1 Úvod

Cieľom tejto práce je navrhnúť vhodnú technológiu na realizáciu webovej služby lokality distribuovaného laboratória počítačových sietí a preskúmať možnosti implemetácie klienta a servera tejto webovej služby v jazykoch C/C++ a PHP.

V prvej kapitole sa budem venovat teórii webových služieb, popíšem protokoly SOAP a XML-RPC, ich históriu, využitie a štruktúru XML dokumentov ktoré zasielajú. Budem sa zaoberať jazykom WSDL, ktorý je založený na XML a slúži na popis webovej služby, a jeho možnosťami.

Druhá kapitola bude zameraná na samotnú implementáciu klienta a servera v XML-RPC a SOAP, v jazykoch C/C++ a PHP, budú tu uvedené jednoduché príklady a návody ako ich realizovať, a budú tu charakterizované použité implmentácie.

Tretia kapitola zhrnie získané poznatky a skúsenosti a predloží výhody a nevýhody oboch technológií, z ktorých si čitateľ bude môcť spraviť vlastný obraz o tom, ktorá z technológi je vhodnejšia pre určité konkrétne použitie v praxi.

2 Teória webových služieb

2.1 Všeobecná charakteristika konceptu webových služieb

Webové služby všeobecne umožňujú klientským programom volať prostredníctvom nosného protokolu (najčastejšie HTTP[1]) funkcie na serveri webowej služby, posielať dokumenty a vymieňať informácie medzi klientským programom a servrom. Sú teda založené na architektúre klient-server. Webové služby umožňujú komunikáciu v heterogénnom prostredí, nezáleží teda na jazyku v ktorom je klientská alebo serverová aplikácia naprogramovaná. Pre posielanie parametrov, výsledkov a chybových hlásení sa používajú dokumenty formátu XML.

Sféra použitia webových služieb je veľmi široká – dajú sa použiť pri poskytovaní zoznamu tovaru e-shopu, predpovede počasia, pri posielaní výsledkov vyhľadávania, až po špecializované prípady použitia, kde spadá napríklad komunikácia medzi dvoma rovnocennými informačnými systémami.

Protokoly webových služieb definujú spôsob preposielania správ, formy popisu webovej služby, nosný prokokol, dátové typy, možnosti zabezpečenia a ďalšie veci potrebné k plnej funkčnosti a bezpečnosti prevádzky webovej služby. Dva najpoužívanejšie protokoly sú XML-RPC a SOAP. Oba sú založené na preposielaní dokumentov vo formáte XML. Tieto dokumenty sú relatívne jednoducho čitateľné pre ľudí – naproti tomu ich spracovanie spotrebuváva omnoho viac procesorového času v porovnaní s binárnymi protokolmi GIOP/IIOP, COBRA alebo DCOM.

Priblížme si trochu tieto protokoly. GIOP je abstraktný protokol prostredníctvom ktorého medzi sebou komunikujú dva ORB-y. ORB je program - prostredník, ktorý slúži na serializáciu a deserializáciu dát posielaných po sieti. IIOP je konkrétna implementácia GIOP protokolu pre TCP/IP siete. V binárnom výpise komunikácie sa GIOP dá spoznať podľa hlavičky obsahujúcej štyri ASCII znaky : G I O P.

CORBA je mechanizmus na štandardizáciu volaní metód medzi aplikačnými objektmi. Používa Interface Description Language (IDL) na špecifikovanie rozhraní ktoré budú prístupné k vzdialenemu volaniu. CORBA špecifikuje prevod z IDL do bežne používaných jazykov. Štandardné prevody sú k dispozícii pre jazyky Ada, C/C++, Lisp, Smalltalk, Java, Cobol, PL/I a Python. Existujú však aj neštandardné pre jazyky Perl, Visual Basic, Ruby, Erlang a Tcl, implementované pomocou ORBov napísaných pre tieto jazyky.

DCOM je patentovaná technológia od firmy Microsoft slúžiaca na komunikáciu medzi softwarovými komponentami distribuovaných v sieti medzi počítačmi. Rozširuje COM technológiu a poskytuje komunikačný základ pre COM+ aplikačnú architektúru. Rieši serializáciu a deserializáciu volaní, a distribuovanú správu pamäti. Pôvodne sa táto technológia nazývala „Network OLE“.

CORBA a IIOP sú registrované značky OMG (Object Management Group). GIOP však nie. OMG konzorcium je napríklad zodpovedné aj za jazyk UML.

1 Protokol SOAP môže využívať ako transportný protokol aj SMTP, existujú aj binárne protokoly

V ďalších kapitolách sa budeme venovať protokolom SOAP a XML-RPC z teoretickej stránky, ukážeme si jednoduché príklady ich implementácie v jazykoch PHP a C/C++, a v záverečnom zhrnutí porovnám oba protokoly.

2.2 XML-RPC

XML-RPC je pomerne jednoduchý protokol, slúžiaci na vzdialené volanie procedúr. Nie je to však samostatná technológia, ale len súbor pravidiel, ktoré hovoria ako používať už existujúce technológie pre potreby RPC. Funguje nad protokolom HTTP/HTTPS[2], pomocou ktorého prenáša dáta vo formáte XML. Umožňuje programom bežiacim pod rôznymi operačnými systémami a v rôznych prostrediach vzdialene volať funkcie pomerne jednoduchým spôsobom – telo požiadavku je XML súbor, procedúra sa vykoná na serveri, a odpoveď je opäť XML súbor, obsahujúci výsledok, alebo informáciu o chybe.

V porovnaní s architektúrou CORBA je XML-RPC jednoduchší protokol. CORBA vyžaduje značne sofistikované klientské programy, je to relatívne komplexný protokol, a je viac vhodný pre podnikové a desktopové aplikácie.

2.2.1 História

XML-RPC bolo vytvorené v roku 1998 Davidom Winerom z firmy UserLand v spolupráci s firmou Microsoft. Malo pôvodný pracovný názov SOAP. Postupne, ako pribúdali nové funkcie vyvinulo sa do dnešného protokolu SOAP, ktorým sa budeme zaoberať v ďalšej kapitole. XML-RPC zakladalo na staršom protokole firmy UserLand ktorý sa volal RPC. Oproti tomuto protokolu však umožňovalo použitie komplexných dátových typov, pracoval však podobne ako XML-RPC na základe protokolu HTTP prostredníctvom zasielania XML správ.

2.2.2 Popis požiadavky a odpovede

Požiadavok aj odpoveď majú dve časti – hlavičku a telo. Najprv sa budem venovat požiadavku. Jeho hlavička obsahuje v prvom riadku tri informácie:

- druh požiadavku, v prípade XML-RPC vždy POST
- URI, teda lokalizáciu serveru, nie je povinné
- verziu a druh protokolu, pre potreby XML-RPC sa používa HTTP verzie 1.0

Ďalšie riadky obsahujú vždy názov položky a jej hodnotu, a sú povinné:

2 Rozdiel v použití HTTP a HTTPS spočíva len v nastavení šifrovania ktoré obsluhuje HTTP sever, nie v implementácii konkrétnej služby

Položka	Popis	Povinnosť	Hodnota
User-Agent	identifikácia klienta, verzia implementác	povinné	`Frontier/5.1.2`
Host	adresa XML-RPC servera	povinné	`localhost`
Content-Type	MIME typ správy	povinné	`text/xml`
Content-length	udáva dĺžku správy v znakoch	povinné	181

Tabuľka 1: Hlavička požiadavky

Telo požiadavku je formátované jazykom XML. Ako úvodná značka sa sa musí použiť párová značka `<methodCall>`, dokument je teda ukončený značkou `</methodCall>`. Vnútri tejto značky je značka `<methodName>`, za ktorou nasleduje názov volanej procedúry (obsahujúci len malé a velké písmená, čísla, lomítko, podčiarnik, pomlčku a bodku) ukončený jej párovou značkou `</methodName>`. Zvyšok tela tvoria parametre potrebné pre zavolanie danej procedúry, uvedené značkou `<params>` a ukončené značkou `</params>`. Hodnota parametru je uzavretá v značkách `<value>` s nepovinnou definíciou dátového typu. V prípade, že volaná metóda nemá žiadne vstupné parametre, je vynechaná celá značka `<params>.`

Príklad XML-RPC požiadavku :

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: localhost
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>
```

Teraz sa bližšie pozrieme na odpoveď. Jej hlavička v prvom riadku obsahuje protokol a jeho verziu (vždy HTTP 1.1) a stavový kód. Ďalšie riadky majú obdobne ako u požiadavku skladbu názov položky a jej hodnotu :

Položka	Popis	Povinnosť	Hodnota
connection	určuje stav spojenia	povinné	close
Content-Length	udáva dĺžku správy v znakoch	povinné	١٢٤
Content-Type	MIME typ správy	povinné	text/xml
Date	dátum a čas zaslania správy	povinné	
Server	meno servra odosielajúceho odpoveď	povinné	localhost

Tabuľka 2: Hlavička odpovede

Telo odpovede je koncipované podobne ako telo požiadavku. Vonkajšia párová značka `<methodResponse>` obsahuje párovú značku `<params>`, ktorá je rovnako štruktúrovaná ako rovnomenná značka pri požiadavke. Na rozdiel od požiadavky musí ale odpoveď obsahovať túto značku, teda XML-RPC nedokáže volať metódy, ktoré nemajú žiadnu návratovú hodnotu. Je preto vhodné koncipovať svoje metódy tak, aby vracali aspoň návratovú hodnotu symbolizujúcu úspech alebo neúspech vykonanej procedúry.

Špeciálnym prípadom odpovede je chybová odpoveď, ktorá má za úlohu informovať klienta o chybe ku ktorej došlo na serveri. Jej hlavička je rovnaká ako pri normálnej odpovedi, ale jej telo je presne špecifikované. Namiesto párovej značky `<params>` obsahuje značku `<fault>`. V ňom je značka `<value>`, ktorá už obsahuje štruktúru s hodnotou chybového kódu, a textom chyby. Číslo chyby, až na pár výnimiek, nie je univerzálne pre všetky XML-RPC servre – teda záleží na programátorovi aké čísla použije pre aké chyby.

Príklad štandardnej odpovede (bez hlavičky):

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

2.2.3 Dátové typy v XML-RPC

XML-RPC definuje 6 jednorozmerných a 2 viacrozmerné dátové typy:

Značka	**Typ**
<i4> alebo <int>	32 – bitové celé číslo
<double>	číslo s pohyblivou desatinnou čiarkou
<boolean>	pravdivostná hodnota – *true / false*
<string>	textový reťazec
<dateTime.iso8601>	dátum a čas podľa normy ISO 8601
<base64>	binárne dáta v kódovaní BASE 64

Tabuľka 3: jednorozmerné dátové typy

Parameter typu string môže podľa špecifikácie na stránkach http://www.xmlrpc.com/spec obsahovať akékoľvek znaky okrem „<“ (je zakódovaný ako „<“) a „&“ (je zakódovaný ako „&“). Reťazec sa takisto môže použiť na zakódovanie binárnych dát.

Viacrozmerné dátové typy zastupuje pole – *array,* a štruktúra – *struct.* Pole sa definuje párovou značkou `<array>`, ktorá obsahuje párovú značku `<data>`. V nej sa nachádzajú

definície prvkov poľa, umiestnené vo vyššie spomínanej značke `<value>`. V XML-RPC indexy ani názvy položiek nepodporuje, prvkom poľa môže byť ale ďalšie pole alebo štruktúra.

Príklad poľa:

```
<array>
        <data>
                <value><i4>8</i4></value>
                <value><string>retazec</string></value>
                <value><boolean>false</boolean></value>
                <value><double>2100.45</double></value>
        </data>
</array>
```

Štruktúra má ako úvodnú značku `<struct>`, obsahujúcu prvky `<member>`. Každý prvok je zložený z mena - `<name>` a hodnoty - `<value>`. Prvkom štruktúry môže byť takisto pole alebo ďalšia štruktúra.

Príklad štruktúry:

```
<struct>
        <member>
                <name>polozka1</name>
                <value><string>text</string></value>
        </member>
        <member>
                <name>polozka2</name>
                <value><i4>24</i4></value>
        </member>
</struct>
```

2.2.4 Implementácie XML-RPC

Protokol XML-RPC sa v súčasnej dobe považuje za ukončený projekt. Je rozšírený do dostatočne veľkého množstva programovacích jazykov – na internete sa dajú nájsť implementácie pre každý hromadne používaný programovací jazyk – C/C++, COM, Delphi, Flash, Java, J2ME, JavaScript, Microsoft .NET, Perl, PHP, Python, Ruby, TCL a mnoho ďalších.

2.3 SOAP

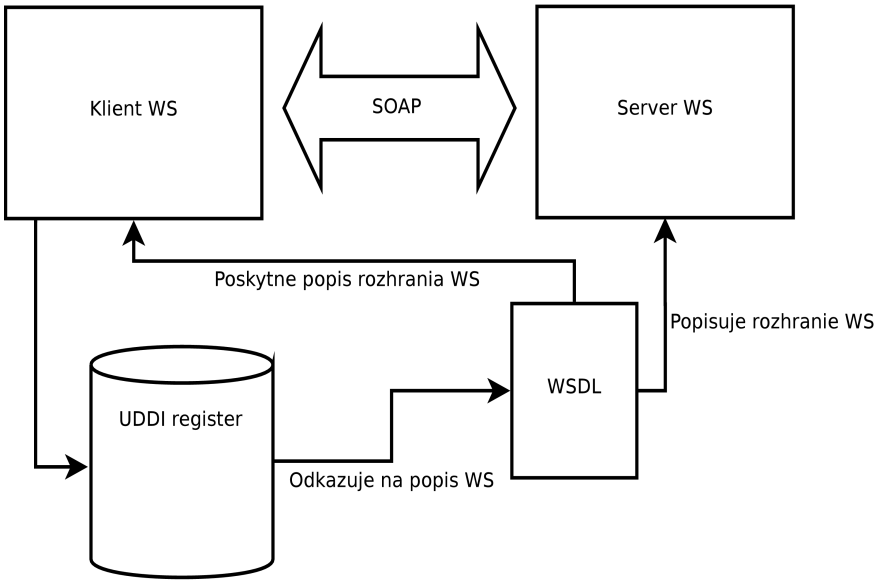
Protokol SOAP naväzuje na protokol XML-RPC, poskytuje však omnoho väčší potenciál pre používanie. Neslúži totiž už len na vzdialené volanie procedúr (rpc formát), ale aj na jednoduché zasielanie správ medzi aplikáciami (document formát). Jeho hlavná výhoda oproti konkurencii ako je DCOM alebo GIOP/IIOP je v použití HTTP/HTTPS a SMTP ako

transportných protokolov, pretože mu umožňujú prechod štandardne konfigurovaným firewallom , využíva totižto porty ktoré sú obyčajne otvorené (25 pre SMTP a 80 pre HTTP). Nevýhoda naopak spočíva v neefektívnosti pomeru prenášaných dát k reálnej komunikácii – jeho XML dokumenty sú pomerne objemné aj oproti dokumentom, ktoré používa XML-RPC.

2.3.1 História

Keď protokol SOAP v roku 1998 začínal, neexistoval žiadny schematický jazyk pre XML, formát (XML – RPC sa oddelil od SOAP práve v roku 1998). Pôvodný názov bola skratka – Simple Object Access Protocol – pri verzii 1.2 však toto prestalo platiť, pretože sa skratka považovala za mätúcu. Pôvodný protokol SOAP bol navrhnutý ako „object – access“ protokol, teda protokol na prístup k objektom, a prvé verejné vydanie bolo vlastne XML-RPC s mennými priestormi a dlhšími názvami elementov. Vznikol v spolupráci firiem UserLand, DevelopMentor a Microsoft. Neskôr však bol predaný pracovnej skupine konzorcia W3C.

2.3.2 Popis štruktúry webovej služby



Obrázok 1: Popis štruktúry WS

Webová služba, ktorá podporuje protokol SOAP môže byť popísaná súborom WSDL.
WSDL (Web Service Description Language) je jazyk založený na XML, ktorý slúži špeciálne na popis rozhrania webovej služby. V niektorých implementáciách SOAP (PHP) sa na základe

tohto popisu generuje aj samotný objekt klienta. Bližšiemu popisu jazyka WSDL sa budem venovať v nasledujúcej časti práce.

Popis webovej služby sa v otvorenom internete, alebo vo väčších systémoch môže zapísať do UDDI registra (Universal Description, Discovery and Integration). Register UDDI obsahuje odkazy na WSDL súbory jednotlivých služieb a tým podstatne uľahčuje prístup k webovým službám. Registrácia v registri UDDI má tri časti :

- biele stránky – adresa, kontakt na poskytovateľa
- žlté stránky – zaradenie služieb na základe štandardnej kategorizácie
- zelené stránky – technické informácie o poskytovaných službách

Jednoduchý princíp fungovania služieb spolu s UDDI registrom je znázornený na obr. 1.

2.3.3 Popis protokolu

Protokol SOAP naproti prokolu XML-RPC disponuje omnoho širšími možnosťami. Podľa definície v špecifikácii verzie 1.2 je SOAP jednoduchý protokol slúžiaci na výmenu štruktúrovaných informácií v decentralizovanom a distribuovanom prostredí. Používa XML na definovanie rozšíriteľného komunikačného rámca poskytujúceho štruktúru správ ktoré môžu byť vymieňané prostredníctvom množstva základných protokolov.

2.3.4 Popis požiadavky a odpovede

Základnou časťou správy je komunikačný rámec, ktorý definuje súpravu elementov XML slúžiacich na zabalenie ľubovolných XML správ. Rámec obsahuje štyri základné časti : obálku (Envelope), HTTP hlavičku (Header), chybovú hlášku (Fault) a telo (Body). Všetky tieto elementy pochádzajú z jedného menného priestoru http://schemas.xmlsoap.org/soap/envelope. Správa protokolu SOAP sa musí riadiť nasledujúcimi pravidlami :

- musí byť napísaná vo formáte XML
- musí používať SOAP envelope menný priestor
- musí používať SOAP encoding menný priestor
- nesmie obsahovať DTD (Document Type Definition)
- nesmie obsahovať inštrukcie na spracovanie XML

Vo WSDL dokumente popisujúcom službu určujeme aj kódovanie správ. Kódovanie správy definuje mapovanie objektov, premenných a programových typov do XML. SOAP podporuje viacero druhov kódovania dokumentu, ale hlavné, najpodporovanejšie a najpoužívanejšie sú RPC, RPC-Literal a Document-Literal. Pre programátora je najjednoduchšie RPC, pre systém je najjednoduchšie spracovanie správy vo formáte Document. Kódovanie RPC je definované špecifikáciou SOAP verzie 1.1, a pre programátora je výhodné pretože nemusí pred volaním samotnej procedúry na serveri získavať osobitne jednotlivé argumenty z požiadavky a predávať ich ako parametre pri volaní, ale robí to namiesto neho SOAP – automaticky prevedie podľa štruktúry XML dokumentu parametre na objekty a tie predá priamo

procedúre. Keďže programátor sám pozná svoje XML najlepšie, jeho vlastný kód prejde XML stromom oveľa efektívnejšie ako generalizovaný SOAP parser. Práve to je dôvod väčšej záťaže systému – spracovanie XML dokumentu pri kódovaní RPC spotrebuje viac systémových zdrojov, ale vyžaduje len malú, respektíve žiadnu podporu od programátora. Naproti tomu spracovanie požiadavku pri formáte kódovania Document sa od programátora vyžaduje aby sa postaral o kompletné spracovanie XML stromu, vyhľadávaniu v ňom, a konštrukciu odpovede. SOAP v tomto prípade zaobstará len odoslanie požiadavky, a dokonca ani nevyžaduje odpoveď serveru. RPC–Literal je niečo medzi oboma predošlými : RPC štýl vyžaduje aby bola v značke `<soap:Body>` uvedená značka s názvom volanej procedúry, ale jej parametre sa už neriadia žiadnym menným priestorom, teda o ich preklad sa musí postarať autor aplikácie. Dá sa použiť napríklad v prípade že parametre dostaneme zakódované v XML od inej aplikácie.

Požiadavka vo formáte RPC vypadá nasledovne :

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```

Vidíme, že správa obsahuje HTTP hlavičku a XML správu s koreňovým elementom `<soap:Envelope>`, ktorý obsahuje atribúty `xmlns:soap` a `soap:encodingStyle` odkazujúce na potrebné menné priestory. Vovnútri tohto elementu sa nachádza element `<soap:Body>` s atribútom `xmlns:m`, ktorý spolu s obsahom tohto elementu už nespadá do špecifikácie SOAPu, ale je špecifický pre danú aplikáciu. Konkrétne obsahuje meno metódy (GetStockPrice), parameter s ktorým ju voláme (StockName) a jeho hodnotu (IBM). Ak by mala byť táto správa zaslaná vo formáte Document – Literal, neobsahovala by žiadne menné priestory a o jej spracovaní a stavbe by rozhodol programátor.

Tu je príklad odpovede :

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-
```

```
encoding">

<soap:Body>
   <m:GetPriceResponse
           xmlns:m="http://www.w3schools.com/prices">
      <m:Price>1.90</m:Price>
   </m:GetPriceResponse>
</soap:Body>

</soap:Envelope>
```

Odpoveď má rovnakú štruktúru ako požiadavka, ale obsah elementu `<soap:Body>` je výsledok volanej metódy : `<m:Price>1.90</m:Price>`.

2.3.5 SOAP správy s prílohou

V mnohých prípadoch použitia webových služieb nestačí poslať len jednoduchý textový dokument, ale je potrebné preniesť celý súbor, či už multimediálny alebo iný. Existujú 2 štandardy prenosu takéhoto súboru – prvý je od W3C konzorcia, a ďalší od organizácie WS-I (Web Service-Interoperability). Organizácia WS-I pre tieto prípady definovala mechanizmus nazývaný SOAP with attachments (SwA) – SOAP s prílohou. Nepredstavuje novú špecifikáciu, ale metódu ako zasielať súbory používaním kombinácie SOAP a MIME, primárne cez HTTP. Využíva sa pri nej MIME formát `multi-part message`, ktorý určuje, že zasielaná správa bude pozostávať z viacerých častí s obsahom rôzneho typu. Jednu túto časť potom predstavuje SOAP XML dokument a ďalšie časti prenášané súbory. Podľa MIME špecifikácie každá časť správy má vlastnú hlavičku, a v nej definované Content-ID, pomocou ktorého sa na súbor odkazujeme zo SOAP dokumentu. V prílohe je príklad správy tohto typu.

Naproti tomu W3C konzorcium prišlo s riešením ktoré zavrhuje akékoľvek prílohy v správe mimo SOAP obálky, a prenos súboru podľa tohto štandardu prebieha priamo v SOAP správe. Vzhľadom na to že správa nemôže niesť binárne dáta, súbor sa zakóduje pomocou kódovania Base64. MTOP/XOP[3] definujú serializáciu takejto správy tak, že časti ktoré vyžadujú kódovanie sa serializujú osobitne a pripoja sa ako časť MIME balíčku.

Služba ktorá umožňuje prenášať súbory týmto mechanizmom, má ich spôsob ich prenosu definovaný v jej WSDL dokumente.

Nie každá implementácia SOAPu podporuje SwA. Implementácie podporujúce SwA sú napríklad SOAP with Attachments API for Java (SAAJ), podporuje ho aj Apache Axis 2, pre C++ je to gSOAP. Štandardné knižnice pre SOAP v PHP SwA nepodporujú, pre PHP nájdeme podporu v implementácii nuSOAP.

2.3.6 Dátové typy v SOAPe

3 MTOP/XOP (Message Transmission Optimization Mechanism and XML-binary Optimized Packaging) sú prostriedky na efektívnejšiu serializáciu XML dokumentov s rôznym obsahom

SOAP používa dátové typy XML schémy. Sú popísané v tabuľke v prílohe.

2.3.7 Implementácie SOAPu

Implementácie protokolu SOAP existujú pre dostatočný počet dnes najpoužívanejších programovacích jazykov. Implementácia Apache Axis je pre jazyky C++ a Java, Apache SOAP pre Javu, gSOAP pre C++, kSOAP pre JavuME. Jazyk PHP má integrované funkcie SOAPu, a podporuje SOAP verzie 1.2 rovnako ako ostatné spomínané implementácie. Nato aby bola podpora SOAP v PHP sprístupnená, musí byť PHP konfigurované s prepínačom `–enable-soap`. Všetky spomínané implementácie sú k dispozícii zdarma.

2.4 Štruktúra WSDL dokumentu, jazyk WSDL

WSDL je jazyk založený na XML, ktorý umožňuje popis webových služieb ako kolekcie koncových bodov, pracujúcich so správami ktoré obsahujú dokumenty alebo volania procedúr, a ich prístupu k nim. Ako sme si už povedali, WSDL dokument slúži na popis webovej služby. Je to však dosť jednoducho povedané. Vo WSDL dokumente môžeme pre službu definovať vlastné dátové typy, definujeme v ňom aj to, cez aké protokoly bude služba prístupná, metódy, ich parametre a návratové hodnoty.

Služba je popísaná pomocou typov, správ, operácií, pripojení, a koncových bodov. Pohľadom na WSDL dokument zistíme ze je vcelku zložité pochopiť, čo presne popisuje, čítanie odzadu však toto uľahčí. Jeho totižto štruktúrovaný presne opačným spôsobom, ako by sme očakávali – na začiatku sú tu popisy typov parametrov a pod nimi sú popisy komunikačných správ, kde využívame naše vlastné typy, ktoré sme si definovali vyššie, teda pre pochopenie toho čo vlastne obsahujú sme nútení vracať sa v zdrojovom kóde.

Takže štruktúra WSDL dokumentu je nasledujúca :

– koreňovým elementom celého dokumentu je element `<definitons>` ktorý ako atribúty obsahuje základné menné priestory pre daný dokument.

```
<definitons name="moja_sluzba"
xmlns="http://schemas.xmlsoap.org/wsdl/">
     <service>
     …
     </service>
</definitions>
```

– služba je popísaná v elemente `<service>`. V tomto elemente sú špecifikované porty (nie porty v chápaní TPC-IP protokolu) na ktorých je služba prístupná. Jedna služba môže byť prístupná na viacerých portoch – jeden cez HTTP, iný cez SMTP. Potom pre túto službu existujú dva porty, každý s iným menom. Každý port má teda priradené unikátne meno a atribút binding. V prípade používania SOAP je v elemente port element `<soap:address>` udávajúci aktuálnu adresu na ktorej je služba prístupná.

```
<port name="moj_port" binding="moj_binding">
```

```
        <soap:address location="http://adresa_sluzby">
    </port>
```

- element `<binding>` viaže spolu viaceré operácie, tak aby sme ich neskôr mohli sprístupniť na jednom porte. Taktiež pri každej operácii určuje vstup a výstup – aké kódovanie správy sa použije. V tomto elemente máme taktiež definovaný transportný protokol (v tomto prípade je to HTTP) a pri každej operácii aribút `soap:Action` ktorým určujeme HTTP hlavičku ktorú klient pošle keď sa bude snažiť zavolať službu. Operácia obsahuje ešte popis vstupu a výstupu operácie, nie však jej správy, ale štýlu kódovania a menného priestoru.

```
<binding name="moj_binding" type="moj_portType">
      <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="moja_metoda">
            <soap:operation
      soap:Action="http://adresa_sluzby/akcie/moja_metoda">
            <input>
                  <soap:body use="encoded"
                  namespace="http://adresa_sluzby/sprava"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
            </input>
            <output>
                  <soap:body use="encoded"
                  namespace="http://adresa_sluzby/sprava"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
            </output>
      </operation>
</binding>
```

- jednotlivé správy sú reprezentované elementom `<message>`, ktorý môže obsahovať elementy `<part>` - teda časti. Sú to vlastne v programátorskom ponímaní parametre volania alebo výsledok volanej funkcie. Tu vidíme rozdiel oproti XML-RPC – správy SOAP protokolu môžu ale nemusia obsahovať parametre. Je teda možné previesť volanie procedúry ktorá žiadny parameter nepotrebuje a žiadnu návratovú hodnotu nevracia. Každá správa má unikátne meno – udané ako atribút elementu `<message>`. Meno správy môže byť akékoľvek, WSDL špecifikácia nedefinuje žiadnu konvenciu na pomenovávanie správ, ak však používate nástroj na generovanie WSDL dokumentu, je pravdepodobné že tento má vlastný systém pomenovávania správ.

```
<message name="moja_sprava_volanie">
      <part name="parameter_1" type="xsd:string"/>
      <part name="parameter_2" type="moj_typ"/>
</message>
<message name="moja_sprava_odpoved">
      <part name="navratova_hodnota" type="xsd:int"/>
</message>
```

- operácie – element `<operation>` - reprezentujú jednu ucelenú metódu – definujú vstup

a výstup volanej metódy. Tento element môže obsahovať atribút `parameterOrder`, ktorý je nepovinný a určuje poradie parametrov pri volaní metódy. Vstup je definovaný v elemente `<input>` ktorého atribút `message` obsahuje meno vopred opísanej správy – `message`. Výstup je definovaný elementom `<output>` ktorý rovnako obsahuje meno výstupnej správy. Operácie zaobaľuje element `<portType>` ktorý má takisto svoj atribút `name`.

```
<portType name="moj_portType">
      <operation name="moja_metoda"
            parameterOrder="parameter_1 parameter_2"
            <input message="wsdlns:moja_sprava_volanie"/>
            <output message="wsdlns:moja_sprava_odpoved"/>
      </operation>
      …
</portType>
```

– elementy `<part>` obsahujú atribút `<element>` .Tieto elementy sú popísané vovnútri elementu `<types>` a reprezentujú dátové typy. Tu sa naskytuje možnosť definovať si vlastné dátové typy – štruktúry, polia. Element `<types>` ale nie je povinný, pretože ak v elementoch `<part>` použijeme len natívne XSD dátové typy, nemusíme tieto deklarovať.

```
<types>
      <schema    targetNamespace="http://adresa_sluzby/"
                 xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="moj_typ">
                 <complexType>
                      <element name="el_1" type="xsd:int"/>
                      <element name="el_2">
                           <complexType>
                                 <complexContent>
                      <restriction base="soapenc:Array">
                          <attribute ref="soapenc:arrayType"
                                wsdl:arrayType="string[]"/>
                      </restriction>
                                  </complexContent>
                           </complexType>
                      </element>
                 </complexType>
            </element>
      </schema>
</types>
```

Ak má špecifikovaná služba poskytovať možnosť prenášať súbory ako prílohy, musíme importovať MIME schému, a daný element uviesť nasledovným spôsobom :

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://adresa_sluzby/"
         xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
```

```
        <import
        schemaLocation="http://www.w3.org/2005/05/xmlmime"
        namespace="http://www.w3.org/2005/05/xmlmime"/>
        <element name="imagejpeg"
  xmime:contentType="image/jpeg" type="xmime:base64Binary"/>
</schema>
```

3. Ukážky implementácie základného klienta a serveru

V tejto kapitole sa budem venovať praktickej implementácii klienta a servra XML-RPC a SOAP protokolu pre jazyky PHP a C/ C++. Pôjde o jednoduchú službu poskytujúcu jednu metódu – sčítanie dvoch celých čísel. V texte budem popisovať len podstatné časti kódu, kompletný kód bude vždy uvedený v prílohe bakalárskej práce.

3.1 XML-RPC

3.1.1 C++

Pre implementáciu príkladu v C++ použijem implementáciu nazvanú XmlRpc++, napísanú v C++ a založenú na py-xmlrpc knižnici. Posledná zverejnená verzia 0.7 bola uvoľnená 6. 3. 2008, a túto verziu použijem. Táto implementácia je prenositeľná a voľne dostupná. Bližie informácie o tomto projekte ako aj fórum nájdete na http://sourceforge.net/projects/xmlrpcpp.

Ideálna inštalácia tejto implementácie na Linux pozostáva z nasledujúcich krokov:

- stiahneme si archív s potrebnými súbormi z internetových stránok projektu
- rozbalíme archív a spustíme v koreňovom adresári program `make`. V tejto verzii nastáva pri kompilovaní problém s ukážkovým programom `Validator.cpp`, ale túto chybu je možno ignorovať, nakoľko sa nejedná o súbor potrebný k používaniu knižníc
- vytvoríme adresár `/usr/local/include/xmlrpc++` a nakopírujeme doň obsah adresáru src z adresára kde sme rozbalili stiahnutý archív
- skopírujeme knižnicu `XmlRpc.a` do adresára `/usr/local/lib/`

Naproti iným implementáciám pre C++ je použitie mnou zvolenej jednoduchšie. Nevyžaduje žiadne iné knižnice okrem sieťových systémových knižníc. Má zabudované jednoduché spracovávanie XML dokumentov a podporu HTTP a malé API.

Ako prvý uvediem príklad XML-RPC klienta. Bude schopný zavolať XML-RPC webovú službu na sčítanie dvoch čísel. Pri spustení udávame 2 parametre – adresu a port na ktorej beží server služby. Jeho zdrojový kód je v prílohe. Tu si popíšeme dôležité časti kódu.

Začneme kódom klientského programu. Dôležité časti programu sú:

– vytvorenie objektu samotného klienta

```
XmlRpcClient client(adresa_serveru, port);
```

Vytvorí objekt klienta ktorý bude volania smerovať na server na adrese `adresa_serveru` a porte `port`.

– deklarovanie premennej pre vstupné parametre a pre výsledok volania metódy

```
XmlRpcValue args, result;
```

Vytvorí dve premenné schopné niesť akúkoľvek hodnotu používanú v XML-RPC. Pre priradenie hodnoty do premennej použijeme normálny priradzovací príkaz a s premennou pracujeme ako s poľom :

```
args[0] = 2;
args[1] = „retazec“ ;
```

– zavolanie metódy na serveri

```
client.execute("metoda", argumenty, navratova_hodnota);
```

Metóda `execute` zavolá na serveri metódu `metoda` so vstupnými argumentmi `argumenty` a výsledok uloží do premennej `navratova_hodnota`. Samotná metóda `execute` vracia pravdivostnú hodnotu 1 ak volanie prebehlo v poriadku a 0 ak nastala vo volaní chyba.

– v tejto implementácii máme možnosť volať viac metód jedným príkazom.

```
XmlRpcValue multicall, result;
multicall[0][0]["methodName"] = "metoda_1";
multicall[0][0]["params"][0] = parameter_prvej_metody;
multicall[0][0]["params"][1] = iny_parameter_prvej_metody;
multicall[0][1]["methodName"] = "metoda_2";
multicall[0][1]["params"][0] = parameter_druhej_metody;
client.execute("system.multicall", multicall, result);
```

Príkaz zavolá na serveri všetky metódy ktoré boli uložené do premennej `multicall` a ich výsledok uloží do premennej `result`.

Zdrojový kód servera pozostáva z hlavnej metódy `main` a metód ktoré poskytuje. V metóde `main` nastavíme parametre serveru a spustíme ho, teda server začne na danom porte načúvať a spracuvávať a odpovedať na požiadavky klientov.

– vytvorenie objektu serveru

```
XmlRpcServer server;
```

– nastavenie úrovne zapisovania udalostí do logu. Nastavujeme hodnoty od 0 po 5, kde 0 je vypnuté a 5 znamená zapisovanie každej udalosti.

```
XmlRpc::setVerbosity(5);
```

– vytvorenie serveru na žiadanom porte

```
server.bindAndListen(port);
```

– nastavenie introspekcie

```
server.enableIntrospection(true);
```

– samotný začiatok činnosti servera. Ako parameter typu double sa udáva čas v milisekundách, po ktorý bude server odpovedať na požiadavky klientov. Pri zadaní záporného čísla bude pracovať až pokiaľ program neskončí na základne vonkajšej udalosti.

```
server.work(-1.0);
```

Metódy ktoré sú serverom poskytované sú udávame ako samostatnú triedu obsahujúcu jedinú metódu `execute` s dvoma parametrami typu `XmlRpc` – jeden bude obsahovať pole vstupných hodnôt, do druhého uložíme pole výstupných hodnôt. Každá metóda musí byť zaregistrovaná. Funkcia na súčet čísel vypadá nasledovne :

```
class a_plus_b : public XmlRpcServerMethod
{
public:a_plus_b(XmlRpcServer*s ) :
            XmlRpcServerMethod("a_plus_b", s) {}
      void execute(XmlRpcValue& params, XmlRpcValue& result)
      {
            int nArgs = params.size();
            double sum = 0.0;
            for (int i=0; i<nArgs; ++i)
            sum += double(params[i]);
            result = sum;
      }
} a_plus_b(&server);
```

3.1.2 PHP

Pre programovací jazyk PHP využijem knižnice projektu „XML-RPC for PHP“. Táto knižnica je navrhutá vcelku jednoducho a cieľom projektu je aj rýchlosť a nízka spotreba pamäti. Použijem najnovšiu verziu – 2.2.1 – ktorá bola zverejnená 6. 3. 2008. Stránky projektu sú http://phpxmlrpc.sourceforge.net/. Podporuje PHP verzie 4 a vyššie.

Klientský program je veľmi jednoduchý. Narozdiel od implementácie XML-RPC protokolu v C++ tieto knižnice vyžadujú zložitejšiu prácu s návratovými hodnotami. Popíšme si dôležité časti kódu -

– vytvorenie objektu klienta

```
$client = new xmlrpc_client('/xmlrpc/server.php' ,
                                  'localhost' , 80);
```

Vytvorí klienta pre XML-RPC na stroji `localhost`, adrese `/xmlrpc/server.php` a porte `80`.

– vytvorenie objektu správy

```
$message = new xmlrpcmsg('aplusb',array(new xmlrpcval(5,
                      'int'),new xmlrpcval(3,'int')));
```

Vytvorí správu ktorá bude volať metódu `aplusb` a ako parameter jej predá pole s dvoma hodnotami – 5 a 3. Hodnoty však nepridávame ako obyčajnú premennú ale vytvoríme ich ako objekt `xmlrpcval`, a ako parametre konštruktoru predáme požadovanú hodnotu a jej typ, v tomto prípade `int`.

– zaslanie požiadavky na server

```
$result = $client->send($message);
```

Zašle predom vytvorenú správu `message` a návratovú hodnotu volania uloží do premennej `result`.

– spracovanie návratovej hodnoty

```
$sum_val = $result->value();

$sum = $sum_val->scalarval();

print $sum;
```

Prvým príkazom do premennej `sum_val` uložíme výslednú hodnotu typu `xmlrpcval` z premennej `result` ktorá predstavuje odpoveď serveru. V druhom príkaze metóda `scalarval` slúži na získanie skalárnej hodnoty z odpovede servra. Typ premennej rozoznávame pomocou metódy `kindOf` ktorá vráti reťazec s hodnotou „scalar“, „struct“ alebo „array“. Typ skalárnej premennej rozoznávame pomocou metódy `scalarTyp` ktorej návratová hodnota je reťazec s typom premennej. Ak je v premennej uložená štruktúra, na získanie jej prvkov použijeme metódu `structEach` ktorá vracia pole s dvoma prvkami – názvom prvku štruktúry a jeho hodnoutou. V prípade poľa zistíme najprv veľkosť poľa pomocou metódy `arraySize` a potom môžeme napríklad v cykle prechádzať celé pole – metóda `arrayMem` vracia premennú typu `xmlrpcval`. Ako parameter jej predáme číslo žiadaného prvku.

Vzhľadom na to, že XML-RPC sa odporúča používať v prípade že kontrolujeme stranu servra aj stranu klienta, sami vieme čo nám server v odpovedi vracia, a teda by nemalo byť potrebné kontrolovať návratové typy.

Ak bol však `faultCode`[4] odpovede nenulový, tak `value` nemusí byť objektom a preto nie je správne s nou pracovať. Preto by predošlý kód mala predchádzať táto podmienka :

```
if ( $result->faultCode()==0) {
```

V prípade že je `faultCode` nenulový, `$result->faultString()` nám vráti chybovú hlášku serveru.

Kód servera nie je o nič zložitejší. Obsahuje len vytvorenie objektu serveru kde zároveň

4 Návratová hodnota tejto funkcie sa zhoduje s hodnotou v značke `<value>`

registrujeme poskytované funkcie, a samotné funkcie vracajúce objekt typu `xmlrpcresp`.

Jediným podstatným úryvkom kódu je teda vytvorenie objektu servera.

```
$server = new xmlrpc_server(array('aplusb' =>
          array('function' => 'aplusb')));
```

Ako parameter konštruktoru predáme asociatívne pole, ktoréh prvky sú dalšie polia. Názov prvku je pseudonymom reálnej funkcie ktorá je uvedená v k nemu asociovanom poli v prvku pod názvom „function“. Ak je funkcia v rovnakej triede ako volanie konštruktoru, udáva sa len jej názov, ale v prípade, že je funkcia v inej triede ako konštruktor, udáva sa v tvare `trieda_funkcie::nazov_funkcie`.

Funkcia má 2 typické rysy : vracia objekt typu `xmlrpcresp` a ako parameter dostane objekt typu `xmlrpcmsg`. Pre získanie n-tého parametru typu `xmlrpcval` voláme na tento objekt metódu `getParam($n)`. Pre spracovanie premennej typu `xmlrpcval` používame rovnaké metódy aké som popísal pri klientovi.

3.2 SOAP

3.2.1 C++

Pre programovací jazyk C++ je najpoužívanejšou implementáciou protokolu SOAP implementácia gSOAP. Táto implementácia funguje na princípe generovania kódu. Obsahuje 2 nástroje – jeden na generovanie kódu hlavičkového súboru z WSDL súboru služby, a druhý na generovanie XML súborov, programových súborov a hlavičkových súborov potrebných pri kompilovaní klienta alebo serveru z hlavičkového súboru služby. Môžeme teda z WSDL súboru vygenerovať takmer kompletnú webovú službu, kde treba doplniť len výkonný kód jednotlivých poskytovaných funkcií. GSOAP patrí k najrýchlejším implementáciám, je až 30 krát rýchlejší ako Apache Axis, čo je implementácia pre Javu. Nepoužíva totiž generalizovaný XML parser, ale volania má pevne dané v kóde. Toto umožnilo práve generovanie kódu z hlavičkového súboru – generátor totiž vie, čo aké budú volania a odpovede servera, a dynamicky sú do nich dosadzované už len potrebné hodnoty.

Hlavičkový súbor služby obsahuje okrem štandardného C++ kódu aj direktívy pre gSOAP.

Po rozbalení archívu nájdeme v adresári bin 2 programy – `soapcpp2` a `wsdl2h`. Prvý program generuje z hlavičkového súboru programu služby všetky potrebné kódy okrem `stdsoap.h` a `stdsoap.cpp`, ktoré sú k dispozícii v stiahnutom archíve. Vygeneruje aj XML súbory, akési šablóny posielaných volaní a odpovedí. Druhý program generuje z WSDL popisu služby jej hlavičkový súbor, ktorý môžeme podať ako vstup prvému programu.

Hlavičkový súbor obsahuje okrem štandardného C++ kódu aj direktívy pre gSOAP generátor kódu (`wsdl2h` ich automaticky pridáva). Sú to komentáre so špeciálnou syntaxou, ktorú rozoznáva len gSOAP, a sú nastavené podľa ich ekvivalentu v spracuvávanom WSDL

súbore. Skladajú sa z dvoch lomítok (začiatok komentára), slova „gsoap“, predpony menného priestoru, ktorá slúži na spojenie našej služby s menným priestorom, a používame ju aj v názvoch funkcií, názov hodnoty ktorú chceme nastaviť a konkrétnu hodnotu oddelenú od jej názvu dvojbodkou. Vymenujem a popíšem tie základné :

- `service name` - určuje názov webovej služby, ekvivalent atribútu name v značke definitons vo WSDL

```
//gsoap ns service name: meno_sluzby
```

- `type` – vlastný typ služby, ekvivalent názvu značky porttype vo WSDL

```
//gsoap ns service type: typ_sluzby
```

- `location` – umiestnenie služby (URI) spolu s portom a cestou

```
//gsoap ns service location: cesta_k_sluzbe
```

- `documentation` – dokumentácia služby, popis

```
//gsoap ns1 service documentation: dokumentacia
```

- `method-style` - štýl volania metódy, ktorej názov je umiestnený bezprostredne za dvojbodkou, a samotný štýl nasleduje oddelený medzerou

```
//gsoap ns service method-style: aplusb rpc
```

- `method-encoding` – URI určujúce kódovanie volaní a odpovedí metódy

```
//gsoap ns1 service method-encoding: aplusb
                  http://schemas.xmlsoap.org/soap/encoding/
```

Ku kompletnej webovej službe však (ako už bolo povedané) nestačí len generovaný kód, je potreba aj výkonný kód funkcií a metód. Pozrime sa naňho bližšie.

Všeobecne môže webová služba bežať ako samostatná aplikácia, alebo ako CGI skript pod webovým serverom. Varianta CGI skriptu je omnoho jednoduchšia. Okrem už spomínaných výkonných metód ktoré služba poskytuje, je tu len jedna jednoduchá metóda `main`, ktorá obsahuje kód :

```
return soap_serve(soap_new());
```

ktorý zabezpečí, že bude vytvorený objekt, ktorý sa postará o celú ďalšiu činnosť zaoberajúcu sa prišlým volaním. Názvy metód, ktoré poskytujeme svetu majú tvar :

```
int predpona_menneho_priestoru__nazov_metody( argumenty ) {
      //kod
}
```

Prvý argument je vždy ukazovateľ na štruktúru typu `soap`, a posledný ukazovateľ na premennú do ktorej sa uloží návratová hodnota. Ďalšie argumenty sú závislé od metódy a nesú parametre predané serveru od klienta.

Každá metóda musí v prípade úspechu vracať preddefinovanú hodnotu `SOAP_OK`, a v prípade neúspechu volá metódu `soap_sender_fault` s prvým parametrom menom ukazovateľa na štruktúru typu `soap` a s ďalšími, ktoré budú obsahovať reťazec nami vytvorenej chybovej hlášky, ktorá bude poslaná klientovi.

Kód servera, ktorý má fungovať ako samostatne bežiaca aplikácia, je rozdielny oproti predchádzajúcemu kódu len v metóde `main`, ktorá je oproti predchádzajúcemu prípadu zložitejšia.

```
int main()
{
  int m, s;
  int port = 9090;
  struct soap *soap = soap_new();
  m = soap_bind(soap, NULL, port , 100);
if (m < 0)
{ soap_print_fault(soap, stderr);
 exit(-1);
}
  for (;;)
  {
    s = soap_accept(soap);
    if (s < 0)
    {
      soap_print_fault(soap, stderr);
      break;
    }
    fprintf(stderr, "spojenie uspesne\n");
    soap_serve(soap);
    soap_end(soap);
  }
  soap_done(soap);
  free(soap);
  return 0;
}
```

Teraz si popíšme kód. Najprv si vytvoríme štruktúru ktorá bude neskôr slúžiť na obsluhu volaní. V ďalšom kroku si do štruktúry `soap` pomocou funkcie `soap_bind` uložíme port na ktorom bude server naslúchať, a vo štvrtom argumente udáme maximálny počet klientov vo fronte čakajúcich na obslúženie. Tretí argument udáme nulový, pretože slúži na určenie cieľovej adresy v prípade že sa funkcia použije na strane klienta.

Ďalej v nekonečnom cykle obsluhujeme klientov – čakáme na pripojenie v metóde `soap_accept`, metóda `soap_serve` obslúži práve pripojeného klienta a metóda `soap_end` ukončí aktuálne spojenie. V prípade skončenia cyklu ukončíme spojenie a uvoľníme pamäť pomocou metód `soap_done` a `free`.

Kód klienta je jednoduchý, a využiva proxy triedy generované programom `soapcpp2` ktoré umožňujú vytvoriť objekt, schopný volať metódy poskytované servrom. Ich návratová hodnota zodpovedá tým na serveri, teda buď hodnotu SOAP_OK alebo chybový kód.

```
int main()
{
  calc c;
  double n;
  if (c.add(2, 3, &n) == SOAP_OK)
    cout << "2 plus 3 is " << n << endl;
  else
    soap_print_fault(c.soap, stderr);
  return 0;
}
```

Najprv si vytvoríme objekt typu `calc`, ktorý reprezentuje SOAP server, a ďalej na ňom zavoláme metódu `add`.

Pre kompiláciu klienta a servera použijeme nasledovné príkazy (v Linuxe pomocou kompilátora g++)

```
g++ -o client client.cpp soapC.cpp soapClient.cpp
stdsoap2.cpp

g++ -o server.cgi server_cgi.cpp soapC.cpp soapServer.cpp
stdsoap2.cpp

g++   -o server   server.cpp   soapC.cpp   soapServer.cpp
stdsoap2.cpp
```

3.2.2 PHP

PHP obsahuje rozšírenie pre protokol SOAP. PHP musí však byť nainštalované s voľbou –enable-soap. Rozšírenie pracuje na vyššej úrovni ako XML-RPC knižnice pre PHP.

Celkovo hodnotím prácu s protokolom SOAP v programovacom jazyku PHP ako najjednoduchšiu. Dokazujú to aj pomerne krátke a prehľadné ukážkové kódy. Výhoda spočíva napríklad vo vytvorení objektu serveru alebo klienta priamo z WSDL súboru. Na strane serveru po zaregistrovaní funkcií jednou metódou môže server plnohodnotne fungovať, a na strane klienta po vytvorení objektu klienta z WSDL súboru služby môžeme okamžite volať funkcie servra.

Objekty servera a klienta môžu byť vytvorené pomocou WSDL súboru, ale oba objekty obsahujú aj konštruktory schopné sfunkčniť komunikáciu medzi klientom a serverom aj bez popisu služby.

```
<?php
   $client =new SoapClient( $wsdl_url , $options );
   echo $client->a_plus_b($a,$b);
?>
```

Kód je jednoduchý, ako parameter predávame len adresu WSDL súboru a pole obsahujúce ďalšie voľby pre vytvorenie. Ak chceme zistiť, aké metódy server poskytuje, zavoláme na objekte klienta metódu `getFunctions`, ktorá vracia pole obsahujúce funkcie servera.

Kód servera je obdobne jednoduchý :

```
<?php
 function a_plus_b($a, $b) {
   return $b + $a;
 }
 $server = new SoapServer($wsdl_url, $options);
 $server->addFunction(SOAP_FUNCTIONS_ALL);
 $server->handle();
?>
```

Obdobne ako pri klientovi vytvoríme objekt servera z WSDL súboru predaného ako prvý parameter, a poľa ktoré obsahuje dalšie možné voľby. Pri serveri aj klientovi v prípade že vytvárame ich objekt pomocou WSDL súboru, parameter s voľbami je nepovinnný. V prípade, že chceme použiť konštruktory bez WSDL, je tento parameter povinný, a musí obsahovať aspoň lokáciu služby a jej menný priestor, teda parametre `location` a `uri`. Metóda addFunctions určí, ktoré metódy budú poskytované službou. Pri použití konštanty SOAP_FUNCTIONS_ALL budú použité všetky funkcie prístupné v danom kóde. V prípade, že chceme použiť funkcie obsiahnuté v triede, nastavíme ju príkazom setClass, ktorému predáme ako parameter názov triedy. Pri klientovi sú ďalšie nepovinné parametre `proxy_host`, `proxy_port`, `proxy_login` a `proxy_password` pri použití proxy serveru pre volania, `local_cert` a `passphrase` pre HTTPS autentifikáciu, `login` a `password` pre HTTP autentifikáciu, `compression` pre kompresiu volaní alebo napríklad `soap_version` pre špecifikovanie verzie protokolu. Pri konštruktori serveru je povinný parameter pri nepoužití WSDL len URI serveru.

Tu by som chcel upozorniť na prípadnú stratu času pri vývoji, respektíve ladení služby. V súbore `php.ini` je nastavenie v odstavci `[soap] wsdl_cache_enabled`, ktoré je ako predvolené nastavené na 1, teda pri pokusoch vytvoriť klienta WS po pár sekundách alebo minútach znova v domení že sme našu chybu vo WSDL súbore našli a opravili, dochádzalo k vytvoreniu objektu klienta nie z udaného URL na ktorom sa WSDL súbor nachádzal, ale z vyrovnávacej pamäte, v ktorej sa nachádzal pravdepodobne ešte pôvodný WSDL súbor s chybou. Je treba pre ladenie nastaviť túto možnosť na hodnotu 0.

Podrobnejší popis týchto a ďalších metód súvisiacich so SOAP protokolom odporúčam manuál PHP na stránkach http://www.php.net.

3.2.4 Implementácia webovej služby v PHP pre potreby Virtlabu

V zadaní bakalárskej práce bolo určené vypracovať niekoľko príkladov použitia webovej

služby v projekte Virtlab. Po dohode s vedúcim bakalárskej práce som vypracoval klienta aj server webovej služby v jazyku PHP s nasledujúcimi funkciami :

- select

  Funkcia všeobecného výberu hodnoty z databázy Virtlabu

- selectTable

  Funkcia všeobecného výberu riadkov a stĺpcov z tabuľky v databázi Virtlabu

- userList

  Vráti zoznam užívateľov

- userInfo

  Vráti informácie o konkrétnom užívateľovi

- userAutentize

  Overí heslo užívateľa v databázi

- getTaskCategories

  Vracia zoznam kategórií úloh

- getTaskList

  Vracia zoznam úloh na základe žiadanch kateórií.

Webová služba v čase písania tohto textu ešte nebola integrovaná v systéme Virtlab. Ukážky naimplementovaných funkcií nájdete v prílohe bakalárskej práce.

4. Záver

Cieľom tejto práce bolo preskúmať možnosti implementácie klienta a servera webovej služby pre potreby projektu Virtlab, implementovať ukážkové príklady a v spolupráci s vedúcim práce implementovať niekoľko príkladov využitia webových služieb v tomto projekte.

Hlavnou úlohou webových služieb jednotlivých lokalít Virtlabu je zdieľat informácie o užívateľoch a úlohách tak, aby došlo k efektívnejšiemu a jednoduchšiemu prístupu k sieťovým prvkom. V praxi by to znamenalo možnú prihlásenie užívateľa cudzej lokality na inej lokalite a jeho prácu v nej, pričom jeho autentifikácia prebehne prostredníctvom zvoleného protokolu webových služieb. Rovnako by si lokality medzi sebou mohli zdieľať ďalšie informácie, napríklad naplánované úlohy.

Pri rozobraní problému nastáva viacero otázok – ktorý protokol zvoliť, ktorú implementáciu zvoliť, ako riešit šifovanie prenášaných dát alebo ako zariadiť autentifikáciu lokalít medzi sebou. Šifrovanie dát je v tomto prípade vcelku dôležité práve pri prihlasovaní užívateľov, kedy sa kvôli používanej technológii XML prenáša heslo v textovom formáte, a mnoho užívateľov ostravskej lokality používa heslo zo školského LDAP serveru. Riešenie tohto problému by som videl v použití HTTPS ako transportného protokolu, teda dáta by boli prenášané šifrovane.

Pri voľbe protokolu webových služieb treba zvážiť fakt, že v zadaní navrhovaný protokol SOAP je pri dnešných požiadavkách na funkčnosť a vlastnosti webových služieb jednotlivých lokalít príliš zložitý a komplexný, nehovoriac o tom, že štandardizácia tohto protokolu je neukončená, nástroje používané pri implementáciách boli podľa mojich skúseností nedokonalé, niektoré dokonca navzájom nekompatibilné (WSDL dokument generovaný GUI nástrojom v Eclipse podľa iného nástroja nespĺňal špecifikácie), jediná open-source implementácia pre C/C++ ktorá mi pripadla použiteľná (gSOAP) bola omnoho zložitejšia ako je podľa môjho názoru vhodné (funkcie a objekty v PHP rozšírení dokážu spĺňať rovnakú úlohu a sú neporovnateľne jednoduchšie). Veľké zmeny vo filozofii SOAPu, jeho smerovaní a špecifikáciách tiež nevyhovujú tak špecifickému použitiu. WSDL popisy služieb jednotlivých lokalít sú viacmenej zbytočné, nakoľko ich služby nebudú využívané verejne, ale budú ich využívať len lokality medzi sebou. Naproti tomu protokol XML-RPC sa považuje za ukončený projekt, a tak sa dá očakávať, že implementácie ktoré pre tento protokol existujú neprejdú takým vývinom, ktorý by mohol narušiť vývoj a funkciu webových služieb Virtlabu. Protokol XML-RPC takisto môže naraziť na problémy v prípade, že knižnice ktoré využíva prestanú byť aktualizované a napríklad v novej verzii PHP nemusí fungovať dôležitá funkcia.

Odpoveď na otázku, akú implementáciu zvoliť je vcelku jasná pri protokole SOAP – v PHP je jeho podpora integrovaná a gSOAP je jediná širšie podporovaná voľná implementácia. Pri protokole XML-RPC a jazyku PHP je mnou použitá imlementácia dostatočná na realizovanie potrebných funkcií, a pri jazyku C++ takisto.

Zoznam použitej literatúry

[1] *PHP: SOAP - Manual* [online]. 2001 , 02.05. 2008 [cit. 2008-05-03]. Dostupný z WWW: <http://cz.php.net/soap>.

[2] *XML-RPC Specification* [online]. 1999 , 30. 06. 2003 [cit. 2008-04-30]. Dostupný z WWW: <http://www.xmlrpc.com/spec>.

[3] *XML-RPC* [online]. 2008 , 1. 5. 2008 [cit. 2008-04-30]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/XML-RPC>.

[4] *SOAP* [online]. 2008 , 23.04.2008 [cit. 2008-05-01]. Dostupný z WWW: <http://en.wikipedia.org/wiki/SOAP>.

[5] *GSOAP: SOAP C++ Web Services* [online]. 2003 , 06. 2007 [cit. 2008-04-14]. Dostupný z WWW: <http://www.cs.fsu.edu/~engelen/soap.html>.

[6] *Web Services Description Language* [online]. 2003 [cit. 2008-05-02]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Web_Services_Description_Language>.

[7] SCHLOSSNAGLE, George. *Advanced PHP Programming*. [s.l.] : [s.n.], 2004. 650 s. ISBN 0-672-32561-6.

Prílohy

A.1 Dátové typy SOAPu

Názov typu	Popis	Možná hodnota
anyURI	Reťazec podľa syntaxe URI	http://www.vsb.cz
base64Binary	Binárna sekvencia v Base64	TWFuIGlzIGRpc3Rpbmd1a
boolean	Pravdivostná hodnota	True/False
byte	8-bitové číslo	-128 až 127
date	Dátum	"1889-09-24"
dateTime	Čas a dátum	"2004-10-31T21:40:35.5-07:00"
decimal	Desiatkové číslo s pevnou des.čiarkou	"-3.14159"
double	64-bitové číslo s pohyblivou des. Čiarkou	-1.79769E+308
duration	Časový úsek	"P1Y2MT2H5.6S"
float	32-bitové číslo s pohyblivou čiarkou	"6.0235e-23"
gDay	Deň v mesiaci podľa Greg. Kalendára	"---27"
gMonth	Číslo mesiaca podľa Greg. Kalendára	"--06--"
gMonthDay	Gregoriánsky mesiac a deň	"--07-04"
gYear	Gregoriánsky rok	"1889"
gYearMonth	Gregoriánsky mesiac a rok	"1995-08"
hexBinary	Sekvencia bytov, každý uvedený ako 2 hexadecimálne čísla	"0047dedbef"
ID	Unikátny identifikátor podľa XML štandardu	
IDREF	Referencia na ID	
IDREFS	Sekvencia referencií na ID oddelených medzerou	
int	32-bitové celé číslo	-12453
integer		
language	Štandardizovaný kód jazyka	"fj"
long	Integer so zvýšenou presnosťou, 18 cifier	
Name	Meno podľa XML štandardu	
NCName	Lokálna časť presného mena	
negativeInteger	Záporný Integer	
NMTOKEN	Sekvencia znakov mena	
NMTOKENS	Sekvencia NMTOKENS oddelených medzerou	
nonNegativeInteger	Číslo väčšie alebo rovné nule	
nonPositiveInteger	Číslo menšie alebo rovné nule	
normalizedString	Normalizovaný reťazec, teda bez tabulátorov atď.	
positiveInteger	Kladné číslo	
QName	Názov podľa XML štandardu	"xsl:stylesheet"
short	16-bitové číslo	
string	Sekvencia znakov	
time	Čas	"13:04:00"
token	Znakové meno	
unsignedByte	16-bitové celé číslo - kladné	

A.2 Príklad WSDL súboru

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WebSluzba"
 targetNamespace="urn:http://hyzko.ic.cz/soap/"
 xmlns:tns="urn:http://hyzko.ic.cz/soap/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:ns1="urn:http://hyzko.ic.cz/soap/"
 xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

<!-- definicia typov -->
<types>
 <schema targetNamespace="urn:http://hyzko.ic.cz/soap/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns="http://www.w3.org/2001/XMLSchema"
     elementFormDefault="unqualified"
     attributeFormDefault="unqualified">
  <element name="id" type="xsd:string"/>
  <element name="passwd" type="xsd:string"/>
  <element name="bool_value" type="xsd:boolean"/>
  <element name="a" type="xsd:int"/>
  <element name="b" type="xsd:int"/>
  <element name="vysledok" type="xsd:int"/>
 </schema>
</types>

<!-- komunikacne spravy -->
<message name="a_plus_bRequest">
```

```
  <part name="a" element="tns:a"/>
  <part name="b" element="tns:b"/>
</message>

<message name="a_plus_bResponse">
 <part name="vysledok" element="tns:vysledok"/>
</message>

<message name="a_minus_bRequest">
  <part name="a" element="tns:a"/>
  <part name="b" element="tns:b"/>
</message>

<message name="a_minus_bResponse">
  <part name="vysledok" element="tns:vysledok"/>
</message>

<message name="AutorizeUserRequest">
  <part name="id" element="tns:id"/>
  <part name="passwd" element="tns:passwd"/>
</message>

<message name="AutorizeUserResponse">
  <part name="bool_value" element="tns:bool_value"/>
</message>

<!-- dostupne operacie -->
<portType name="cisla">
 <operation name="a_plus_b">
  <input message="tns:a_plus_bRequest"/>
  <output message="tns:a_plus_bResponse"/>
 </operation>
 <operation name="a_minus_b">
```

```
  <input message="tns:a_minus_bRequest"/>
  <output message="tns:a_minus_bResponse"/>
 </operation>
 <operation name="AutorizeUser">
  <input message="tns:AutorizeUserRequest"/>
  <output message="tns:AutorizeUserResponse"/>
 </operation>
</portType>

<!-- volatelne cez HTTP -->
<binding name="WebSluzba" type="tns:cisla">
 <SOAP:binding style="rpc" transport="http://schemas.xmlsoap.org/
soap/http"/>
 <operation name="a_plus_b">
  <SOAP:operation style="rpc"
soapAction="http://hyzko.ic.cz/soap/"/>
  <input>
   <SOAP:body use="literal"
namespace="urn:http://hyzko.ic.cz/soap/"/>
  </input>
  <output>
   <SOAP:body use="literal"
namespace="urn:http://hyzko.ic.cz/soap/"/>
  </output>
 </operation>
 <operation name="a_minus_b">
  <SOAP:operation style="rpc"
soapAction="http://hyzko.ic.cz/soap/"/>
  <input>
   <SOAP:body use="literal"
namespace="urn:http://hyzko.ic.cz/soap/"/>
  </input>
  <output>
   <SOAP:body use="literal"
namespace="urn:http://hyzko.ic.cz/soap/"/>
  </output>
```

```
 </operation>
 <operation name="AutorizeUser">
  <SOAP:operation style="rpc"
soapAction="http://hyzko.ic.cz/soap/"/>
  <input>
   <SOAP:body use="literal"
namespace="urn:http://hyzko.ic.cz/soap/"/>
  </input>
  <output>
   <SOAP:body use="literal"
namespace="urn:http://hyzko.ic.cz/soap/"/>
  </output>
 </operation>
</binding>


<!-- adresy komunikacnych bodov -->
<service name="WebSluzba">
 <documentation>Testovacia web sluzba</documentation>
 <port name="WebSluzba" binding="tns:WebSluzba">
  <SOAP:address location="http://hyzko.ic.cz/soap/server.php5"/>
 </port>
</service>
</definitions>
```

B.1.1 Príklad XML-RPC klienta v C++

```
#include "XmlRpc.h"
#include <iostream.h>

using namespace XmlRpc;

int main(int argc, char* argv[])
{
  if (argc != 3)
  {
    std::cerr << "Pouzitie: client serverHost serverPortn";
    return -1;
  }

  const char* hostname = argv[1];
  int port = atoi(argv[2]);
  XmlRpcClient c(hostname, port);
  XmlRpcValue args, result;
  args[0] = 2;
  args[1] = 2;
  if (c.execute("a_plus_b", args, result))
    std::cout << result << std::endl;

}
```

B.1.2 Príklad XML-RPC serveru v C++

```
#include "XmlRpc.h"
#include <iostream>
#include <stdlib.h>
using namespace XmlRpc;
XmlRpcServer s;
// scita lubovolne mnozstvo cisel typu double a vrati vysledok
class a_plus_b : public XmlRpcServerMethod{
public:
   Sum(XmlRpcServer* s) : XmlRpcServerMethod("a_plus_b", s) {}
     void execute(XmlRpcValue& params, XmlRpcValue& result)
     {
       int nArgs = params.size();
       double sum = 0.0;
       for (int i=0; i<nArgs; ++i)
         sum += double(params[i]);
       result = sum;
     }
} a_plus_b(&s);
int main(int argc, char* argv[]){
  if (argc != 2) {
    std::cerr << "Pouzitie: server Port\n";
    return -1;
  }
  int port = atoi(argv[1]);
  XmlRpc::setVerbosity(5); //nastavuje uroven zaznamenavania, od
0 pre ziadne spravy po 5 ked                          zapisuje
vsetko
  s.bindAndListen(port); //vytvori server na zadanom porte
  s.enableIntrospection(true);
  s.work(-1.0);
  return 0; }
```

B.1.3 Príklad XML-RPC klienta v PHP

```
<?php
      include 'xmlrpc.inc';

      $client = new xmlrpc_client('/xmlrpc/server.php',
                              'localhost', 80);

      $message = new xmlrpcmsg('aplusb',array(new xmlrpcval(5,
'int'),new xmlrpcval(3, q                                  'int')));
      $result = $client->send($message);

      $sum_val = $result->value();
      $sum = $sum_val->scalarval();
      print $sum;

?>
```

B.1.4 Príklad XML-RPC serveru v PHP

```
<?php
include 'xmlrpc.inc';
include 'xmlrpcs.inc';

function aplusb ($params) {

    $a_val = $params->getParam(0);
    $a = $a_val->scalarval();
    $b_val = $params->getParam(1);
    $b = $b_val->scalarval();

    return new xmlrpcresp(new xmlrpcval($a + $b, 'int'));
}


new xmlrpc_server(array('aplusb' => array('function' =>
'aplusb')));
?>
```

B.2.1 Príklad SOAP klienta v C++

```
#include "soapcalcProxy.h"
#include "calc.nsmap"
int main()
{
  calc c; /* calc objekt */
  double n; /* vysledok */
  if (c.add(2, 3, &n) == SOAP_OK)
    cout << "2 plus 3 is " << n << endl;
  else
    soap_print_fault(c.soap, stderr); /* chyba */
  return 0;
}
```

B.2.2 Príklad SOAP serveru ako CGI skriptu v C++

```
#include "soapH.h"
#include "calc.nsmap" /* subor s vygenerovanymi mennymi
priestormi */
#include <math.h>


int main()
{
      return soap_serve(soap_new());
}


int ns__aplusb(struct soap *soap, double a, double b, double
*result)
{
      *result = a + b;
      return SOAP_OK;
}
```

B.2.3 Príklad SOAP serveru ako samostatnej aplikácie v C++

```
#include "soapH.h"
#include "calc.nsmap" /* generovany subor s mennymi priestormi
sluzby */
#include <math.h>
int main(int argc, char** argv)
{
  int m, s; /* sockety */
  struct soap *soap = soap_new();
  if (argc < 2)
    soap_serve(soap); /* ak neboli zadane argumenty, pokracuj ako
CGI aplikacia */
  else {
     m = soap_bind(soap, NULL, atoi(argv[1]), 100);
     if (m < 0){
          soap_print_fault(soap, stderr);
               exit(-1);
     }
     for (;;){
          s = soap_accept(soap);
               fprintf(stderr, "Spojenie prijate");
               if (s < 0){
               soap_print_fault(soap, stderr);
               exit(1);
               }
               soap_serve(soap);
          soap_end(soap);
     }
  }
  soap_done(soap);
  free(soap);
  return 0;
```

```
}


int ns__aplusb(struct soap *soap, double a, double b, double
*result)
{
      *result = a + b;
      return SOAP_OK;
}
```

B.2.4 Príklad SOAP klienta v PHP s WSDL súborom

```
<?php
     $wsdl_location = "";    // URL wsdl suboru popisujuceho
webovu sluzbu
     $options = array();  //pole nastaveni, pri wsdl mode
klienta nepovinne, pri non-wsdl                              mode
je url a location servera povinne


     $client =new SoapClient( $wsdl_location , $options );
     $a=5;
     $b=4;
     echo "<br/>";
     echo $client->a_plus_b($a,$b);


?>
```

B.2.5 Príklad SOAP klienta v PHP bez WSDL súboru

```
<?php
     $options = array('location' =>
"http://hyzko.ic.cz/soap/server_nowsdl.php5",'uri' =>
"http://hyzko.ic.cz/soap");


     $client = new SoapClient(null, $options);
     echo $client->a_plus_b(3, 2);


?>
```

B.2.6 Príklad SOAP serveru v PHP s WSDL súborom

```
<?php

    function a_plus_b($a, $b) {
      return $b + $a;
    }
  $wsdl_location = "" ; //umiestnenie wsdl suboru sluzby
  $options = array(); //pole nastaveni, nepovinne
  $server = new SoapServer($wsdl_location, $options);
  $server->addFunction(SOAP_FUNCTIONS_ALL);
  $server->handle();
?>
```

B.2.7 Príklad SOAP serveru v PHP bez WSDL súboru

```
<?php
  class MojServer{
      function a_plus_b($a, $b) {
          return $a + $b;
      }

  }
      $server = new SoapServer(null, array('uri' =>
"http://hyzko.ic.cz/soap"));
      $server->setClass("MojServer");
      $server->handle();
?>
```

B.3 Implementované funkcie pre potreby Virtlabu

```
class WSServer{

      private $usingBase64;
      private $sql;

      function __construct($sql_server, $sql_user, $sql_pass,
$sql_db, $useBase64){
            $this->sql_server   = $sql_server;
         $this->sql_user     = $sql_user;
         $this->sql_pass = $sql_pass;
         $this->sql_db = $sql_db;
         $this->usingBase64 = $useBase64;
         $tmp = new virtlabSQL($sql_server, $sql_user, $sql_pass,
$sql_db);
         $this->sql = &$tmp;

      }

      function usingBase64(){
            return $usingBase64;
      }

      function select( $pKeyValue,  $table, $column) {

         $primaryKey = $this->sql->get_primary_key($table);

            $query = "SELECT ".$column." FROM ".$table." WHERE ".
$primaryKey."='".$pKeyValue."';";

            $result = $this->sql->query($query, 0);

            $value = $this->sql->fetch_assoc($result);
```

```
            if($usingBase64 == 1)
                  return base64_encode($value[$column]);
            else
                  return $value[$column];
      }

      function selectTable($columns, $table, $expr) {

            $query = "SELECT ".$columns." FROM ".$table." WHERE ".
$expr;

            $result = $this->sql->query($query, 0);

            $i = 0;

            while($row = $this->sql->fetch_assoc($result)){

                  $data[$i] = $row;

                  $i=$i+1;

            }

            if($usingBase64 == 1)
                  return base64_encode(serialize($data));
            else
                  return serialize($data);
      }

      function userList(){

            $query = "SELECT id FROM users WHERE 1";
```

```
            $result = $this->sql->query($query, 0);

            $i = 0;

            while($row = $this->sql->fetch_assoc($result)){

                  $data[$i] = $row;

                  $i=$i+1;

            }

            if($usingBase64 == 1)
                  return base64_encode(serialize($data));
            else
                  return serialize($data);

      }

      function userInfo($id){

            $query = "SELECT first_name, surname, email, lang,
timezone, quota FROM users WHERE id='".$id."'";

            $result = $this->sql->query($query, 0);

            $data = $this->sql->fetch_assoc($result);

            if($usingBase64 == 1)
                  return base64_encode(serialize($data));
            else
                  return serialize($data);

      }
```

```
	function userAutentize($id, $passwd){

		$query = "SELECT passwd FROM users WHERE id='".
$id."'";

		$result = $this->sql->query($query, 0);

		if($row = $this->sql->fetch_assoc($result)){
			if($row[passwd]=="--LDAP--"){
				//if(checkLDAP($id, $passwd)) value="0";
				//else value="1";
			}
			else{
				if($row[passwd]==$passwd){
					$value="0";  //spravne heslo
				}
				else
				{
					$value="2";  //zle heslo
				}
			}
		}
		else
		{
			$value="1";  //neexistujuci uzivatel
		}

		return $value;

	}

	function getTaskCategories(){
```

```
            $query = "SELECT id, name FROM task_classes WHERE 1";

            $result = $this->sql->query($query, 0);

            $data = array();

            while($row = $this->sql->fetch_assoc($result)){

                  $data[$row[id]][0] = $row[name];

            }

            $query = "SELECT id, class_id , name FROM
task_categories WHERE 1";

            $result = $this->sql->query($query, 0);

            while($row = $this->sql->fetch_assoc($result)){

                  $tmp = $row[class_id];
                  unset($row[class_id]);
                  $data[$tmp][$row[id]] =$row;

            }

            if($usingBase64 == 1)
                  return base64_encode(serialize($data));
            else
                  return serialize($data);
      }

      function getTaskList($exp){

            $query = "SELECT id, name_short FROM tasks where id IN
```

```
(SELECT task_id FROM tasks_categories WHERE ".$exp.")";
            //return serialize($query);
            $result = $this->sql->query($query, 0);

            $i = 0;

            while($row = $this->sql->fetch_assoc($result)){

                  $data[$i] = $row;

                  $i=$i+1;

            }

            if($usingBase64 == 1)
                  return base64_encode(serialize($data));
            else
                  return serialize($data);
      }
}
```