisovatelný člověkem a snadno analyzovatelný i generovatelný strojově“ [jso()]. JSON je vytvořen na základech jednoho z možných způsobu vytváření objektů v programovacím jazyce JavaScript. J e textový formát, který, přestože vychází z JavaScriptu, je na použitém jazyce zcela nezávislý.

JSON podporuje dvě struktury:

- Struktura reprezentující ***objekt***. Objekt je brán jako množina párů obsahující název členu objektu a jeho hodnoty. Při deserializaci objektů v JSON formátu do jiných jazyků než JavaScriptu se obvykle využívají struktury typu objekt, záznam, struktura, slovník, hash tabulka a další.
- Struktura reprezentující ***pole***.

Obě struktury se vyskytují v nějaké formě v podstatě ve všech programovacích jazycích a proto je JSON podporovaný většinou programovacích jazyků a značně rozšířený. Kromě těchto struktur se v JSON mohou objevit ještě textové řetězce, číselné hodnoty, hodnoty *boolean* a hodnota *null*.

[14]serializace objektu - zpracování a uložení instance do perzistentního uložiště

2.4.2 Příklady JSON formátu

- Příklad zápisu jednoduchého pole číselných hodnot

```
[ 10, 5, 98, 20, 30 ]
```

- Příklad zápisu jednoduchého objektu

```
{
    color: "red",
    value: "#f00"
    array: [ 10, 5, 98, 20, 30 ]
}
```

- Příklad zápisu pole objektů

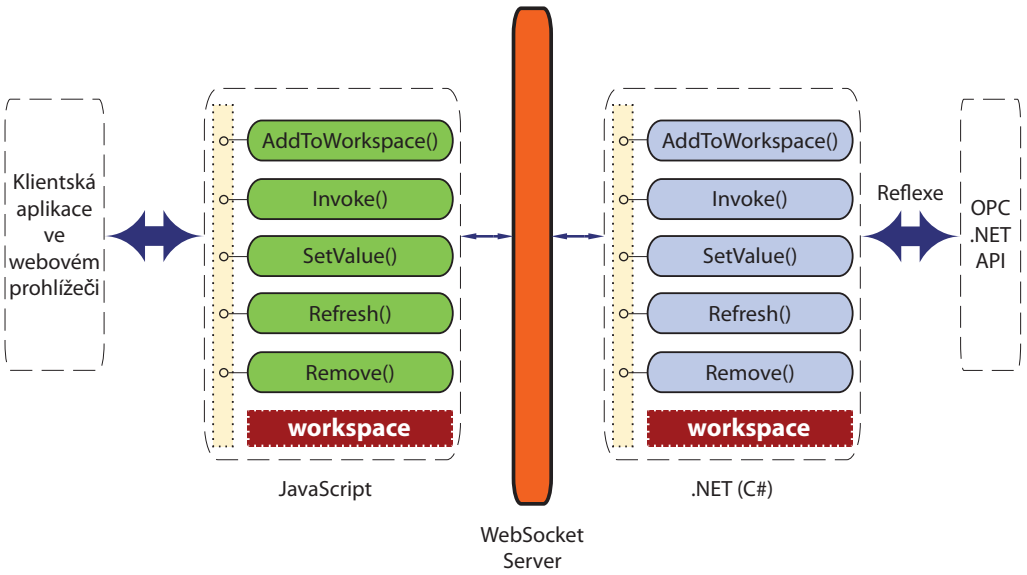
```
[
    {
        color: "red",
        value: "#f00"
    },
    {
        color: "green",
        value: "#0f0"
    },
    {
        color: "blue",
        value: "#00f"
    }
]
```

3 Implementace

3.1 Schéma aplikace

Aplikace funguje jako most propojující klienta v JavaScriptu a prostředí OPC Serverů, ke kterým se přistupuje prostřednictvím OPC .NET API. Mezi JavaScriptem a OPC Servery tedy musí existovat mezičlánek v .NET, který bude zpracovávat příkazy z JavaScriptu a podle nich manipulovat s rozhraním OPC serverů. Z podporovaných programovacích jazyků prostředí .NET byl vybrán C#, který byl vytvořen společností Microsoft právě pro účely .NET Frameworku.

Na obrázku 3.1 je nastíněno základní schéma aplikace. V JavaScriptu je vytvořen objekt s určitým rozhraním, které dále využívá klient z webového prohlížeče. Hlavním úkolem tohoto objektu je posílat zprávy s příkazy prostřednictvím WebSocket serveru do prostředí .NET, kde jsou tyto příkazy vykonány. Rozhraní v JavaScriptu a v .NET jsou stejná, protože ke každému příkazu je v JavaScriptu metoda , která tento příkaz odešle, a v .NET metoda, která tento příkaz vykoná. Příkazy obsahují pokyny pro manipulaci s OPC .NET API, která probíhá dynamicky na základě reflexe (viz. sekce 2.2.2).



Obrázek 3.1: Schéma aplikace.

3.2 Proměnná *workspace*

Jak již bylo řečeno, aplikace má podobu pomyslného mostu mezi klientem v JavaScriptu a OPC .NET API, se kterým manipuluje jakýsi mezičlánek implementovaný v jazyce C#. Základem funkce aplikace je proměnná *workspace*, která se nachází v JavaScriptové části a v části aplikace implementované v C#(viz. obr.3.1). Na straně JavaScriptu se jedná o datový typ *objekt*. Objekty v JavaScriptu jsou v podstatě množiny párů klíč|hodnota, kde klíče jsou řetězce a hodnota může být jakýkoliv datový typ. Na straně C# je tato struktura reprezentována slovníkem *Dictionary<string,object>* ze jmenného prostoru *System.Collections.Generic*. Klíče v tomto slovníku jsou rovněž řetězce a hodnotami, které odpovídají klíčům, mohou být libovolné objekty.

V proměnné *workspace* se uchovávají veškeré uživatelem vytvářené objekty. Uživatel v JavaScriptu zavolá metodu pro vytvoření objektu, ten se na straně C# vytvoří, uloží se do proměnné *workspace* v C# a zpětně se zrcadlí do proměnné *workspace* na straně JavaScriptu. Navíc se ke každému objektu ve *workspace* v JavaScriptu přidává položka *dotnettype*, která obsahuje řetězec s datovým typem, jenž odpovídá typu objektu v C#, podle kterého tento obraz vznikl. Tato položka je zavedena z důvodu absence typové kontroly v JavaScriptu při volání metod (více o volání metod později v sekci 3.4). Zrcadlení objektů probíhá výměnnou dat ve formátu JSON(viz. 2.4) přes WebSocket server(viz. 2.3).

3.3 Rozhraní aplikace

Při návrhu aplikace bylo cílem vytvořit co nejjednodušší rozhraní, které umožní snadno vytvářet objekty a vyvolávat jejich metody. Velmi obecné řešení využívající reflexi(viz. 2.2) na straně C# umožňuje, aby základní rozhraní objektu v JavaScriptu mělo pouze 5 metod.

Téměř každá metoda rozhraní aplikace v JavaScriptu má jako první argument položku *callback*, která označuje název funkce v JavaScriptu, která bude vyvolána po vyplnění původní funkce. Toto opatření je nutné, protože JavaScript je pouze jednovláknový jazyk. To znamená, že pokud by se v průběhu skriptu muselo čekat na dokončení nějaké časově náročné operace, vyřadilo by to všechny komponenty webové stránky, které jsou na JavaScriptu závislé. Z tohoto důvodu musí všechny časově náročné operace probíhat asynchronně. V našem případě je asynchronní přístup obzvláště nutný

při vytváření objektu a následném volání metody tohoto objektu, která musí být spuštěna až po úspěšném zrcadlení nového objektu do JavaScriptu, tedy v *callbacku*.

Aplikace podporuje vytváření objektů jednoduchých datových typů ve funkcích primárně určených pro jiné účely, než je vytváření objektů. Například pokud v parametru vyvolávané metody bude řetězec "int:5", vytvoří se dočasný objekt typu *System.Int32* s hodnotou 5 a tento objekt se předá vyvolávané metodě jako parametr. Tato vlastnost je vždy uvedena v popisu parametrů dané funkce.

Další vlastnost, kterou rozhraní má, je možnost potlačení serializace funkcí (více o serializaci funkcí v sekci 3.4). Serializace je časově náročná operace a tímto způsobem je možné ji částečně urychlit. Například není potřeba serializovat metody u čtených dat ze serveru, jejichž metody nehodláme použít. Serializace metod se potlačí tak, že v daném řetězci předávaném metodě bude první znak roven `'*'`. Například chceme-li nový objekt uložit pod jménem *"dummy"* a chceme-li potlačit serializaci funkcí u tohoto objektu, předáme metodě parametr s hodnotou *"*dummy"*. U parametrů, kterých se tato vlastnost týká je to vždy uvedeno dále v popisu rozhraní.

Základní rozhraní aplikace v JavaScriptu: Tyto metody vysílají prostřednictvím WebSocket serveru požadavky na vyvolání stejnojmenné metody v mezičlánku aplikace implementované v C#.

- **AddToWorkpace(***callback, itemName, itemType***)**
  - Přidává nový objekt do *workspace* zavoláním vhodného konstruktoru, vybíraného podle daných vstupních parametrů. Při chybném zvolení parametrů metoda vyvolá výjimku s popisem možných kombinací parametrů.
  - Popis parametrů:
    - *callback* - jméno funkce, která bude spuštěna po úspěšném vytvoření objektu.
    - *itemName* - Jméno, pod kterým bude objekt uložen v proměnné *workspace*. Může obsahovat požadavek na potlačení serializace metod.
    - *itemType* - Jméno datového typu objektu. Může obsahovat deklaraci jednoduchých datových typů. Také může obsahovat deklaraci generických datových typů (např.: *"System.Collections.Generic.Dictionary<string,int>"*)

    - Metodu je možné volat s proměnlivým počtem parametrů. Všechny další parametry jsou považovány za jména parametrů ve *workspace*, se kterými má být volán konstruktor.
  - Příklad volání:

    ```
    opcClient.AddToWorkspace("connectRexServer", serverName,
    "Opc.Da.Server", "Factory", "URL");
    ```

  - Příklad protokolu dat přenášených přes WebSocket server:

    ```
    "connectRexServer_|_AddToWorkspace~rexServer;
    Opc.Da.Server;Factory,URL"
    ```

- **Invoke**(*callback, itemName, methodName, saveResultName, argsArray*)
  - Vyvolává zvolenou metodu nad vybraným objektem s požadovanými argumenty. Pokud se uživatel pokouší vyvolat metodu se špatnými argumenty, metoda vyvolá výjimku s informacemi o možných způsobech volání dané metody.
  - Popis parametrů:
    - *callback* - jméno funkce, která bude spuštěna po úspěšném vyvolání metody a které bude předán obraz návratové hodnoty vyvolané metody.
    - *itemName* - Jméno objektu ve *workspace*, nad kterým se bude vyvolávat metoda.
    - *methodName* - Jméno metody, která má být vyvolána.
    - *saveResultName* - Jméno, pod jakým se do *workspace* uloží návratová hodnota metody. Může obsahovat požadavek na potlačení serializace metod.
    - *argsArray* - pole jmen parametrů, které mají být metodě předány. Může obsahovat deklaraci jednoduchých datových typů.
  - Příklad volání:

    ```
    opcClient.Invoke("callbackFce","rexServer","Connect",
    null,["ConnectData"]);
    ```

  - Příklad protokolu dat přenášených přes WebSocket server:

    ```
    "callbackFce_|_Invoke~rexServer;Connect;
    null;ConnectData"
    ```

- **SetValue**(*callback, itemName, valueName*)
  - Metoda nastavuje objekt ve *wokrspace* na určenou hodnotu.

  - Popis parametrů:
    * *callback* - Jméno metody, která bude vyvolána po úspěšném nastavení objektu na požadovanou hodnotu.
    * *itemName* - Jméno objektu, jehož hodnota má být přenastavována.
    * *valueName* - Jméno objektu ve *workspace*, jehož hodnota se použije pro nastavení objektu. Může obsahovat deklaraci jednoduchých datových typů.
  - Příklad volání:

    ```
    opcClient.SetValue(null, "URL.Scheme", "string:opcda");
    ```

  - Příklad protokolu dat přenášených přes WebSocket server:

    ```
    "null_|_SetValue~URL.Scheme;string:opcda"
    ```

- **Refresh**(*callback, itemName, boolWithFunctions*)
  - Aktualizuje objekt ve *workspace* v JavaScriptu. Aktualizace je nutná, pokud se hodnoty členů objektu mění po vytvoření objektu jiným způsobem než prostřednictvím metody *SetValue()*. Dalším možným využitím je dodatečná serializace metod, pokud jsme při vytvoření objektu tuto možnost potlačili.
  - Popis parametrů:
    * *callback* - Jméno funkce, která bude spuštěna po úspěšné aktualizaci objektu.
    * *itemName* - Jméno objektu, který má být aktualizován.
    * *boolWithFunctions* - Hodnota typu *boolean* určující, zda se má objekt aktualizovat včetně funkcí. To může být nutné, pokud objekt s daným jménem přepíšeme ve *workspace* jiným objektem.
  - Příklad volání:

    ```
    opcClient.Refresh(null,"group.ReadResults",false);
    ```

  - Příklad protokolu dat přenášených přes WebSocket server:

    ```
    "null_|_Refresh~group.ReadResults;false"
    ```

- **Remove**(*callback, itemName*)
  - Odstraňuje objekty z proměnné *workspace*.

  - Popis parametrů:
    * *callback* - Jméno funkce, která bude spuštěna po odstranění daného objektu.
    * *itemName* - Jméno objektu, který má být odstraněn.
  - Příklad volání:

    ```
    opcClient.Remove(null, "URL");
    ```

  - Příklad protokolu dat přenášených přes WebSocket server:

    ```
    "null_|_Remove~URL"
    ```

Specifické metody rozhraní: Tyto metody jsou specifické pro JavaScriptovou část aplikace a nemají svého protějška v mezičlánku implementovaném v C#.

- **Connect**(*host*)
  - Metoda pro připojení objektu v JavaScriptu k WebSocket serveru.
  - Popis parametrů:
    * *host* - URI adresa serveru, ke kterému se má objekt připojit
- **OnConnectedToWS**()
  - Metoda, která je spuštěna po úspěšném připojení k WebSocket serveru. Předpokládá se, že klient tuto metodu přepíše svým kódem.
- **ConnectToREX**(*callback, serverName*)
  - Metoda speciálně určená pro připojení k OPC serveru v řídícím systému *REX*. Využívá k tomu výše zmíněné metody pro vytváření objektů, nastavení hodnoty a volání funkce.
  - Popis parametrů:
    * *callback* - Jméno funkce, která bude spuštěna po připojení OPC .NET API k OPC serveru řídícího systému *REX*.
    * *serverName* - Jméno, pod kterým bude server uložen ve *wokspace*.
- **getItem**(*name*)
  - Metoda pro získání položky z proměnné *workspace*.

  - Popis parametrů:
    - *name* - jméno položky nebo řetězec obsahující cestu k položce ve *workspace*

- **getWorkspace()**
  - Metoda, která vrací proměnnou *workspace*.

- **SubscriptionCallback(***itemsArray***)**
  - Metoda, která slouží pro zpracování asynchronně čtených dat objektem *OpcSimpleIO* (viz. sekce 3.5). Předpokládá se, že uživatel přepíše metodu svým kódem.
  - Popis parametrů:
    - *itemsArray* - pole, které obsahuje asynchronně čtená data

3.4 Datový formát JSON a serializace funkcí

Rozhraní definované v sekci 3.3 je dostatečné pro základní komunikaci s libovolným rozhraním v .NET. Na druhou stranu není příliš uživatelsky přátelské, neboť uživatel musí rozhraní v .NET znát do té míry, aby věděl jaká metoda se volá s jakými parametry.

Pro vyřešení tohoto problému byl rozšířen datový formát JSON tak, aby kromě textových řetězců, číselných hodnot, boolean hodnot a hodnoty null umožňoval navíc použití funkcí jako hodnoty v páru *klíč:hodnota* (více o JSON v sekci 2.4).

Rozšíření JSON formátu umožňuje zrcadlení objektu ze C# do JavaScriptu včetně vlastních veřejných metod . Samozřejmě tyto obrazy metod neprovádí stejné úkony, ale pouze ve svém nitru volají metodu *Invoke()* s příslušnými parametry. Tento způsob navíc umožňuje provádět typovou kontrolu parametrů už v JavaScriptu před zavoláním metody *Invoke()* a tím zabraňuje nadbytečné síťové komunikaci, která by nastala v případě, kdy by se uživatel pokoušel zavolat metodu s nesprávnými parametry přímo prostřednictvím metody *Invoke()* a o chybném zvolení parametrů by se poté dozvěděl až po vyvolání výjimky v C# a její propagaci do JavaScriptu.

Serializace funkcí umožňuje aby uživatel přistupoval k zvolenému API v .NET pomocí obrazů funkcí samotného API a manipulovat tak s ním v JavaScriptu téměř stejně, jako by sním manipuloval v prostředí .NET.

Příklady serializace metod

Hlavičky metod v C#:

```
public void bar(double d, object obj);
public double foo(int i);
public int foo(double d, string str);
```

Obrazy funkcí v JavaScriptu: Obraz ukázkového objektu je ve *workspace* uložen pod jménem "dummy". U přetížených metod se nejprve kontroluje počet parametrů obsažených v poli `opcB_argsArray`. Poté se kontrolují datové typy jednotlivých parametrů. Parametry mohou být buď `null`, nebo to mohou být řetězce obsahující deklaraci nového objektu jednoduchého datového typu (např.: "*int:5*"), nebo je parametrem jméno objektu, pod kterým je uložen ve *workspace*. V tom případě se objekt vyhledá ve *wokspace* a zkontroluje se jeho člen `.dotnetType`. Výjimku tvoří případ, kdy má být požadovaný argument typu *object*. V tom případě může být parametrem hodnota `null` nebo libovolná deklarace jednoduchého typu a nebo jméno libovolného objektu, který je definovaný ve *workspace*. Pokud jsou všechny parametry úspěšně zkontrolovány, je zavolána funkce `Invoke()`. Pokud se typová kontrola nevydaří, dojde k vyvolání výjimky.

Obraz metody *bar()*

```
function (js_callback, d, obj) {
   if (d === null || d.split(':')[0] === "double" ||
    opcClient.getItem(d).dotnetType === "double") {
      if (obj === null || obj.split(':').length === 2 ||
       opcClient.getItem(obj) !== 'undefined') {
         opcClient.Invoke(js_callback, "dummy", "bar",
          "null", [d, obj]);
         return;
      }
   }
   throw "There is no 'bar' method callable with given arguments!";
}
```

Obraz přetížené metody *foo()*

```
function (js_callback, opcB_saveResultName, opcB_argsArray) {
   if (opcB_argsArray.length === 1) {
      if (opcB_argsArray[0] === null ||
       opcB_argsArray[0].split(':')[0] === "int" ||
       opcClient.getItem(opcB_argsArray[0]).dotnetType === "int") {
         opcClient.Invoke(js_callback, "dummy", "foo",
          opcB_saveResultName, opcB_argsArray);
         return;
      }
   }
   if (opcB_argsArray.length === 2) {
      if (opcB_argsArray[0] === null ||
       opcB_argsArray[0].split(':')[0] === "double" ||
       opcClient.getItem(opcB_argsArray[0]).dotnetType === "double") {
         if (opcB_argsArray[1] === null ||
          opcB_argsArray[1].split(':')[0] === "string" ||
          opcClient.getItem(opcB_argsArray[1]).dotnetType === "string") {
            opcClient.Invoke(js_callback, "dummy", "foo",
             opcB_saveResultName, opcB_argsArray);
            return;
         }
      }
   }
   throw "There is no 'foo' method callable with given arguments!";
}
```

3.5 Zjednodušené API pro přístup k OPC

Díky serializaci funkcí nemusí uživatel používat metodu `Invoke()` ze základního rozhraní aplikace, ale může volat požadované metody v C# prostřednictvím jejich obrazů v JavaScriptu. Může tak využívat API, ke kterému se aplikace připojuje, v jeho původní podobě. Další možností, která se uživateli nabízí, je vytvořit si v C# objekt, který bude požadované API libovolně zaobalovat a nadále se z JavaScriptu připojovat přímo k tomuto objektu. Tímto způsobem si uživatel může rozhraní, se kterým bude nadále pracovat, libovolně přizpůsobovat podle svých vlastních požadavků.

Pro přístup k OPC DataAccess Serveru (jehož implementaci obsahuje řídící systém REX) byl vytvořen jednoduchý objekt, který si klade za cíl ulehčovat základní operace zápis a čtení dat včetně asynchronního čtení.

Zjednodušené rozhraní - objekt *OpcSimpleIO*

- **OpcSimpleIO**(Opc.Da.Server *server*)
  - Konstruktor objektu.
  - parametr *server* - připojený server typu Opc DA. Pro vytvoření Opc serveru systému REX stačí zavolat metodu `ConnectToREX()` z rozhraní aplikace v JavaScriptu .
- **Add_R_Item**(string *itemName*)
  - Metoda přidává položku pro čtení.
  - parametr *itemName* - jméno položky, která má být přidána
- **Remove_R_Item**(string *itemName*)
  - Metoda odebírá položku pro čtení.
  - parametr *itemName* - jméno položky, která má být odebrána
- **Add_W_Item**(string *itemName*, object *value*)
  - Metoda přidává položku pro zápis.
  - parametry:
    - *itemName* - jméno položky, která má být přidána
    - *value* - hodnota, která má být zapsána na server

- **Remove_W_Item(**string *itemName***)**
  - Metoda odebírá položku pro zápis.
  - parametr *itemName* - jméno položky, která má být odebrána
- **Read()**
  - Metoda přečte položky přidané metodou Add_R_Item().
- **Write()**
  - Metoda zapisuje položky přidané metodou Add_W_Item().
- **ChangeState(**string *propertyName*, object *value***)**
  - Metoda mění hodnoty podmínek pro asynchronní čtení.
  - parametry:
    - *propertyName* - jméno vlastnosti, jejíž hodnota má být změněna
      Některé možné hodnoty:
      - *Active* - aktivace asynchronního čtení
      - *UpdateRate* - perioda obnovování dat
      - *KeepAlive* - průběžná aktualizace hodnot pro udržení spojení
      - *Deadband* - určuje pásmo necitlivosti na změny dat
      - *SamplingRate* - frekvence čtení dat
    - *value* - hodnota, kterou má daná vlastnost nabývat
- **SwitchSubscriptionState()**
  - Metoda aktivuje nebo deaktivuje asynchronní čtení.

3.6 Zpětné odezvy a chybové hlášky

Aplikace je navržena tak, aby jakýkoliv požadavek z JavaScriptové strany vždy vyvolal odpověď nebo chybovou hlášku na straně C#, která je poté poslána zpět. Z důvodu snadného rozlišení odezev bylo každé přiděleno identifikační číslo, které se přidává ke zprávě s odezvou.

Identifikační čísla odezev:

100x	Množina odezev potvrzujících korektní průběh operací
1001	úspěšně přidán nový objekt do proměnné *workspace*
1002	úspěšné nastavení objektu na požadovanou hodnotu
1003	úspěšné vyvolání metody
1004	úspěšné odebrání objektu z *workspace*
1005	asynchronní upozornění na změnu hodnoty prvku OPC serveru
1006	úspěšné obnovení objektu
200x	Množina běžných chyb
2001	neznámý příkaz
2002	neočekávaná výjimka
2003	pokus o přístup k objetu, který není ve *workspace* definovaný
2004	datový typ objektu nemohl být identifikován
2005	chybná deklarace generického datového typu
2006	chybná deklarace jednoduchého datového typu
300x	Chyby vzniklé při vytváření nového objektu
3001	chybný tvar odeslaného požadavku
3002	snaha o zavolání neexistujícího konstruktoru
400x	Chyby vzniklé při nastavování objektu na danou hodnotu
4001	chybný tvar odeslaného požadavku
4002	snaha o nastavení objektu na neexistující objekt
4003	pokus o nastavení objektu na hodnotu jiného datového typu
500x	Chyby vzniklé při vyvolávání metod
5001	chybný tvar odeslaného požadavku
5002	volání neexistující metody
5003	volání metody se špatnými parametry

4 Testování výkonu aplikace

4.1 Reflexe vs. přímý přístup

Reflexe musí být z principu své funkce (viz. sekce 2.2.2) znatelně pomalejší než přímý přístup. Otázkou je ale o kolik je pomalejší a do jaké míry to ovlivňuje výkon aplikace.

V tabulce 4.1 jsou výsledky testování rychlosti volání metody. Testovací metoda neprováděla žádné operace, jen vracela svůj argument, kterým byl integer. Metoda byla volána 10^6-krát. Z výsledků je patrné že samotné vyvolání metody reflexí je přibližně 100-krát pomalejší, přímé zavolání.

V tabulkách 4.2 a 4.3 jsou výsledky porovnávání reflexivního a přímého přístupu při volání metod používaných pro zisk dat z OPC Serveru. Při prvním měření byla volána metoda `Add_R_Item(string name)` pro přidání prvku do skupiny pro čtená data, v druhém případě byla zvolena metoda `Read()`, která vyčítá data ze serveru. Metoda `Read()` byla vždy spouštěna 1000-krát s různými počty čtených prvků. Metoda `Add_R_Item()` byla spuštěna tolikrát, kolik prvků bylo přidáváno.

V částech tabulek *Přírůstek* je v procentech vyčísleno o kolik je průběh metody včetně zavolání delší při použití metody *Invoke()* využívající reflexi než při přímém zavolání. Metoda *Invoke()* je vždy o něco pomalejší, ale maximální naměřený rozdíl o hodnotě navýšení rovné 32% není nijak závažný. Toto zpoždění navíc není způsobeno pouze reflexí, neboť metoda *Invoke()* provádí i další operace jako například manipulaci s proměnnou *workspace* nebo zisk hodnoty parametrů z řetězců obsahujících deklaraci jednoduchých datových typů a další.

Ze získaných výsledků se dá vyvodit, že opoždění při zavolání metody reflexí je takřka zanedbatelné. Průběh a rychlost metod daleko více závisí na operacích, které metoda provádí, než na způsobu zavolání metody.

doba trvání $[s]$	
Reflexe	Přímé volání
1.5109	0.0184

Tabulka 4.1: Výsledky testování rychlosti reflexe.

Počet přidaných prvků	doba trvání [s] Reflexe	Přímé volání	Přírůstek [%]
10	0.148	0.121	22
50	0.598	0.493	21
100	1.196	0.911	31
500	5.848	4.401	32
1000	11.159	8.848	26

Tabulka 4.2: Výsledky testování rychlosti volání metody `Add_R_Item()`.

Počet čtených prvků	doba trvání [s] Reflexe	Přímé volání	Přírůstek [%]
10	0.411	0.347	18
50	1.109	1.079	3
100	1.942	1.939	<1
500	8.156	7.615	7
1000	17.185	15.106	14

Tabulka 4.3: Výsledky testování rychlosti volání metody `Read()`.

4.2 Přenos dat

Dalším faktorem, který ovlivňuje výkonnost aplikace je přenos dat přes WebSocket server (viz. sekce 2.3). Objem dat přenášených z prohlížeče na server je velmi malý, neboť obsahuje jen příkazy, které se mají na serveru vykonat. Naopak řetězce obsahující serializované objekty, které jsou posílány ze serveru prohlížeči, mohou být velice rozsáhlé.

Rychlost přenosu byla testována po lokální síti. V reálných podmínkách bude samozřejmě výkonnost záviset na rychlosti připojení. Testována byla rychlost přenášení serializovaných objektů pro různé počty objektů. Přenos dat byl prováděn 1000-krát. Výsledky jsou zanesené v tabulce 4.4.

4.3 Serializace a deserializace objektů

Další časově náročnou operací v běhu aplikace je nutnost serializace objektů do formátu JSON v C# a následná deserializace v JavaScriptu. Při

Počet přenášených prvků	doba trvání $[s]$
10	0.259
50	1.359
100	3.275
500	25.741
1000	80.838

Tabulka 4.4: Výsledky testování rychlosti přenášení dat ze serveru.

testování byla serializace i deserializace provedena 1000krát. Pro serializaci nemohla být využita žádná knihovní funkce jazyka C#, protože aplikace využívá specificky upravený formát JSON rozšířený o funkce (viz. sekce 3.4). Proto byl implementován vlastní objekt pro serializaci, který využívá reflexi.

Pro deserializaci je využívána funkce `eval()`, která je přirozenou součástí JavaScriptu. Používání této funkce není obecně příliš doporučováno, protože sebou nese jistá bezpečnostní rizika. V našem případě je ale tato funkce používána jen pro zpracování řetězců z vlastního serveru, takže k žádnému ohrožení dojít nemůže. Nemohli být využity žádné moduly pro JavaScript, které zajišťují zpracování JSON formátu kvůli obsahu funkcí, což tyto moduly nepodporují.
Výsledky měření lze nalézt v tabulce 4.5.

	doba trvání $[s]$	
Počet prvků	Serializace	Deserializace
10	0.662	0.160
50	3.392	0.834
100	7.499	1.585
500	33.191	8.054
1000	71.897	17.566

Tabulka 4.5: Výsledky testování rychlosti serializace a deserializace objektů

4.4 Celkový výkon aplikace

Lze předpokládat, že aplikace najde největší využití při vyčítání dat z OPC Serveru. Z toho důvodu byl celkový výkon aplikace testován na metodě `Read()`. Ze serveru byla 1000-krát čtena skupina dat a měřena byla doba od prvního odeslání požadavku na čtení dat z JavaScriptu až po poslední příjem dat. Testování bylo provedeno po lokální síti. Výsledky jsou v tabulce 4.6. V části frekvence čtení je přibližná hodnota v herzích, která odpovídá frekvenci s jakou je aplikace schopna číst dané množství prvků. Pro dosažení lepších výsledků je možné využít asynchronní čtení (viz. sekce 3.5), kde odpadá neustálé posílání příkazů z JavaScriptu a následné volání metody pro čtení.

Počet čtených prvků	doba trvání $[s]$	Frekvence čtení $[Hz]$
10	1.893	528
50	6.149	162
100	11.939	83
500	68.992	14
1000	167.257	6

Tabulka 4.6: Výsledky testování rychlosti čtení dat z OPC serveru.

5 Ukázka

Pro ukázku funkce aplikace byla vytvořena jednoduchá *SVG* grafika (viz. obr. 5.1) ve webovém prohlížeči v podobě hodin, která je řízena daty přijímanými prostřednictvím vytvořené aplikace z řídícího systému REX Control. Schéma zapojení algoritmu v systému REX lze nalézt v Příloze 5. Algoritmus probíhá s periodou $1s$.

Popis významných částí kódu webové stránky:

Inicializace připojení - V prvním kroku, který je nutné provést před připojením k OPC Serveru, se překrývá metoda `OnConnectedToWS()`, která se vykoná po úspěšném připojení k WebSocket serveru, a vytvoří se připojení. Dalším krokem je vytvoření připojení k OPC serveru systému REX metodou `ConnectToREX()`. Po úspěšném připojení se vyvolá metoda `addGroup()`, ve které se vytváří objekt `OpcSimpleIO` konstruktorem, který přijímá OPC server jako svůj argument.

```
opcClient.Connect("ws://localhost:8181/");
opcClient.OnConnectedToWS = function () {
   opcClient.ConnectToREX("opcClient.addGroup", "rexServer");
}
opcClient.addGroup = function () {
   opcClient.AddToWorkspace("opcClient.initIO", "group",
    "OPCBridge.OpcSimpleIO", "rexServer");
}
```

Inicializace čtení a zápisu dat - Použitím metod `Add_R_Item()`, respektive `Add_W_Item()`, se inicializuje čtení, respektive zápis, položek na server. Zapisují se data, která odpovídají aktuálnímu času.

```
opcClient.initIO = function () {
   var date = new Date();
   var group = opcClient.getItem("group");
   group.Add_R_Item(null, "string:rex_clock.Seconds.y");
```

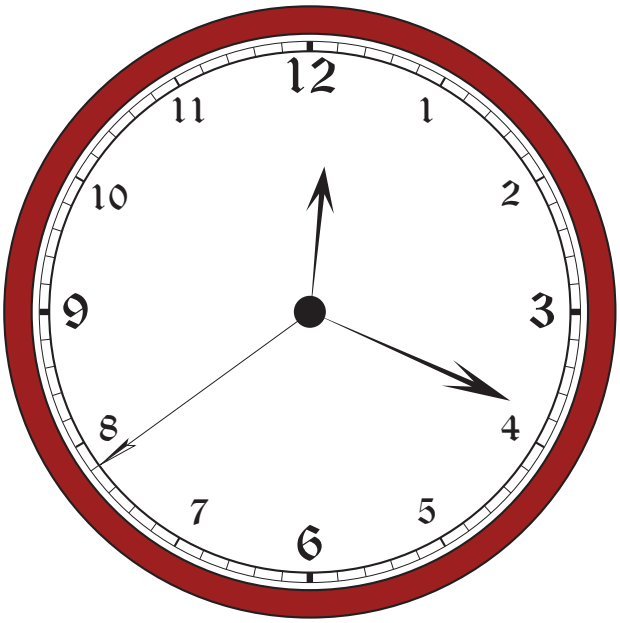
```
   group.Add_R_Item(null, "string:rex_clock.Minutes.y");
   group.Add_R_Item(null, "string:rex_clock.Hours.y");
   group.Add_W_Item(null, "string:rex_clock.InitSeconds.ycn",
    "int:" + (date.getSeconds() + 1));
   group.Add_W_Item(null, "string:rex_clock.InitMinutes.ycn",
    "int:" + date.getMinutes());
   group.Add_W_Item(null, "string:rex_clock.InitHours.ycn",
    "int:" + date.getHours());
   group.Write("opcClient.WaitForInitRex", null);
}
```

Čtení dat ze serveru - Ze serveru jsou přečtena data a následně ve funkci `StartClock()` jsou podle nich nastaveny hodiny v SVG grafice. Je aktivováno asynchronní čtení dat metodou `SwitchSubscriptionState()`.

```
opcClient.InitClock = function () {
   var group = opcClient.getItem("group");
   group.SwitchSubscriptionState(null);
   group.Read("opcClient.StartClock", null);
}
```

Spuštění algoritmu a asynchronní příjem dat - Po úspěšném nastavení hodin v SVG grafice je do skupiny dat pro zápis přidána položka `rex_clock.Start.YCN` a je na ni zapsána hodnota `true`, čímž se spustí algoritmus v systému REX. Změnami hodnot v systému REX dochází ke generování asynchronních událostí, které o tom informují uživatele v JavaScriptu. Uživatel na tyto události reaguje překrytím metody `SubscriptionCallback()`. V našem případě se v této metodě manipuluje s ručičkami hodin.

```
opcClient.StartClock = function (itemsArray) {
   ...
   group.Add_W_Item(null, "string:rex_clock.Start.YCN", "bool:true");
   group.Write(null, null);
}
opcClient.SubscriptionCallback = function (itemsArray) {
   ...
}
```



Obrázek 5.1: Příklad jednoduché vektorové *SVG* grafiky.

6 Závěr

V současné době je velkým trendem vytvářet aplikace v podobě tenkých klientů, protože to přináší řadu výhod. Implementovaná aplikace také vznikla v tomto duchu a díky tomu jediné, co klient na svém počítači potřebuje v případě, že je aplikace spuštěna na serveru, je internetový prohlížeč.

Výsledky testování výkonnosti aplikace z kapitoly 4 jsou pro porovnání zaneseny do grafu 6.1. Je patrné, že aplikace je nejvíce limitována serializací prvků v programovacím jazyce C#. To by mohlo být problematické při čtení velkého počtu dat s požadavkem na vysokou frekvenci čtení. Při dalším vývoji aplikace by tedy bylo vhodné implementovat serializaci efektivnějším způsobem než jaký je použit v aktuální verzi nebo aktuální řešení patřičně optimalizovat. Náročná serializace je jistě největší nevýhodou zvoleného způsobu implementace. Pro dosažení vyššího výkonu je možné potlačit serializaci metod pro objekty, jejichž metody uživatel nehodlá používat. Další možností je pro vyvolávání metod přímo využívat metodu *Invoke()*, která není na serializaci metod závislá. Tím se ale uživatel připraví o typovou kontrolu argumentů v JavaScriptu.

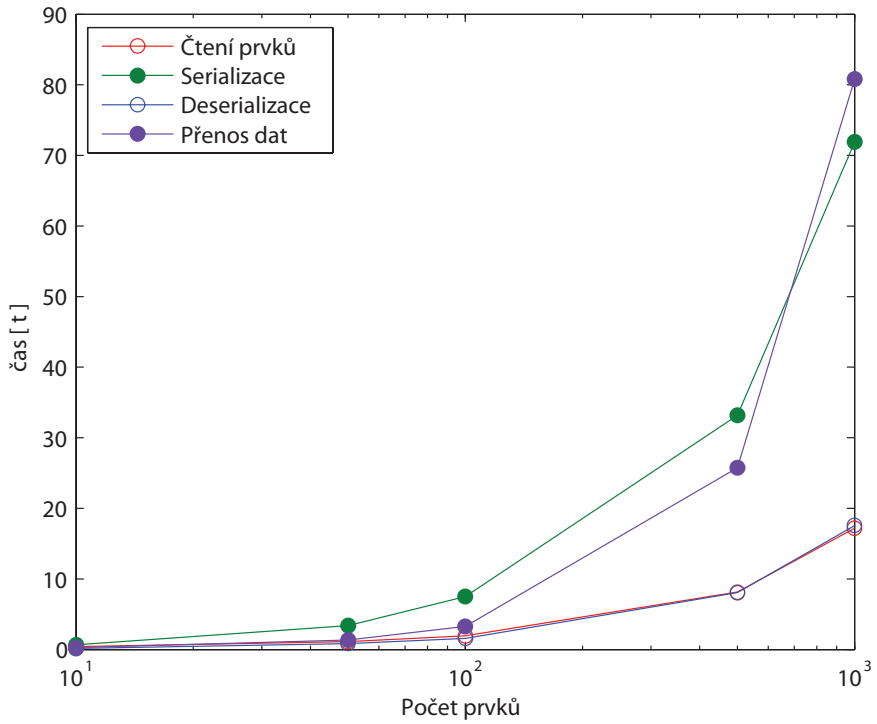
K přednostem aplikace jistě patří nezávislost na použitém API, ke kterému se aplikace připojuje. Pro jeho změnu nebo přidání nového API stačí odpovídajícím způsobem upravit dané pole s informacemi o assemblies[1], ze kterých má aplikace zpracovávat Metadata a vytvářet podle nich objekty. Díky tomu může být aplikace v budoucnu použita s novějšími verzemi OPC .NET API nebo pracovat s několika různými API zároveň.

Další výhodou je možnost si požadované API libovolně zaobalit do objektu v .NET a pro práci s API nadále používat rozhraní tohoto objektu prostřednictvím obrazů jeho metod v JavaScriptu.

Aplikace umožňuje připojení libovolného množství klientů zároveň, přičemž každý z nich má k dispozici vlastní proměnnou *workspace*. Z vlastností OPC .NET API dále vyplývá, že každý z těchto klientů se může připojit na libovolné množství OPC Serverů. Funkčnost aplikace byla ověřena na OPC Serveru řídícího systému REX a na *Matrikon OPC Simulation Serveru*.

Práce by dle mého názoru měla splňovat požadavky, které na ni byly kladeny. Aplikace by se nyní měla dostat do stavu důkladného testování v praxi, což pravděpodobně povede k dalším optimalizačním zásahům. Při vývoji byla využita základní implementace WebSocket serveru, kterou mi poskytl můj vedoucí práce Ing. Miroslav Kocánek.

[1] `string[] _assembliesFullNames`



Obrázek 6.1: Porovnání výkonu jednotlivých součástí aplikace

Literatura

[jso()] *Úvod do JSON*. Dostupné z: http://www.json.org/json-cz.html.

[web(2011)] *WebSocket.org*, 2011. Dostupné z: http://www.websocket.org/.

[Cog(1995-2010)] *What is OPC?* Cogent Real-Time Systems Inc., 1995-2010. Dostupné z: http://www.opcdatahub.com/WhatIsOPC.html#note1.

[Hickson(2012)] HICKSON, I. *The WebSocket API - Editor´s Draft 2 May 2012*. W3C, 2012. Dostupné z: http://dev.w3.org/html5/websockets/.

[Hurlbut(1998)] HURLBUT, M. *A Tutorial on Behavioral Reflection and its Implementation*, 1998. Dostupné z: http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/ref96/ref96.html.

[Ian Fette(2011)] IAN FETTE, A. M. *The WebSocket Protocol*. Google Inc., Isode Ltd., 2011. Dostupné z: http://tools.ietf.org/html/rfc6455.

[Mic(2012a)] *COM:Component Object Model Technologies*. Microsoft, 2012a. Dostupné z: http://www.microsoft.com/com/default.mspx.

[Mic(2012b)] *Reflection (C# Programming Guide)*. Microsoft, 2012b. Dostupné z: http://msdn.microsoft.com/en-us/library/ms173183%28v=vs.80%29.aspx.

[Neitzel(2009)] NEITZEL, L. *EXPRESS interface - Technical Overview*, 2009. Dostupné z: http://www.slideshare.net/JimCahill/express-interface-xi-technical-overview.

[OPC(2012)] *OPC Foundation - The Interoperability Standart for Industrial Automation & Related Domains*. OPC Foundation, 2012. Dostupné z: http://www.opcfoundation.org/.

Příloha

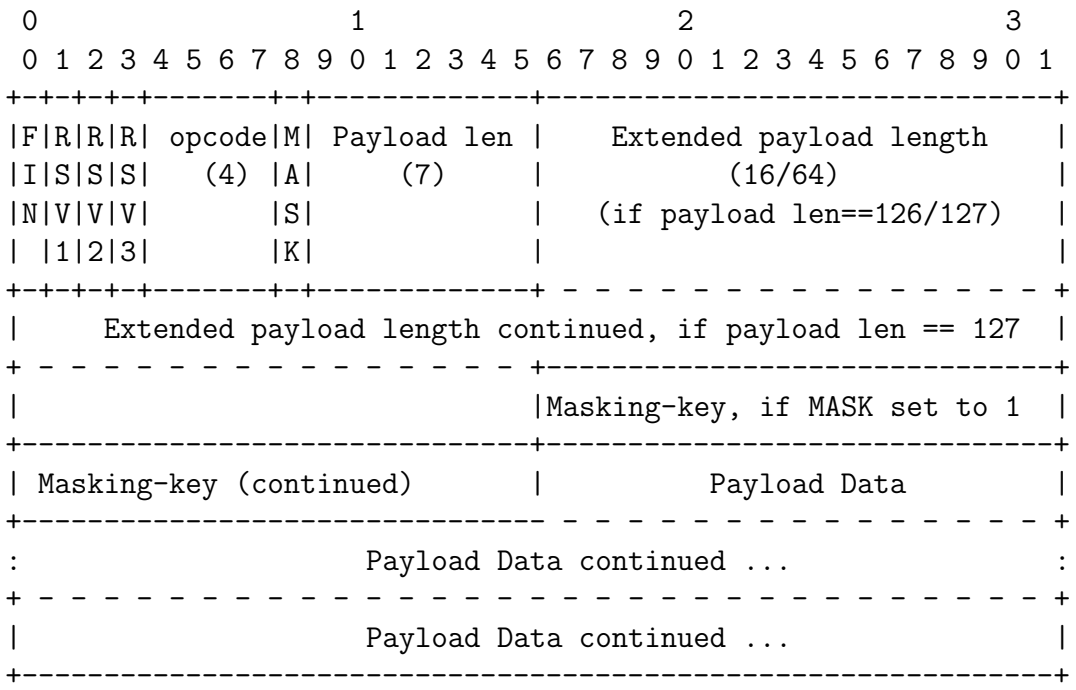
Handshake klienta:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Handshake pocházející ze strany serveru:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Příloha 1: Příklad *handshake* klienta a serveru technologie *WebSocket*.



Příloha 2: Podoba jednoho rámce protokolu *WebSocket*.

Příloha 3: **Legenda k jednotlivým částem rámce.**

FIN: 1 bit

- Tento bit značí zda se jedná o poslední rámec zprávy.

RSV1, RSV2, RSV3: každý 1 bit

- Tyto bity musí být nastaveny na hodnotu 0, pokud při handshake nedošlo
  k dojednání nějakých rozšíření protokolu.

Opcode: 4 bity

- Popisuje druh a tvar dat, přenášených rámcem. Definované jsou následující možná nastavení:
  - %x0 - navazující rámec
  - %x1 - textový rámec
  - %x2 - binární rámec
  - %x3-7 - hodnoty rezervované pro budoucí nekontrolní rámce
  - %x8 - požadavek na uzavření připojení
  - %x9 - ping
  - %xA - pong
  - %xB-F - hodnoty rezervované pro budoucí kontrolní rámce

Mask: 1 bit

- Popisuje zda jsou přenášená data maskována. Při nastavení na hodnotu 1 musí v části rámce Masking-key být přítomen klíč pro odmaskování dat. Maskována jsou data, posílaná klientem.

Payload length: 7 bitů, 7+16 bitů nebo 7+64 bitů

- Sekvence bitů, obsahující velikost přenášených dat. Pokud tato hodnota je v intervalu [0,125], pak odpovídá délce přenášených dat. Pokud je ale tato hodnota 126, následující 2 byty obsahují 16 bitový unsigned integer, jehož hodnota odpovídá délce přenášených dat. Pokud je tato hodnota dokonce 127, poté se na dalších 8 bytech nachází 64 bitový unsigned integer, jehož hodnota odpovídá délce přenášených dat.

Masking-key: 0 nebo 4 byty

- Všechny rámce posílané klientem serveru jsou maskované 32 bitovou hodnotou, která je součástí rámce. Tuto položku rámec obsahuje pouze tehdy, je-li položka rámce *Mask* nastavena na hodnotu 1.
- Maskovací klíč je 32-bitová hodnota náhodně zvolená klientem. Klient musí vždy při přípravě nového rámce vytvořit nový klíč, který musí být nepředvídatelný.

Payload data

- V poslední části rámce se nacházejí samotná data, která mají být přenesena. Tato data vzniknou spojením libovolných aplikačních dat spolu s daty rozšíření protokolu, pokud toto rozšíření bylo domluveno v otevíracím handshake.

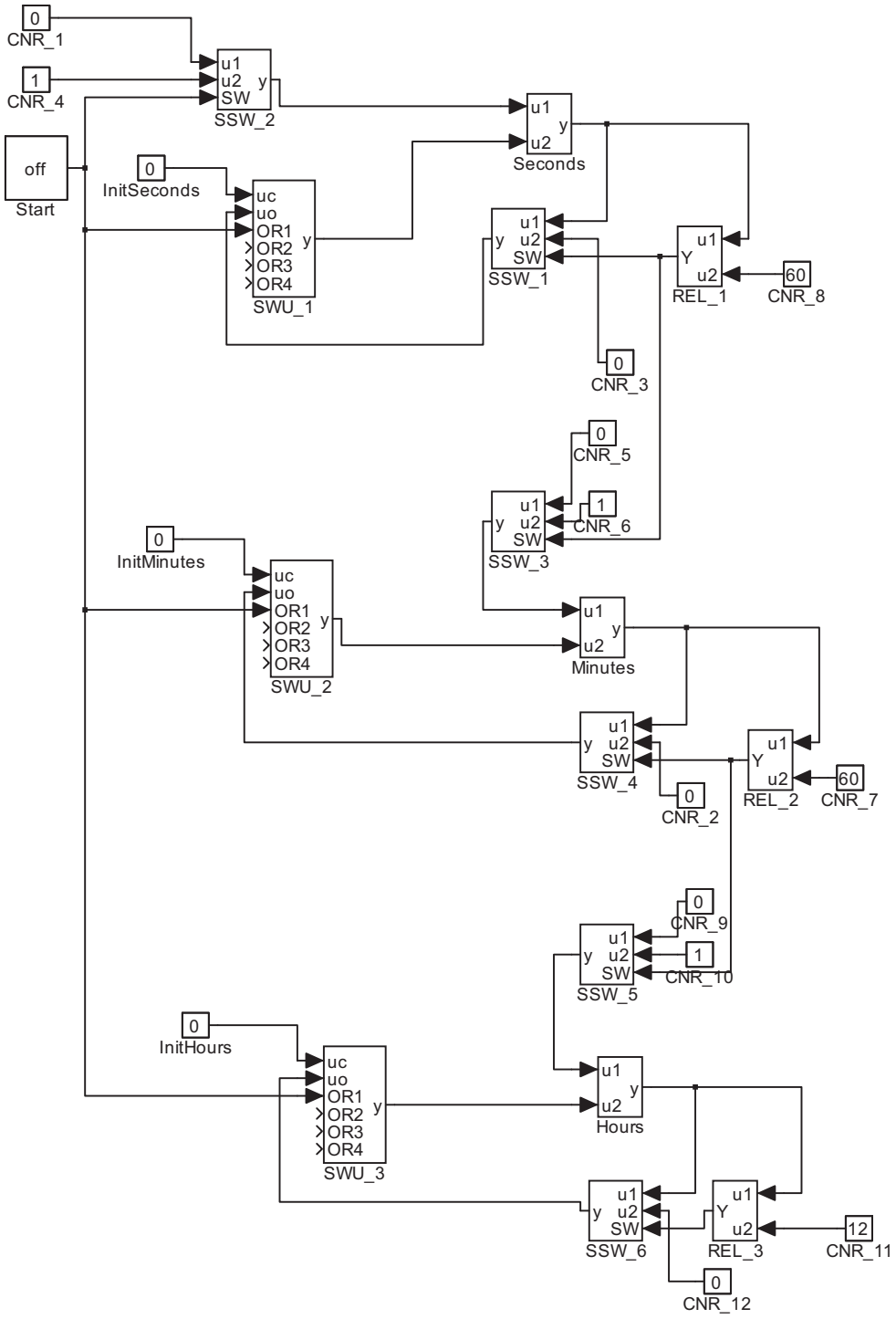
```
[Constructor(DOMString url, optional DOMString protocols),
 Constructor(DOMString url, optional DOMString[] protocols)]
 interface WebSocket : EventTarget {
 readonly attribute DOMString url;

 // ready state
 const unsigned short CONNECTING = 0;
 const unsigned short OPEN = 1;
 const unsigned short CLOSING = 2;
 const unsigned short CLOSED = 3;
 readonly attribute unsigned short readyState;
 readonly attribute unsigned long bufferedAmount;

 // networking
 [TreatNonCallableAsNull] attribute Function? onopen;
 [TreatNonCallableAsNull] attribute Function? onerror;
 [TreatNonCallableAsNull] attribute Function? onclose;
 readonly attribute DOMString extensions;
 readonly attribute DOMString protocol;
 void close([Clamp] optional unsigned short code, optional DOMString reason);

 // messaging
 [TreatNonCallableAsNull] attribute Function? onmessage;
          attribute DOMString binaryType;
 void send(DOMString data);
 void send(ArrayBuffer data);
 void send(Blob data);
};
```

Příloha 4: WebSocket rozhraní



Příloha 5: Schéma zapojení ukázkového příkladu v systému REX Control.