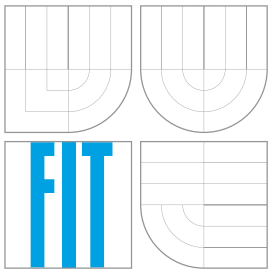
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSPARENTNÍ KLIENT-SERVER KOMUNIKACE POMOCÍ HTTP

TRANSPARENT CLIENT-SERVER HTTP COMMUNICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ONDŘEJ CHMELA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2013

Abstrakt

Dnešní webové aplikace jsou čím dál interaktivnější a uživatelsky komfortnější. To klade větší důraz na technologie umožňující komunikovat na pozadí se serverem, aniž by byl uživatel obtěžován neustálým obnovováním stránky nebo čekáním na stránku po dobu zpracovávání požadavku. Za tímto účelem jsou vyvíjeny nové technologie, které toto umožňují. Jednou z nich je WebSocket. Tato práce se zabývá popisem této technologie a její využití k vytvoření WebSocket serveru a knihovny pro obousměrné vzdálené volání procedur mezi klientem a serverem. Činnost je ověřena v demonstrační aplikaci.

Abstract

Today's web applications are becoming more interactive and user comfort. This places greater emphasis on technology for communicating with the server in the background, without user molested constant renewal of the page or waiting to for processing the request. For this purpose is the development of new technologies that make it possible. One of that is WebSocket. This paper describes the technology and its application to a WebSocket server and libraries for two-way remote procedure calls between the client and the server. The operation is verified in the demo application.

Klíčová slova

WebSocket, JSON–RPC, XML–RPC, vzdálené volání procedur, klient–server komunikace, RPC, WebSocket server

Keywords

WebSocket, JSON–RPC, XML–RPC, Remote Procedure Call, client–server communication, RPC, WebSocket server

Citace

Ondřej Chmela: Transparentní klient-server komunikace pomocí HTTP, bakalářská práce, Brno, FIT VUT v Brně, 2013

Transparentní klient-server komunikace pomocí HTTP

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Chmela
15.5.2013

Obsah

Kapitola 1

Úvod

Dnešní webové aplikace jsou čím dál interaktivnější a uživatelsky komfortnější. To klade větší důraz na technologie umožňující komunikovat na pozadí se serverem, aniž by byl uživatel obtěžován neustálým obnovováním stránky nebo čekáním na stránku po dobu zpracovávání požadavku. To souvisí s rozmachem AJAXu, který je používán snad na každém webu. S dalším vývojem sofistikovanějších aplikací začíná být potřeba, aby server mohl, bez potřeby vyvolání akce z klientké části, samovolně kontaktovat klienta. To AJAX neumožňuje, nebo alespoň ne jednoduše. Vždy je potřeba počáteční akce na klientské straně. Z tohoto důvodu začal vznikat protokol WebSocket, který byl přímo navržen pro tyto potřeby. V této práci se zaměřuji na popis způsobů komunikace mezi JavaScriptovým klientem a PHP serverem a implementací řešení obousměrného vzdáleného volání procedur s využitím protokolu WebSocket.

Druhá kapitola je zaměřena na popis protokolu HTTP. Je zde stručný úvod do problematiky protokolu.

Třetí kapitola se zabývá popisem vzdáleného volání procedur z JavaScriptového klienta (internetového prohlížeče) a jsou zde popsány 2 nejpoužívanější protokoly, které se pro tuto činnost používají. Jeden z těchto protokolů je poté využit v mojí implementaci.

Ve čtvrté kapitole se nachází popis WebSocket protokolu, jeho podpora v internetových prohlížečích a jeho použití.

Pátá kapitola obsahuje návrh vlastního řešení problému a popis naimplementovaných knihoven WebSocket serveru a RPC klienta/serveru.

Šestá kapitola pak popisuje implementaci demonstrační aplikace, kde jsou využity knihovny popsané v předešlé kapitole.

Kapitola 2

Protokol HTTP

HTTP protokol je aplikační protokol postavený nad protokoly TCP/IP využívající porty 80 a 443 pro šifrované spojení. Je založen na schématu dotaz - odpověď: klient pošle požadavek na objekt umístěný na serveru a server tento prostředek doručí klientovi. Jedná se o bezestavový protokol, HTTP transakce se skládá z jednoho požadavku a jedné odpovědi, mezi jednotlivými požadavky není žádný vztah [9, str. 32-33].

2.1 URL

Uniform Resource Locator slouží k identifikaci umístění zdroje (dokumentu, obrázku, ...) na internetu. Existuje také URN (Uniform Resource Name), který identifikuje zdroj podle jeho jména, nikoliv podle umístění, takže např. při přesunu na jiný server bude URN na rozdíl od URL stejná. Toto řešení se však neujalo a nyní se využívá právě URL. Z URL ve spojení s URN vznikne URI (Uniform Resource Identifier) [9, str. 30-32].

Všeobecné schéma URL:

```
scheme://host[:port]/path/.../[?query-string][#anchor]
```

URL se skládá z:

- scheme – určuje použitý protokol (např. http, ftp)
- host – doménové jméno nebo IP adresa cíle
- port – volitelná část specifikuje číslo portu, kde webový server naslouchá. Pokud není uveden používá se implicitně port 80
- path – cesta od kořenového adresáře na požadovaný dokument
- query-string – volitelná část obsahuje dynamické parametry. Např. při odeslání formuláře metodou GET
- anchor – také volitelná část, obsahuje odkaz na tzv. kotvy nebo-li záložky v dokumentu.

2.2 Zpráva

HTTP transakce se skládá z požadavku od klienta na zdroj umístěný na serveru. Server na takovýto požadavek reaguje buď kladnou nebo chybovou odpovědí. Každá zpráva, ať požadavek nebo odpověď se skládá ze 2 částí: záhlaví, obsahující HTTP hlavičky nesoucí informace k přenášenému obsahu a druhou částí jsou samotná data, která jsou pomocí protokolu přenášeny.

Požadavek:

```
GET /text.txt HTTP/1.1
Host: www.domena.cz
Accept-Language: en
```

Odpověď:

```
HTTP/1.1 200 OK
Content-type: text/plain
Content-length: 19

Hi! I´m a message!
```

Požadavek se skládá metody (v tomto případě *GET*), názvu zdroje (*/text.txt*) a verze protokolu (*HTTP/1.1*). Následují další HTTP hlavičky, z nichž je povinná host, určující doménu (*www.domena.cz*). Odpověď je složena z verze protokolu (*HTTP/1.1*), stavového kódu (*200*) a textového popisu kódu (*OK*). Dále následují další hlavičky. Hlavička *Content-type* udává v jakém datovém formátu je daný objekt. Hlavička *Content-lenght* pak udává délku dat přenášených v datové části. Po hlavičkách následuje volný řádek a po něm data [4, str. 13-15].

2.3 Metody

HTTP protokol podporuje různé příkazy nazývané HTTP metody. Metoda je součástí každého HTTP požadavku a říká serveru jakou akci má vykonat. Ne každou metodu však jsou servery povinni provádět, minimum jsou metody GET a HEAD, avšak většina serverů podporuje minimálně ještě POST, PUT a DELETE [9, str. 37][4, str. 11].

GET – nejjednodušší požadavek, podporován od prvních verzí protokolu HTTP. Slouží k získání zdroje (dokumentu, obrázku, ...).

POST – narozdíl od GET požadavku má tělo, ve kterém předává parametry. Typicky při odeslání formuláře.

HEAD – funguje podobně jako GET, ale v odpovědi vrací pouze HTTP hlavičky.

PUT – uloží data od klienta na server

DELETE – smaže zdroj na serveru

2.4 Stavové kódy

Stavový kód se nachází v každé odpovědi od serveru a oznamuje klientovi, zda vrací očekávanou odpověď, nebo klient musí provést dodatečnou akci (např. při přesměrování). Stavový kód může být parametrizován informací v záhlaví. Stavové kódy jsou seskupeny do kategorií, v případě HTTP verze 1.1 je jich pět [9, str. 42-46]:

1xx – kódy začínající *1* jsou klasifikovány jako informační. Neznamenají tedy úspěch nebo selhání, ale spíše poskytují informace jak s požadavkem dále pracovat.

2xx – kódy začínající *2* ozamují úspěšnou odpověď. Nejčastější odezva používá kód *200 OK*, což znamená úspěšně dokončení požadavku a požadovaný zdroj byl odeslán klientovi

3xx – kódy začínající *3* jsou pro účely přesměrování. Např. kódy *301 Moved Permanently* a *302 Moved Temporarily* oznamují klientovi, že zdroj se nachází na jiném místě. Toto místo má specifikované v hlavičce odpovědi *Location*

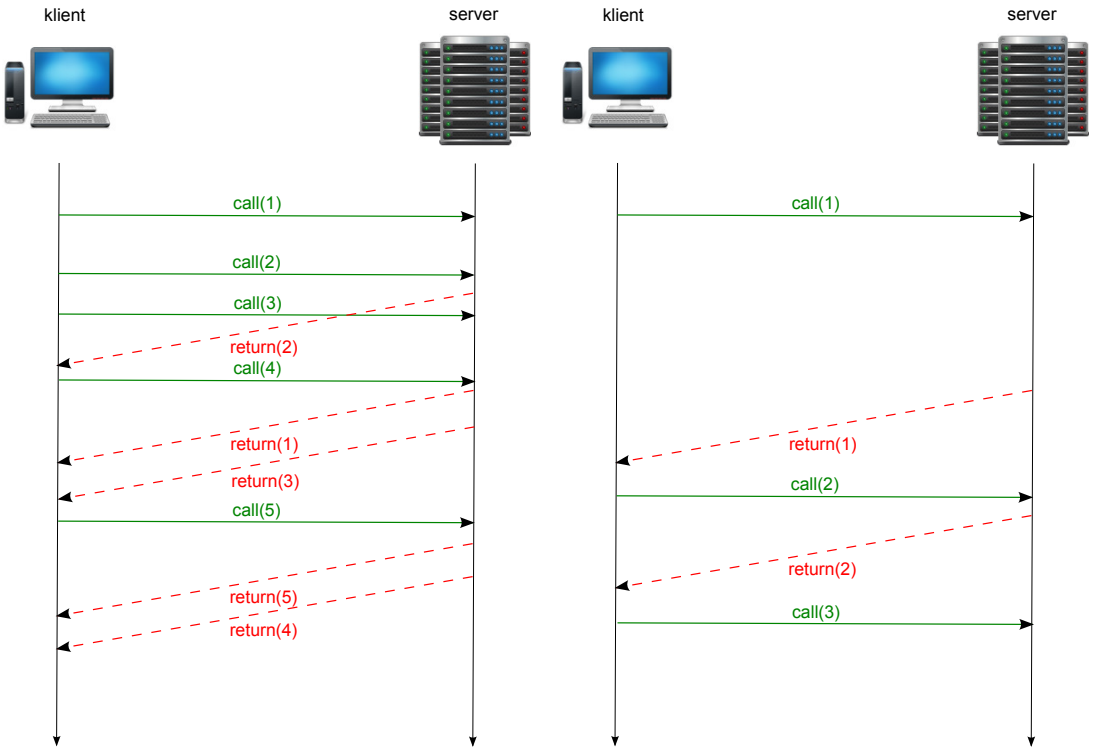
4xx – kódy začínající *4* reprezentují chyby, kterých se dopustil klient chybným požadavkem (ne však syntakticky chybným). Např. kód *404 Not Found* oznamuje klientovi, že zdroj, na který se dotazuje se na serveru nenachází.

5xx – kódy začínající *5* značí chyby na serveru. Např. *500 Internal Server Error*

Kapitola 3

Vzálené volání procedur přes protokol HTTP z klientského JavaScriptu

Vzdálené volání procedur z JavaScriptu je založeno na technologii AJAX. Jako formát dat se nejčastěji využívá XML nebo JSON a jako protokoly pro volání procedur se nejčastěji používají od nich odvozené: XML-RPC nebo JSON-RPC.



(a) Asynchronní požadavek (b) Synchronní požadavek

Obrázek 3.1: Ukázka zaslání AJAXových požadavků od klienta na server.

Požadavek může být iniciován výhradně z klientské strany, server na něj pouze odpovídá.

Požadavek může být synchronní i asynchronní. Při asynchronním požadavku, není běh skriptu během čekání na odpověď od serveru pozastaven a tudíž skript nadále obsluhuje události a nadále může vytvářet další požadavky, viz obrázek 3.1(a). Po příchodu odpovědi ze serveru, je běh skriptu přerušen, zavolá se obslužná funkce pro zpracování odpovědi a po jejím vykonání skript pokračuje od místa přerušení. Naopak při synchronním požadavku je běh skriptu pozastaven a nemůže obhsluhovat další události, viz obrázek 3.1(b).

3.1 AJAX

AJAX (Asynchronous JavaScript and XML) je technologie umožňující komunikovat na pozádí se serverem bez nutnosti znovunačtení celé stránky. Umožňuje tak ze serveru přenášet pouze ty části stránky, které byly na základě požadavku změněny a tím dochází k výraznému snížení datového toku a také přispívá k rychlejší odezvě aplikace. Požadavek může být synchronní i asynchronní, viz obrázek 3. Formát pro přenos dat se obvykle používá XML nebo JSON (v tomto případě se někdy používá označení AJAJ – Asynchronous JavaScript and JSON). [8]

3.2 XML-RPC

Kapitola vychází z [7].

Jedná se o protokol pro vzálené volání procedur. Pro přenos dat využívá POST metodu protokolu HTTP. Data jsou ve formátu XML.

3.2.1 Formát požadavku

```
POST /RPC2 HTTP/1.1
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181


<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
         </param>
      </params>
</methodCall>
```

Hlavička požadavku obsahuje metodu (POST), URL obsluhy požadavku (v tomto případě /RPC2) a verzi HTTP protokolu (1.1). Dále následují HTTP hlavičky, z nichž jsou povinné User-Agent, Host, Content-type a Content-length. Přičemž Content-type musí mít hodnotu *text/xml*. Po hlavičce následují samotná data ve formátu XML. Protokol vyžaduje, aby volání procedury bylo uzavřeno do tagu `<methodCall>`, jehož součástí jsou položky

`<methodName>` se jménem volané procedury a volitelně `<params>` s případnými parametry. Jméno metody může obsahovat pouze velká a malá písmena anglické abecedy, čísla, podtržítko, tečku, dvojtečku a lomítko. Metoda může přebírat libolovné množství parametrů různých typů. Každý parametr je uvozen tagem `<param>`, ve kterém se nachází tag `<value>` s hodnotou parametru. Hodnota parametru může být skalární veličina, struktura, nebo pole. Výčet skalárních veličin je uveden v tabulce 3.1.

Tag	Typ	Příklad

`<i4> nebo <int>`	4-bytové znaménkové celé číslo	`<value>`
		`  <i4>41</i4>`
		`</value>`

`<boolean>`	0 (false) a 1 (true)	`<value>`
		`  <boolean>1</boolean>`
		`</value>`

`<string>`	Řetězec	`<value>`
		`  <string>hello world</string>`
		`</value>`

`<double>`	Znaménkové číslo s plovoucí čárkou s dvojitou přesností	`<value>`
		`  <double>-12.214</double>`
		`</value>`

`<dateTime.iso8601>`	Čas nebo datum	`<value>`
		`  <dateTime.iso8601>`
		`  19980717T14:08:55`
		`  </dateTime.iso8601>`
		`</value>`

`<base64>`	Base64 kódované binární data	`<value>`
		`  <base64>`
		`  eW91IGNhbid0IHJlYWQgdGhpcyE=`
		`  </base64>`
		`</value>`

Tabulka 3.1: Výčet skalárních veličin protokolu XML-RPC

Není-li typ uveden, považuje se zadaná hodnota za typ *string*.

Datový typ struktura se definuje následujícím způsobem: uvozující tag je `<struct>`, který obsahuje 1-n položek `<member>`, jež obsahuje jméno položky `<name>` a hodnotu položky `<value>` definovanou podle pravidel v tabulce 3.1. Hodnota položky může být kromě skalárních hodnot i další struktura, nebo pole. Příklad definice struktury o 2 položkách:

```
<struct>
   <member>
      <name>lowerBound</name>
      <value><i4>18</i4></value>
```

```
   </member>
   <member>
      <name>upperBound</name>
      <value><i4>139</i4></value>
   </member>
</struct>
```

Definice typu pole se skládá z tagu `<array>`, obsahující libovolné množství položek `<value>` uzavřených do tagu `<data>`. Hodnoty moho být opět skalární, struktury nebo pole. Přiklad 4-prvkového pole:

```
<array>
   <data>
      <value><i4>12</i4></value>
      <value><string>Egypt</string></value>
      <value><boolean>0</boolean></value>
      <value><i4>-31</i4></value>
    </data>
 </array>
```

3.2.2 Formát odpovědi

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
      </param>
   </params>
</methodResponse>
```

Formát odpovědi je klasická HTTP odpověď. Zde je vhodné upozornit na návratový kód, který respektuje specifika standardu HTTP. Tudíž, pokud vznikla chyba, která nesouvisí s protokolem HTTP, ale s protokolem RPC, vrací se návratový kód 200 OK a specifikaci chyby řeší RPC protokol ve vlastní režii. Obsah hlavičky vyžaduje opět některé povinné parametry a to Content-Type (s hodnotou *text/xml*) a Content-Length. Samotná odpověď protokolu RPC je uzavřena v tagu `<methodResponse>`, který obsahuje buď tag `<params>`, obsahující návratovou hodnotu procedury, nebo v případě chyby tag `<fault>` obsahující její popis. Bezchybná odpověď je, jak již bylo zmíněno, uzavřená v tagu `<params>`, který obsahuje jednu položku `<param>` a ten obsahuje jednu položku `<value>` (opět definovanou podle pravidel uvedených výše). Chybovou odpověď uvozuje v tag `<fault>`, který obsahuje

položku `<value>`, což je struktura s 2-mi položkami: `<faultCode>` typu interger obsahující chybový kód a `<faultString>` typu *string* obsahující textový popis chyby. Přiklad odpovědi při chybě:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>faultCode</name>
               <value><int>4</int></value>
            </member>
            <member>
               <name>faultString</name>
               <value><string>Too many parameters.</string></value>
            </member>
          </struct>
      </value>
   </fault>
</methodResponse>
```

3.3 JSON-RPC 2.0

Kapitola vychází z [5].

Další z protokolů používaných pro RPC je JSON-RPC. Budu se zabývat popisem 2. verze, která není zpětně kompatibilní. Jedná se o bezstavový RPC protokol využívající JSON jako formát dat. Může být využít v rámci jednoho procesu, může být provozován přes HTTP a v dalších podobných aplikacích založených na zasílání zpráv.

3.3.1 JSON

JSON (JavaScript Object Notation) je jednoduchý formát pro výměnu dat. Jedná se o textový, na programovacím jazyce nezávislý formát. Je založen na dvou strukturách:

- Kolekce párů klíč/hodnota – v různých jazycích bývá realizováno různě: např. objekt, záznam, struktura, asociativní pole
- Tříděný seznam hodnot – pole, seznam

Příklad zápisu:

```
{"klíč1": "hodnota1", "klíč2": [1, 2, 3], "klíč3": {"klíč31": "hodnota31"}}
```

3.3.2 Formát požadavku

Existují 2 typy požadavků. Obyčejný požadavek a tzv. notifikace. Formát těchto požadavků je stejný až na položku *id*, kterou notifikace nemá. Notifikace se vyznačuje kromě absencí položky *id*, tím, že server na tento požadavek nesmí odpovědět. V tomto důsledku se klient nemusí dozvědět o chybě, která by případně mohla nastat. Formát požadavku se skládá z následujících položek:

jsonrpc – řetězec specifikující verzi protokolu. V tomto případě 2.0.

method – řetězec názvu metody, která má být vyvolána. Metody, které začínají rpc následovaném znakem U+002E nebo ASCII 46 jsou rezervované pro interní použití protokolu a nesmí být použity.

params – volitelná položka obsahující parametry volané metody

id – identifikátor požadavku, může obsahovat řetězec, číslo nebo hodnotu NULL (která se nedoporučuje). Identifikátor určuje klient a slouží ke spárování s odpovědí od serveru.

Pokud metoda vyžaduje parametry, musejí být zadány buď jako pole hodnot v pořadí, v jakém je metoda na serveru očekává, nebo mohou být pojmenovány. V tomto případě bude parametr *params* obsahovat objekt, jehož atributy odpovídají názvům parametrů, které metoda očekává.

Příklady volání:

- volání metody s parametry zadanými v pořadí, jak jej metoda očekává:

```
{"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
```

- volání metody s pojmenovanými parametry:

```
{"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23,
"minuend": 42}, "id": 3}
```

- notifikace:

```
{"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}
```

3.3.3 Formát odpovědi

Server musí na každý požadavek od klienta odpovědět, vyjma požadavku typu notifikace. Struktura odpovědi je následující:

jsonrpc – řetězec specifikující verzi protokolu. V tomto případě 2.0.

result – výsledek volání metody. Položka musí být přítomna v případě úspěšného vyvolání metody. V případě chyby být přítomna nesmí.

error – hodnota této položky musí být objekt viz níže. Položka musí být přítomna pouze tehdy, když nastane chyba.

id – identifikace požadavku. Hodnota musí být stejná jako hodnota požadavku. Pokud se nepodaří zjistit hodnotu *id* z požadavku (například chyba při analýze atd.), musí mět položka hodnotu *NULL*.

Pokud nastane chyba, odpověď musí obsahovat položku *error*, jejíž hodnota je objekt s následujícími položkami:

code – celé číslo udávající kód chyby

message – řetězec stručného popisu chyby (1 věta)

data – volitelná položka obsahující další informace o chybě

Kód chyby	Zpráva	Význam
-32700	Parse error	Chyba při analýze JSON dat.
-32600	Invalid Request	JSON neobsahuje validní požadavek podle specifikace protokolu.
-32601	Method not found	Metoda neexistuje nebo není dostupná.
-32602	Invalid params	Neplatné parametry metody.
-32603	Internal error	Interní chyba v protokolu.
-32000 až -32099	Server error	Uživatelem definované chyby.

Tabulka 3.2: Výčet chybových kódů protokolu JSON-RPC

Tabulka 3.2 zobrazuje chybové kódy protokolu. Chybové kódy z rozmezí -32768 až -32000 jsou vyhrazeny pro předem definované chyby. Kódy z této řady, které nejsou uvedeny v tabulce jsou vyhrazeny pro budoucí použití.

Příklady odpovědí (- - > požadavek, < - - odpověď):

```
-- > {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
< -- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

Příklady chybných odpovědí (- - > požadavek, < - - odpověď):

- neexistující metoda:

```
-- > {"jsonrpc": "2.0", "method": "foobar", "id": "1"}
< -- {"jsonrpc": "2.0", "error": {"code": -32601, "message": "Method
     not found"}, "id": "1"}
```

- neplatný JSON:

```
-- > {"jsonrpc": "2.0", "method": "foobar, "params": "bar", "baz]
< -- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse
     error"}, "id": null}
```

- neplatný požadavek:

```
-- > {"jsonrpc": "2.0", "method": 1, "params": "bar"}
< -- {"jsonrpc": "2.0", "error": {"code": -32600, "message": "Invalid
     Request"}, "id": null}
```

3.3.4 Dávkové zpracovaní

Protokol umožňuje tzv. dávkové zpracování, kdy klient může poslat více požadavků najednou. Jednotlivé požadavky jsou zabaleny do pole. Server odpoví na každou žádost vyjma notifikací. Odpovědi přitom můžou být v libobolném pořadí, to dovoluje serveru zpracovat dávku paralelně. Pokud nastane chyba při analýze JSONu dávky, nebo pole dávky obsahuje jednu hodnotu, odpoví server právě jednou odpovědí. V opačném případě server odešle pole s opdovědmi. Nemá-li server co vrátit, např. dávka byla tvořena notifikacemi, nesmí server vratít vůbec nic, ani prázdné pole.

Příklady dávkového zpracování:

- dávka:

```
-- > [
        {"jsonrpc": "2.0", "method": "sum", "params": [1,2,4],
        "id": "1"},
        {"jsonrpc": "2.0", "method": "notify_hello", "params": [7]},
        {"foo": "boo"},
        {"jsonrpc": "2.0", "method": "foo.get", "params": {"name":
        "myself"}, "id": "2"},
        {"jsonrpc": "2.0", "method": "get_data", "id": "3"}
     ]
< -- [
        {"jsonrpc": "2.0", "result": 7, "id": "1"},
        {"jsonrpc": "2.0", "error": {"code": -32600, "message":
        "Invalid Request"}, "id": null},
        {"jsonrpc": "2.0", "error": {"code": -32601, "message":
        "Method not found"}, "id": "2"},
        {"jsonrpc": "2.0", "result": ["hello", 5], "id": "3"}
      ]
```

- dávka s notifikaceni:

```
-- > [
        {"jsonrpc": "2.0", "method": "notify_sum",
        "params": [1,2,4]},
        {"jsonrpc": "2.0", "method": "notify_hello", "params": [7]}
     ]
< -- // žádná odpověď
```

- prázdná dávka:

```
-- > []
< -- {"jsonrpc": "2.0", "error": {"code": -32600, "message":
     "Invalid Request"}, "id": null}
```

- chybný JSON:

```
-- > [
```

```
        {"jsonrpc": "2.0", "method": "sum", "params": [1,2,4],
        "id": "1"},
        {"jsonrpc": "2.0", "method"
    ]
< -- {"jsonrpc": "2.0", "error": {"code": -32700, "message":
    "Parse error"}, "id": null}
```

- chybná dávka:

```
-- > [1,2]
< -- [
        {"jsonrpc": "2.0", "error": {"code": -32600, "message":
        "Invalid Request"}, "id": null},
        {"jsonrpc": "2.0", "error": {"code": -32600, "message":
        "Invalid Request"}, "id": null}
    ]
```

Kapitola 4

WebSocket

WebSocket protokol umožňuje webovým aplikacím plně duplexní obousměrnou komunikaci mezi webovým prohlížečem a serverem. Nabízí tak nové řešení, které současná řešení pomocí protokolu HTTP nejsou schopny nabídnout a simulují jej např. periodický dotazováním na server apod. [1, str. 137-138]. Výhodou WebSockets je, že komunikace neprobíhá přes HTTP protokol, takže komunikace není zatížena zbytečnými HTTP hlavičkami a dochází tak k výraznému snížení datového toku (až 1000:1) a doby odezvy (až 3:1) [1, str. 143-145].

4.1 Protokol

WebSocket je aplikační protokol postaven nad TCP/IP vrstvou. Specifikace protokolu je definovaná v RFC 6455 [3]. V současné době ještě nebyl vývoj ukončen a specifikace se může v budoucnu změnit. Je navržen tak, aby nahradil stávající obousměrné komunikační technologie využívající protokol HTTP a stávající infrastrukturu (proxy, filtrování, autentizace), ale v současné době je kompatibilní i se stávající infrastrukturou, např. je možné využívat porty HTTP 80 a 443 (s příslušným serverem, který dokáže rozpoznat, že se jedná o WebSocket spojení) apod. Nicméně, návrh neomezuje protokol WebSocket na spolupráci výhradně s HTTP serverem a budoucí implementace mohou například využívat jiný port a použít tak jednodušší handshake bez HTTP hlaviček [3, str. 4-5]. Protokol používá URI schéma *ws* pro nešifrovanou a *wss* pro šifrovanou komunikaci.[3]

4.1.1 Handshake

Komunikace je zahájena handshakem. Handshake je kompatibilní s HTTP požadavkem doplněným o některé hlavičky. To umožňuje používat porty 80 a 443, na kterých běží běžné webové servery a ty mohou buď spojení obsloužit nebo ho odmítnout v případě, že WebSokety nepodporují.[3]

Ukázka handshaku:

```
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
```

Request-URI identifikuje koncový bod WebSocket spojení. *Host* slouží k ověření hostitele. Přídavné hlavičky slouží k nastavení protokolu. Jsou dostupné hlavičky pro volbu subprotokolu (*Sec-WebSocket-Protocol*) a seznam rozšíření, které podporuje klient (*Sec-WebSocket-Extensions*). Origin slouží k ochraně proti neoprávněnému cross-origin použití WebSoketového serveru skripty používající WebSoketové API v prohlížeči. *Sec-WebSocket-Key* slouží k prokázání přijetí handshaku. Server odešle v odpovědi *Sec-WebSocket-Accept* hodnotu v base64 kódování sestavenou z SHA-1 hashe hodnoty *Sec-WebSocket-Key* spojenou s jedninečným identifikátorem 258EAFA5-E914-47DA-95CA-C5AB0DC85B11.[3, str. 6-8]

Server v případě úspěchu odpoví:

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

Tímto je spojení ustanoveno a můžou se přenášet data.

4.1.2 Přenos dat

Data jsou přenášená sekvencí tzv. rámů. Rám je v podstatě obdoba paketu. Rozeznáváme 2 druhy rámů: řídící a datové. Řídící rámy slouží k zajištění a kontrole spojení, datové k přenosu dat. Typ rámu se definuje hodnotou *opcode* v hlavičce rámu. Typy rámů s hodnotami *opcode*:

- %x0 navazující rám – Obsahuje data, která bezprostředně navazují na předchozí, viz fragmentace
- %x1 textový datový rám – Obsahuje data v textovém formátu a kódování UTF–8.
- %x2 binární datový rám – Obsahuje data v binárním formátu.
- %x8 řídící rám uzavírající spojení – Slouží k uzavření spojení.
- %x9 řídící rám ping – Slouží k ověření spojení. Po obdržení tohoto rámu musí příjemce odeslat Pong rám.
- %xA řídící rám pong – Reakce na Ping rám, nebo moho být zasílány jednosměrně, např. pro ověření spojení

Hodnoty %x3 – 7 a %xB-F jsou rezervovávy pro budoucí použití datových respektive řídících rámů.

Z bezpečnostních důvodů musejí být všechna data pocházející od klienta maskována. Ty klient vytvoří pomocí 32–bitového klíče, který si náhodně vygeneruje pro každý rám a podle algoritmu 4.1 zakóduje přenášená data. Oktet *i* zakódovaných dat (*transformed-octet-i*) je XOR oktetu *i* originálních dat (*original-octet-i*) s oktetem na pozici: *i modulo 4* maskovacího klíče (*masking-key-octet-j*):

```
j = i MOD 4
transformed-octet-i = original-octet-i XOR masking-key-octet-j
```

Listing 4.1: Algoritmus maskování klientských dat

Pokud server obdrží nemaskovaná data musí ukončit spojení. Naopak server nesmí data maskovat a pokud klient obdrží maskovaná data, musí taktéž ukončit spojení. *Data* nebo-li *Payload data* se skládají z tzv. *Extension data* (data pro rozšíření) následovaných *Application data* (data aplikace). V případě, že není definováno žadné rozšíření, jsou *Extension data* prázdná [3, str. 27-38].

4.1.3 Fragmentace

Fragmentace umožňuje odesílat zprávy s neznámou velikostí, odesílá–li se zpráva bez využití vyrovnávací paměti. Pokud zpráva nemůže být fragmentována, musí jí koncový bod být schopen uložit do svojí vyrovnávací paměti. Při fragmentaci, může server zvolit přiměřenou velikost vyrovnávací paměti, a když je paměť zaplněná, zapsat fragment na síť. Další význam fragmentace je pro multiplexing, kde není žádoucí obsadit výstupní kanál odesíláním velké zprávy, ale posílat menší fragmenty a lépe tak sdílet výstupní kanál.[3, str. 33-34]

4.1.4 Rozšíření

Rozšíření mohou přidávat do protokolu nové vlastnosti. Tyto rozšíření musejí být sjednány mezi zařízeními při počátečním handshaku (kap. 4.1.1).

Příklady rozšíření:

- Definice rezervované hodnoty *opcode*
- Jiné pořadí dat *Extension data* a *Application data* v *Payload data*

4.2 Podpora v prohlížečích

V současné době jsou WebSocket podporovány ve všech moderních prohlížečích. V tabulce 4.1 je výčet nejpoužívanějších prohlížečů a verze, od kterých jsou podporovány.[2]

Prohlížeč	Detail
Chrome	od verze 14
Firefox	od verze 6
Internet Explorer	od verze 10
Opera	od verze 12.1
Safari	od verze 6

Tabulka 4.1: Podpora WebSocket v prohlížečích

4.3 Implementace WebSocket v prohlížeči

Protokol definuje rozhraní pro použití WebSocket v JavaScriptových aplikacích [1, str. 140-141]. Toto rozhraní je implementováno v prohlížeči. Na ukázce 4.2 je znázorněn popis tohoto rozhraní [6].

Na příkladu 4.3 je ukázka použití [1, str. 141]. Vytvoří se instance objektu WebSocket s parametrem adresy serveru. Potom se nastaví callbacky pro jednotlivé události. Rozhraní je založeno na událostech, to znamená, že při vzniku události na rozhraní se přeruší aktuální

```
[Constructor(in DOMString url, in optional DOMString protocol)]
interface WebSocket {
  readonly attribute DOMString URL;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSED = 2;
  readonly attribute unsigned short readyState; // stav spojení
  readonly attribute unsigned long bufferedAmount; // počet bajtů, které jsou
       ve frontě připraveny k~odeslání

  // networking
  attribute Function onopen; // callback volající se při vytvoření spojení
  attribute Function onmessage; // callback volající se při příchozí zprávě
  attribute Function onclose; // callback volající se po ukončení spojení
  boolean send(in DOMString data); // metoda odeslání dat
  void close(); // metoda pro ukončení spojení
};
WebSocket implements EventTarget;
```

Listing 4.2: WebSocket API

vykonávání skriptu a zavolá se příslušná obslužná metoda, která obslouží vzniklou událost. Poté se pokračuje v kódu od místa přerušení.

```
var ws = new WebSocket("ws://example.com");
ws.onopen = function(evt) { alert("Spojení navázáno ..."); };
ws.onmessage = function(evt) { alert( "Přijata zpráva: " + evt.data); };
ws.onclose = function(evt) { alert("Spojení uzavřeno."); };
```

Listing 4.3: Použití WebSocket API

Kapitola 5

Návrh řešení

Řešení vzdáleného volání procedur jsem založil na protokolu WebSocket kap. 4 a jako formát dat jsem zvolil JSON-RPC 2.0 kap. 3.3. Toto řešení umožnuje volání procedur jak z klientské strany tak i ze strany serveru. Existujících implementací WebSocket v PHP není mnoho a ty, které podporují nenjnovější verzi RFC 6455 [3] jsem objevil pouze jednoho a to knihovnu Ratchet[1] používající WAMP protokol[2] (The WebSocket Application Messaging Protocol). Pro klientskou implementaci RPC v JavaScriptu existuje knihovna AutobahnJS[3]. Pro řešení jsem zvolil implementaci vlastního WebSocket a RPC serveru pro serverovou i klientskou část z důvodu pochopopení principů technologie a také z důvodu, že implemetace serveru Ratchet a všechny ostatní implementace pracují jako tzv. *iterativní server*, kde jeden proces serveru obsluhuje postupně všechny připojené klienty a tudíž klienti musejí čekat než jejich požadavky sever vyřídí. Proces je znázorněn na obrázku 5.1(a). Moje implementace serveru pracuje jako *konkurentní server* obr. 5.1(a), kde pro každé nové spojení je vytvořen nový proces serveru, který dané spojení obshluhuje nezávisle na ostatních.

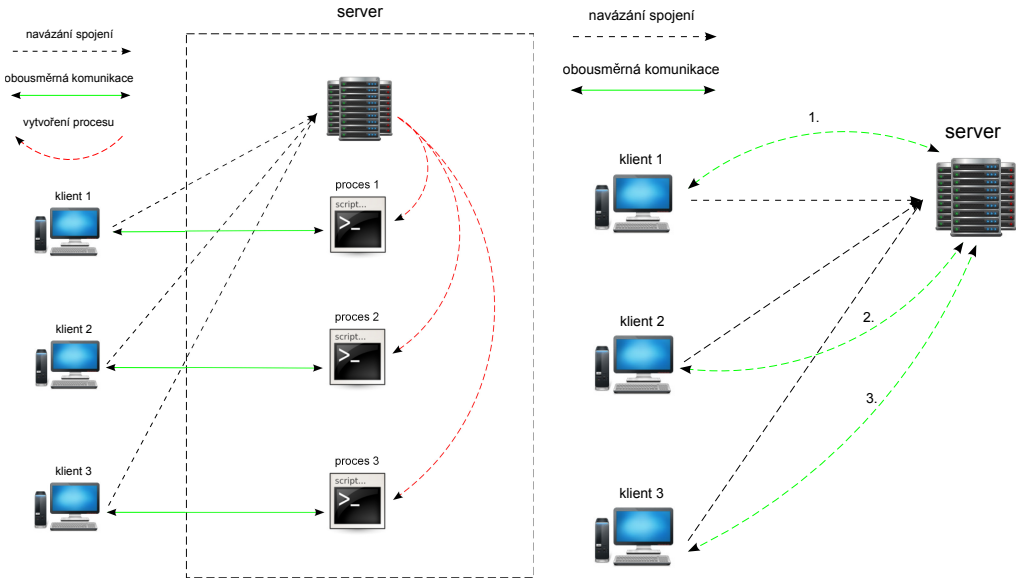
5.1 Implementace serveru

Serverová část se skládá ze dvou částí. První části je WebSocket server starající se o přenos zpráv a druhou částí RPC server obshluhující volání procedur. Nad těmito vrstvami je postavena uživatelská aplikace. Obrázek 5.2.

WebSoketová část serveru je implementována podle současného RFC 6455 [3]. Nebylo implementováno celé RFC, pouze zakladní části jako ustanovení spojení a přenos dat (kap. 4.1.2) bez fragmentace (kap. 4.1.3). Jak bylo psáno výše, jendá se o konkurentní server s možností blokovacího a neblokovacího módu. V blokovacím módu je proces pozastaven a čeká na příchozí data, v neblokovacím módu pokračuje dál. PHP standardně neumožňuje vytváření procesů, proto je nutné použít rozšíření PCNTL[4]. Ukázka 5.1 ukazuje použití serveru.

Nejdříve vytvoříme objekt WebSocket\Server s parametry adresa a port na kterém server běží. Poté voláme metodu *run*, které jako parametr předáme callback na funkci, která je volána v novém procesu po úvodním handskaku kap. 4.1.1. Tímto je server spuštěn a řízení je předáno dané funkci. Tělo funkce tvoří nekonečná smyčka ve které se zpracovávají a odesílají data. Aplikaci ukončíme vyskočením ze smyčky a proces ukončí spojení s klientem.

[1] http://socketo.me
[2] http://wamp.ws/
[3] http://autobahn.ws/js
[4] http://php.net/manual/en/book.pcntl.php



(a) Spojení s konkurentním serverem (b) Spojení s iterativním serverem

Obrázek 5.1: Schéma práce konkurentního a iterativního typu serveru.

Druhá část serveru, RPC server, implementuje protokol JSON-RPC kap. 3.3 a stará o proces volání procedur. Je postaven nad WebSocket serverem.

```
interface IJSONRPCServer {
  public $blockingMode = TRUE;

  public $init = NULL;
  public $beforeReqest = NULL;
  public $afterRequest = NULL;

  public function run();

  public function call($procedureName, array $params = array());

  public function add($procedureName, $callback);
}
```

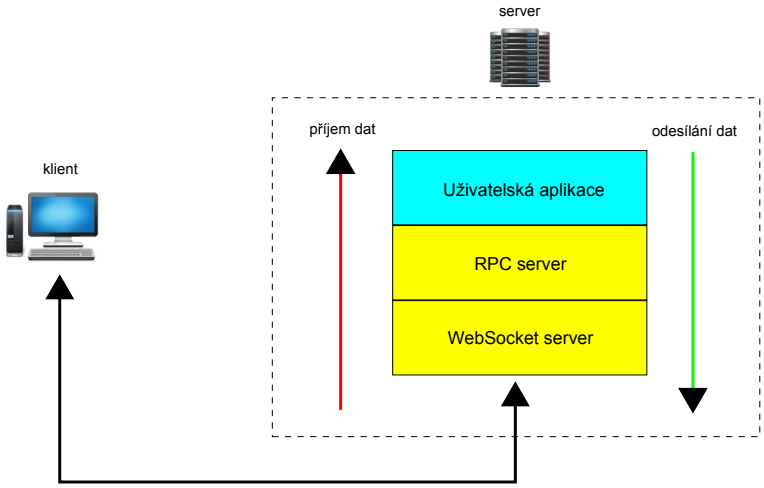
Listing 5.2: Rozhraní RPC serveru

Rozhraní RPC serveru př. 6.2 je tvořeno metodami *run* pro spuštění serveru, metodou *call* pro volání procedury u klienta a metodou *add* pro registraci serverové procedury, aby mohla být volána z klienta. Dále ho tvoří atribut *blockingMode*, který nastavuje, zda server bude pracovat v blokovacím nebo neblokovacím módu. Zbývající 3 atributy: *init,beforeReqest,afterRequest* slouží k uložení callbacku na funkce, které se volají v průběhu zpracování dat přijatých od klienta.

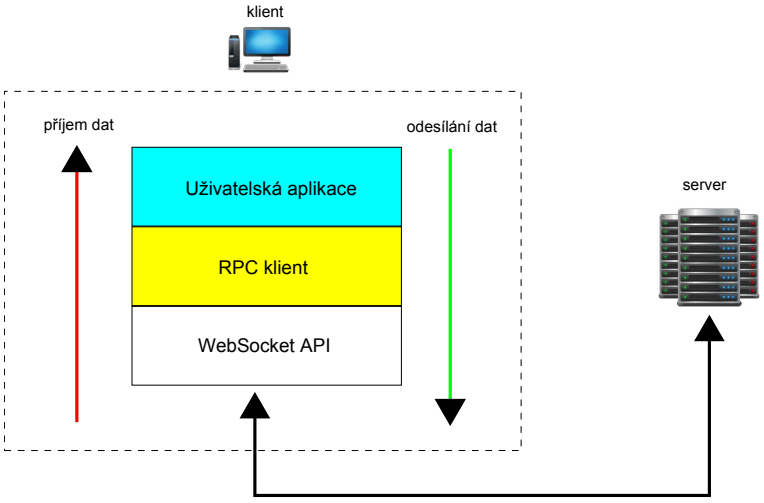
Registrace metody pro volání z klientské strany probíhá následovně:

```
$server->add("soucet","sum");
```

Při zavolání metody *soucet* z klienta RPC server vyvolá proceduru *sum* a její návratovou hodnotu odešle zpět klientovi. Server kontroluje zda daná procedura existuje a také počty popřípadě názvy parametrů a v případě neshody oznámí chybu klientovi.



Obrázek 5.2: Struktura aplikace na serveru.



Obrázek 5.3: Struktura aplikace na klientské straně.

Volání klientské procedury probíhá voláním metody *call* s parametrem jméno procedury, pod kterým je metoda zaregistrovaná na klientské straně a případnými parametry. Metoda vrátí objekt požadavku, na kterém lze nadefinovat callbacky, které jsou volány jakmile se procedura u klienta vykoná. Jedná se o callbacky *onSuccess* a *onError*, které jsou volány při úspěšném resp. neúspěšném volání procedury. Těmto funkcím jsou potom předány parametry návratová hodnota procedury a reference na RPC server resp. objekt chybové zprávy a reference na RPC server. Volání procedur jak ze serveru tak i z klienta lze libovolně křížit, tzn. při volání procedury např. z klienta lze v dané proceduře na serveru před vrácením hodnoty vyvolat metodu na klientovi apod. Je to dáno tím, že při volání metody nebo vrácení hodnoty nedojde okamžitě k odeslání těchto dat, ale jsou uloženy do fronty a odeslány najednou při vhodné příležitosti. Př. 5.4 ukazuje volání procedury na klientovi zakomponované do předchozího příkladu 6.2.

Základní algoritmus RPC serveru je znázorněn v ukázce 5.4. Základ opět tvoří nekonečná smyčka, ve které se postupně volají uživateslky definované callbacky a dochází k vyprázdnění fronty dat.

```
/* vytvoření serveru; nastavení adresy a portu */
$server = new WebSocket\Server("localhost", 40000);

/* spuštění; parametr metody je callback na funkci, která je zavolána novým
    procesem po ustanovení spojení */
$server->run("application");

function application(\WebSocket\Network\IConnection $connection)
{
  /* základní smyčka aplikace */
  while (1) {
    try {
      /* čtení dat ze soketu, v blokovacím módu čeká v neblokovacím pokračuje
           a vrací NULL */
      $data = $conn->recv(/* TRUE pro blokovaci mod, jinak FALSE */);

      if ($request !== NULL) {
        // zpracování dat ...
        $conn->send("odpoved"); /* odeslání dat klientovi */
      }
    }
    catch (\Exception $e) {
      break;
    }
  }
}
```

Listing 5.1: Použití WebSocket serveru

5.2 Implementace klienta

Podobně jako serverovou část jsem se rozhodl implementovat i klientskou část v JavaScriptu. V tomto případě se jednalo pouze o RPC klient. Struktura aplikace je podobná jako u serveru obr. 5.3.

První vrstvu tvoří WebSocket API (kap. 4.3), nad ní se nachází RPC klient. Jeho princip a použití je obdobné jako na straně serveru. Není zde potřeba se zabývat čtením dat ze socketů, to obstarává WebSocket API v prohlížeči, takže použití je snadné, viz příklad 5.5.

V prvním kroku inicializujeme klienta s parametrem adresa serveru a popřípadě portem. Potom pomocí metody *add* zaregistrujeme proceduru *rozdil* a pomocí metody *call* zavoláme serverouvou proceduru *soucet*. Podobně jako v případě serveru můžeme nadefinovat callbacky *onSuccess* a *onError*. Na rozdíl od serveru zde musíme explicitně provést volání voláním metody *execute*, která danou proceduru ihned zavolá. Není zde tedy fronta typu jako je na serveru, ale všechna volání se ihned provedou. Je to dáno povahou WebSoket API, které se stará o příjem/odesílání dat a volá příslušné obslužné funkce a není tedy možné do toho procesu zakomponovat nějaký jiný kód, v tomto případě vyprázdnění fronty jako je to na serveru.

```
$server = new WebSocket\Server("localhost", 40000);
$rpc = new WebSocketJSONRPC\Server($server);
$rpc->init = function () {
  // inicializace aplikace;
};
$rpc->beforeRequest = function () {
  // ....;
};
$rpc->afterRequest = function () {
  // ....;
};

$rpc->add("sum","soucet");
$rpc->run();

function sum($a, $b)
{
  /* získání reference na RPC server - je předán vždy jako poslední parametr.
       Nemůže být uveden v~hlavičce funkce, protože by selhala kontrola počtu
      parametrů */
  $client = func_get_arg(2);

  /* zavolání klientské procedury rozdil s~parametry a~= 1, b = 2 */
  $request = $client->call("rozdil",array("a" => 1, "b" => 2));

  /* nadefinování onSuccess callbacku */
  $request->onSuccess(function($result,$server) {
    echo $result; /* vypsání návratové hodnoty od klienta */
  });

  /* nadefinování onError callbacku - volán při chybě */
  $request->onError(function($error){
    echo $error->getErrorMessage();
  });

  return $a + $b; /* vrácení součtu klientovi */
}
```

Listing 5.3: Volání klientské procedury

```
// volání init callbacku;
/* základní smyčka aplikace */
while (1) {
  try {
    // volání beforeReqest callbacku;
    // odeslání všech dat z~fronty klientovi;

    /* čtení dat ze soketu, v~blokovacím módu čeká v~neblokovacím pokračuje a
       ~vrací NULL */
    $request = $conn->recv(/* TRUE pro blokovaci mod, jinak FALSE */);

    if ($request !== NULL) {
      // zpracování dat ...
    }

    // odeslání všech dat z~fronty klientovi;
    // volání afterReqest callbacku;
  }
  catch (\Exception $e) {
    break;
  }
}
```

Listing 5.4: Algoritmus RPC serveru

```
/* inicializace klienta */
var rpc = new wsrpc("ws://localhost:40000");

/* registrace procedury rozdil */
rpc.add('rozdil',function (a,b) {
  return a~- b;
});

/* volani serverové procedury soucet */
rpc.call("soucet",[2,2]).
  onSuccess(function (result) { /* definování onSuccess callbacku */
    alert(result);
  }).
  onError(function (response) { /* definování onError callbacku */
    alert(response.getErrorMessage())
  }).
  execute(); /* zavolání procedury */
```

Listing 5.5: Použití RPC na klientské straně

```
[Constructor(in DOMString url)]
interface wsrpc {
  void add(procedureName, callback); // zaregistruje proceduru
  outgoingWsRequest call(procedreName, args); // zavolání procedury na
      serveru
};

interface outgoingWsRequest {
  /* zaregistrování funkce volané při úspěchu */
  outgoingWsRequest onSuccess(callback);

  /* zaregistrování funkce volané při neúspěchu */
  outgoingWsRequest onError(callback);

  /*  zavolání procedury na serveru */
  void execute();
};
```

Listing 5.6: Rozhraní RPC klienta

Kapitola 6

Demonstrační aplikace

Jako demonstrační aplikaci využívající implementované knihovny jsem zvolil jednoduché piškvorky, které pouze na hracích plochách ve dvou oknech prohlížeče propisují označená políčka.

6.1 Server

Serverová část aplikace je řešena netradičním způsobem. Základem komunikace mezi procesy hráčů je MYSQL databáze s dvěmi tabulkami. Tabulku players procesy využívají k zjištění, zda jsou oba hráči online. Dokud tabulka neobsahuje 2 záznamy PID procesů hra vyčkává. Tabulku move se ukládají záznamy o daném tahu a procesy se v periodě 1s dotazují, zda nebyl proveden tah protihráče. Jelikož je server v blokovacím módu, tak pro dotazování využívají signál SIGALRM, který je vyvolán časovačem každou 1 sekundu. V obslužné funkci proces zkotroluje, zda byl proveden nový tah a případně zavolá příslušnou proceduru na klientovi.

Server nabízí klientovi následující procedury:

```
/* proceduru klient vyvolá při kliknutí do hrací plochy. Jako parametry
   přijímá číslo řádku a~sloupce, který byl vybrán */
void click($row,$col);
```

Listing 6.1: Dostupné procedury pro klienta v demonstrační aplikaci

6.2 Klient

Klientská část aplikace zobrazuje a obsluhuje herní plochu a vykresluje značky do daných políček na základě pokynů od serveru.

Klientovi procedury dostupné ze serveru jsou zobrazeny v ukázce 6.2.

6.3 Průběh hry

Po připojení 1. hráče si hráč zabere křížky x a oznámí tuto skutečnost klientovi voláním procedury *setCursor* s parametrem x. Zároveň volá proceduru klienta *waitForPlayer*, čímž oznámí, že je ve hře zatím sám. Jakmile se připojí 2. hráč, zabere si kolečka o a oznámí tuto skutečnost klientovi voláním procedury *setCursor* s parametrem o. Při kontrole stavu hry v 1 sekundových intervalech hráč 1 zjistí, že je přítomen další hráč a zavolá proceduru

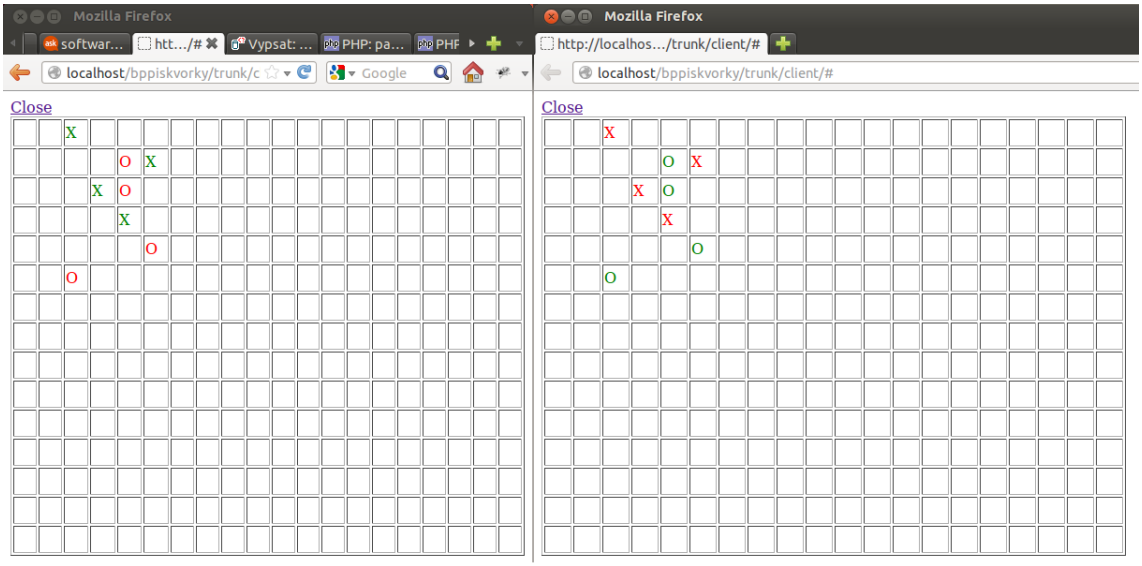
```
/* procedura vykreslí na danou pozici daný kurzor (x nebo o) */
void fill(row,col,cursor);

/* vypíše oznámení čekání na hráče */
void waitForPlayer();

/* odstraní oznámení čekání na hráče */
void playerReady();

/* nastaví kurzor hráče (x nebo o) */
void setCursor(cursor);
```

Listing 6.2: Dostupné procedury pro server v demonstrační aplikaci



Obrázek 6.1: Screenshot demonstrační aplikace.

na klientovi *playerReady* a hra může začít. Při kliku na hrací plochu je zavolána procedura serveru *click* se souřadnicemi políčka a proces tuto událost zaznamená do databáze. Proces spoluhráče toto zaznamená a zavolá proceduru *fill* se souřadnicemi a kurzorem a klient tah zakreslí na svou hrací plochu viz obrázek 6.1. Celý proces se opakuje.

Kapitola 7

Závěr

Cílem této práce bylo navrhnout a implementovat řešení pro transparentní klient–server komunikaci pomocí HTTP. Implementace klienta měla proběhnout v jazyce JavaScript, implementace serveru v jazyce PHP. Zadání se podařilo úspěšně splnit.

Začal jsem popisem HTTP protokolu, kde byly vysvětleny základní principy.

V další části je rozebráno vzdálené volání procedur v JavaScriptu a popsány 2 nejpoužívanější protokoly: XML–RPC a JSON–RPC. K vzdálenému volání procedur v JavaScriptu se v současné době využívá technologie AJAX, která umožňuje na pozadí komunikovat směrem od klienta na server.

Dále se zabývám popisem WebSocket protokolu a jeho použití. Jelikož vývoj ještě nebyl dokončen, může v budoucnu dojít k drobným změnám, ale předpokládám, že by žádné zásadní změny neměli nastat. Tato technologie odbourává nevýhody dosavadních řešení a to nemožnost zahájení komunikace ze strany serveru ke klientovi a poskytuje tak prostor k tvorbě interaktivnějších a náročnějších aplikací v protstředí webu.

V posledních 2 částech se zabývám samotným návrhem řešení problému a demonstrační aplikací. Řešení jsem postavil na technologiích WebSocket a JSON–RPC. Pro implementaci knihovny pro RPC jsem se rozhodl implementovat i jednoduchý WebSocket server, jelikož stávajících řešení na PHP je malé množství a všechny jsou postaveny na principu iterativního serveru. Moje řešení pracuje na principu konkurentního serveru a v některých případech může být zajímavou alternativou. Například u složitějších aplikací při zpracování náročných dat, by obshluha klientů pouze jedním procesem vedla pravděpodobně k dlouhým čekacím dobám ostatních klientů a aplikace by se tak stala do značné míry nekomfortní, např. online hry. Na druhou stranu, toto řešení nemusí být vhodné k použití na běžných webových stránkách s vysokou návštěvností, protože se při každém připojení klienta vytváří nový proces což zvětšuje zátěž serveru.

Demonstrační aplikace ukazje jeden z možných způsobů implementace. Jsou tam znázorněny možnosti obousměrného volání procedur, jak ze strany serveru, tak ze strany klienta. Zároveň tak došlo k otestování knihoven a ukázku využití.

Literatura

[1] Brinzarea-Iamandi, B.; Darie, C.; Hendrix, A.: *AJAX and PHP: Building Modern Web Applications*. Packt Publishing, druhé vydání, 2009, ISBN 978-1-847197-72-6.

[2] Deveria, A.: Can I use... [Online], [vid. 2013-05-05], Duben 2013.
URL http://caniuse.com/#feat=websockets

[3] Fette, I.; Melnikov, A.; Inc., G.; aj.: The WebSocket Protocol. [Online], [vid. 2012-10-20], Prosinec 2011.
URL http://tools.ietf.org/html/rfc6455

[4] Gourley, D.; Totty, B.; Sayer, M.; aj.: *HTTP: The Definitive Guide*. O'Reilly Media Inc., 2002, ISBN 978-1-56592-509-0.

[5] Group, J.-R. W.: JSON-RPC 2.0 Specification. [Online], [vid. 2012-10-20], Březen 2010.
URL http://www.jsonrpc.org/specification

[6] Hickson, I.; Inc., G.: The Web Sockets API. [Online], [vid. 2012-11-03], Říjen 2009.
URL http://www.w3.org/TR/2009/WD-websockets-20091029/

[7] Inc., U. S.: XML-RPC Specification. [Online], [vid. 2012-10-20], Červen 1999.
URL http://xmlrpc.scripting.com/spec

[8] Lubbers, P.; Albers, B.; Salim, F.: *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. Apress, 2010, ISBN 978-1-4302-2791-5.

[9] Shklar, L.; Rosen, R.: *Web Application Architecture: Principles, protocols and practices*. John Wiley & Sons Ltd, 2003, ISBN 0-471-48656-6.

Příloha A

Obsah CD

CD obsahuje:

- Technickou zprávu v PDF
- Zdrojové soubory technické zprávy v Latex
- Zdrojové soubory WebSocket a RPC serveru
- Zdrojové soubory demonstrační aplikace
- Soubor README

Příloha B

Manual

Požadavky

WebSocket server a potažmo celé řešení potřebují k provozu PHP verzi 5.3 a vyšší s nainstalovaným rozšířením PCNTL[1].

Demonstrační aplikace navíc potřebuje server Apache od verze 2 s možností spouštět PHP skripty, MYSQL serer od verze 5, knihovnu dibi[2] (je přibalena k programu) a předpokládá nainstalované POSIX funkce[3] v PHP.

Instalace

Instalace vyjma předhcozích požadavků není nutná, stačí soubory nahrát na disk a spustit.

Ovládání

Spuštění demonstrační aplikace provedeme z spuštěním souboru server.php umístěného ve stejnojmenné složce

```
php server/server.php
```

Tímto je spuštěný server. Klientskou aplikaci spustíme ze složky client/index.php ve dvou oknech prohlížeče. Nyní by měla být aplikace spuštěna a při kliknutí do hrací plochy, by se toto políčko mělo označit v druhém okně a naopak.

[1] http://php.net/manual/en/book.pcntl.php
[2] http://dibiphp.com/
[3] http://php.net/manual/en/book.posix.php