Elektronické bankovníctvo prostredníctvom webservisov

DIPLOMOVÁ PRÁCA

Michal Rajniak 2006

Fakulta matematiky, fyziky a informatiky
Univerzity Komenského v Bratislave
Katedra informatiky

Elektronické bankovníctvo prostredníctvom webservisov

DIPLOMOVÁ PRÁCA

Diplomant: Michal Rajniak
Diplomový vedúci: Ing. Ivan Tarapčík
Názov študijného odboru: Informatika
Bratislava 2006

Ďakujem môjmu diplomovému vedúcemu, Ing. Ivanovi
Tarapčíkovi, za odborné vedenie práce, cenné rady a pripomienky.

Abstrakt

Cieľom tejto práce bolo vytvoriť návrh webservisu pre prístup k elektronickému bankovníctvu – všeobecne pre slovenský platobný styk a špecificky pre jednu konkrétnu banku, jej požiadavky a podnikové procesy. Tento návrh mal vychádzať z niektorého otvoreného štandardu pre výmenu finančných informácií.
V prvej časti tejto práce popisujeme technológie webových služieb, ktoré sú relevantné pre náš návrh. V druhej časti je popísaný model elektronického bankovníctva. V tretej časti práce rozoberáme, prečo sme si ako základ nášho návrhu vybrali štandard IFX (Interactive Financial Exchange) a ako sme ho rozšírili a upravili pre naše potreby.
Štvrtá časť tejto práce je samotný návrh webservisu. Je definovaná štruktúra vymieňaných správ a ich obsah, aké operácie sú pomocou výmeny týchto správ realizované, je tu popísaný komunikačný protokol pre výmenu správ a pravidlá pre spracovanie týchto správ serverom. Je navrhnutý bezpečnostný model zabezpečujúci zodpovednosť komunikujúcich strán za akcie vykonane na základe vykomunikovaných správ. Prílohou k tejto práci je podrobný popis obsahu navrhnutých správ, XML schéma pre tieto správy a WSDL dokument definujúci webservis.

Obsah

Úvod

Ciele diplomovej práce

Primárnym cieľom bolo vytvoriť návrh webservisu pre prístup k elektronickému bankovníctvu.
Naším cieľom bolo vytvoriť návrh pre jedny konkrétne procesy elektronického bankovníctva v jednej banke a súčasne sa pokúsiť vytvoriť všeobecnú časť – spoločné jadro pre slovenský platobný styk – ktoré by (po úpravách) bolo (v prípade záujmu) použiteľné aj v ostatných slovenských bankách.

Vopred danou požiadavkou bolo, aby bol tento návrh založený na nejakom existujúcom štandarde pre výmenu finančných informácií. Tým sme chceli získať niektoré z možných výhod použitia štandardného otvoreného riešenia – interoperabilitu, lepšiu akceptáciu zo strany zákazníkov a možno aj zo strany ostatných slovenských bank a možno aj ľahšiu implementovatelnosť (potenciálne môžu existovať hotové knižnice implementujúce daný štandard).

Hlavnou výhodou webservisu oproti iným kanálom elektronického bankovníctva (Internetbanking atd.) je existencia vzdialeného programovateľného rozhrania. Voči nemu si užívatelia môžu naprogramovať ľubovoľného klienta, ktorý dokáže konformným spôsobom k poskytovaným službám pristupovať. Toto umožňuje automatizáciu vykonávania poskytovaných operácií a takisto vzniká možnosť integrovať prístup k elektronickému bankovníctvu do podnikových procesov firiem pomocou modulov elektronického bankovníctva, ktoré je možné použiť vnútri ekonomického alebo iného softvéru firmy.
Z tohto dôvodu sme chceli do návrhu zahrnúť najmä tie bankové služby, pre ktoré existuje požiadavka automatizácie a integrácie.

Dôležitou požiadavkou bolo, aby návrh umožňoval *bezpečnú* komunikáciu medzi bankou a zákazníkmi a poskytol (kryptografické) prostriedky pre riešenie prípadných sporov, ktoré by mohli vzniknúť na základe operácií vykonaných na základe informácií vykomunikovaných cez tento kanál.

Prehľad obsahu

Táto diplomová práca sa skladá z nasledovných častí (kapitol).
Prvá kapitola (*Webservisy*) poskytuje čitateľovi prehľad technológií webových služieb relevantných pre náš návrh.
V druhej kapitole (*Elektronického bankovníctvo*) je popísané elektronické bankovníctvo, resp. jeho zjednodušený model – aktéri a pasívne prvky, interakcie medzi nimi a spôsob komunikácie medzi aktérmi. V tejto kapitole je takisto uvedený prehľad rôznych foriem elektronického bankovníctva a sú tu popísané výhody elektronického bankovníctva realizovaného prostredníctvom webových služieb.
V tretej kapitole (*Postup riešenia a zdôvodnenie rozhodnutí*) popisujeme, ako a prečo sme vybrali štandard IFX a ako sme ho upravili pre naše potreby.
Štvrtá kapitola (*Návrh*) obsahuje popis navrhnutého riešenia. Popisuje cieľovú architektúru systému, komunikačný protokol a správy, pomocou ktorých sú operácie a služby elektronického bankovníctva realizované. Takisto sú tu spísané bezpečnostné požiadavky a návrh ich riešení. Niektoré časti návrhu sú popísané až v prílohách k tejto práci – tieto prílohy sú popísané v kapitole *Prílohy*.
Zhrnutie výsledkov a odporúčania pre budúce rozšírenia a implementáciu sa nachádzajú v poslednej kapitole tejto práce (*Záver*).

1 Webservisy

1.1 Úvod

Webové služby[1] sú distribuovaná technológia, kde komunikujúce strany (spravidla klient a server) komunikujú prostredníctvom výmeny XML správ cez sieť. Server (poskytovateľ webovej služby) má verejný interfejs popísaný jazykom WSDL, kde sú popísané operácie, ktoré môže klient vzdialene vykonávať a ktoré konštituujú ucelenú službu (napr. elektronické bankovníctvo). Správy sú prenášané v XML formáte spravidla v rámci tzv. SOAP obálky. Na prenos správ sa najčastejšie používa protokol HTTP, ale je možné použiť širokú skalu iných protokolov. Informácie o službách môžu poskytovateľ zverejniť v tzv. UDDI adresári, kde o nich klienti môžu získať informácie a následne webservis využívať. Schematicky si tieto idey môžeme načrtnúť nasledovným spôsobom.

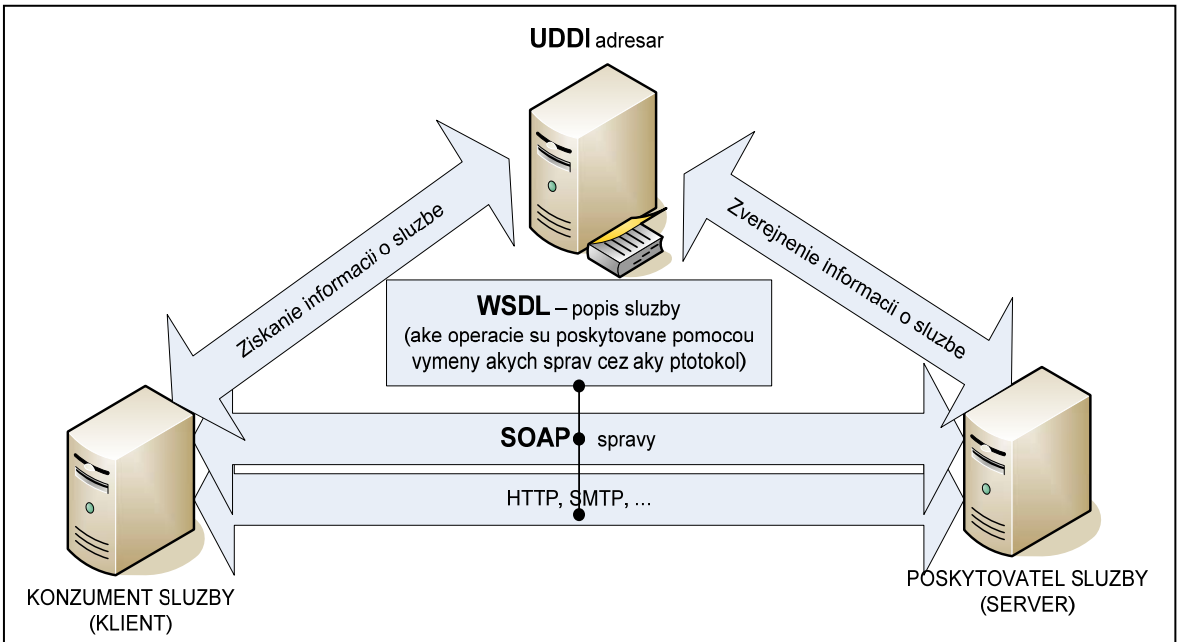


Diagram 1: Webové služby a použité protokoly

Vzhľadom na to, že operácie sú popísané spôsobom nezávislým od programovacieho jazyka a správy sa vymieňajú vo formáte XML transparentne popísaným spôsobom, web servisy umožňujú veľkú flexibilitu implementácie tak na strane serveru ako na strane klienta. Jediná podmienka, ktorú musia implementácie spĺňať, je schopnosť prijímať a odosielať "správne" správy "správnym" spôsobom (podľa definície operácií vo WSDL dokumente).

Technológie webových služieb sú popísané pomocou neustále rastúcej množiny štandardov, základné jadro je však relatívne stabilné a tvoria ho protokoly WSDL a SOAP a jadro XML technológií[2], na ktorom sú SOAP a WSDL postavené. UDDI sa v praxi zatiaľ neuchytilo[3] a ani my ho v našom návrhu nebudeme používať.

---

[1] Na označenie anglického termínu „Web services" budeme používať v tejto práci výrazy „Webové služby" alebo „Webservisy". Podobne, v jednotnom čísle „web service" budeme prekladať ako „webová služba" alebo „webservis".

[2] XML, XML namespaces, XML schema. Viď [XML], [XMLns], [XMLschema0].

[3] Toto môže byt kvôli tomu, že use case, ktorý sa mal pomocou neho realizovať, dnes nie je až tak požadovaný, resp. dá sa realizovať iným, jednoduchším spôsobom. Použitie UDDI má zmysel napríklad v prípade, že adresár obsahuje niekoľko (veľa) záznamov o podobných webových

Jedným zo základných cieľov webových služieb je interoperabilita medzi jednotlivými implementáciami. Jednotlivé štandardy a ich kombinácia však často umožňujú flexibilitu, ktorá síce na jednej strane poskytuje "silnejšie" prostriedky, ale z hľadiska interoperability môže spôsobiť problémy (komunikujúce strany musia podporovať všetky možnosti a odchýlky, ktoré si druhá strana môže dovoliť). Kvôli týmto problémom vznikla WS-Interoperability Organization, ktorá zverejňuje odporúčania[4] pre implementátorov na dosiahnutie interoperability.

V zvyšku tejto kapitoly si popíšeme základné štandardy webových služieb (WSDL, SOAP) a niektoré ďalšie štandardy, ktoré budeme v našom návrhu využívať (XML Signature, WS Security). V ich popise pôjdeme len do takej hĺbky, aby sme popísali prvky, ktoré budú relevantné pre náš návrh, ale na druhej strane chceme čitateľovi poskytnúť aspoň čiastočný prehľad o technológiách webových služieb a pridružených štandardoch.
V nasledujúcom texte predpokladáme, že čitateľ dobre pozná jazyk XML a ako sú v ňom definované menné priestory a takisto že pozná základy XML Schémy.

1.2 SOAP

1.2.1 Úvod

V nasledujúcom si povieme niečo o protokole SOAP vo verzii 1.2. Budeme vychádzať najmä zo [SOAP12pt1].

SOAP je odľahčený protokol určený pre výmenu informácií vo forme XML správ v decentralizovanom, distribuovanom prostredí. Cieľom SOAPu je jednoduchosť a rozšíriteľnosť. SOAP sa preto nesnaží zahrnúť prvky (služby), ktoré sa často nachádzajú v distribuovaných systémoch, akými sú napríklad spoľahlivosť, bezpečnosť, korelácia, routovanie apod. [5]

SOAP špecifikácia sa skladá z dvoch častí.
Prvá časť[6] definuje "rámec pre výmenu správ" (SOAP messaging framework), ktorý popisuje:

- štruktúru vymieňaných správ, t.j. formát SOAP správy (SOAP message)
- proces spracovania SOAP správy (SOAP processing model)
- spôsob ako sa dá SOAP messaging framework rozšíriť (SOAP extensibility model)
- pravidlá ako sa dá SOAP naviazať na protokol nižšej (transportnej) vrstvy (SOAP Protocol Binding Framework)

Konformné SOAP implementácie musia implementovať a dodržiavať všetko, čo je spomenuté v prvej časti SOAP špecifikácie.

V druhej časti špecifikácie[7], ktorou sa budeme zaoberať len minimálne, je definovaných niekoľko dodatkov do jadra definovaného v prvej časti špecifikácie. Táto časť špecifikácie definuje (spomíname iba niektoré časti):

---

službách a klient si môže vybrať tu jemu najviac vyhovujúcu. Verejných webových služieb je však malo – dnes sa webové služby využívajú najmä na integráciu existujúcich aplikácii v rámci organizácii. V prípade verejnej webovej služby nie je problém informovať potenciálnych záujemcov iným spôsobom, napr. jednoduchým zverejnením WSDL súboru a pripadne ďalších informácii na webovej stránke poskytovateľa.

[4] Takýmito sú napríklad WS-I Basic Profile (vid [WSIBP]) a WS-I Basic Security Profile (viď [WSIBSP]).

[5] Viaceré z týchto sú už definovane ako štandardné rozšírenia SOAPu

[6] SOAP Version 1.2 Part 1: Messaging Framework, viď [SOAP12pt1].

[7] SOAP Version 1.2 Part 2: Adjuncts, Viď [SOAP12pt2]

- jeden spôsob ako zakódovať aplikačné dáta do XML a umožniť takto ich zahrnutie v SOAP správach (SOAP data model & SOAP encoding)
- ako použiť SOAP data model na reprezentáciu RPC volaní (SOAP RPC Representation)
- naviazanie SOAPu na HTTP protokol (HTTP Binding). Toto je vytvorené na základe SOAP protocol binding framework.

Prvými dvoma spomenutými sa zaoberať nebudeme, keďže SOAP kódovanie nie je odporúčané[8] kvôli problémom s interoperabilitou a okrem toho my ho v našom návrhu používať nebudeme. Takisto sa nebudeme zaoberať RPC štýlom volaní, pretože náš návrh je založený na výmene správ v štýle document. K rozdielu medzi štýlmi RPC a document sa ešte dostaneme.
Konformné SOAP implementácie môžu (ale nemusia) implementovať ľubovoľný z dodatkov z druhej časti špecifikácie.

1.2.2 Základné pojmy

Na začiatok si uvedieme niekoľko pojmov. Tieto budeme potrebovať v ďalšom texte.
SOAP (SOAP protokol) – množina pravidiel a konvencií týkajúca sa formátu, spracovania a vymieňania SOAP správ medzi SOAP uzlami.
SOAP node (uzol) – stelesnenie procesnej logiky potrebnej na odosielanie, prijímanie a spracovávanie SOAP správ podľa protokolu. Pre každý SOAP uzol existuje URI, ktoré ho identifikuje.
SOAP role (rola) – očakávaná funkcia SOAP príjemcu pri spracovaní správy. Príjemca môže vystupovať vo viacerých roliach.
SOAP binding (naviazanie) – Množina pravidiel na prenos SOAP správy pomocou nižšieho protokolu akým je napríklad HTTP.
SOAP feature (rozširujúci prvok) – Rozšírenie SOAP messaging framework o nejaký prvok funkcionality. Napríklad spoľahlivosť, bezpečnosť, korelácia, routovanie, MEPs.
SOAP module (modul) – Špecifikácia syntaxe a sémantiky jedného alebo viacerých SOAP hlavičkových blokov. Pomocou modulov sú realizované nove prvky funkcionality (SOAP features).
SOAP MEP – Message Exchange Pattern (vzor výmeny správ) – šablóna pre výmenu SOAP správ medzi SOAP uzlami umožnená pomocou SOAP naviazania (binding) na nižší protokol. MEP je príklad SOAP feature. Príkladom MEP je request-response realizovaný ako HTTP request-response.
SOAP aplikácia – program, ktorý vytvára, odosiela, prijíma, alebo iným spôsobom narába so SOAP správami podľa SOAP procesného modelu.
SOAP message (správa) – základná jednotka komunikácie medzi SOAP uzlami.
SOAP envelope (obálka) – XML element tvoriaci obsah SOAP správy.
SOAP header (hlavička) – súbor nula a viac SOAP hlavičkových blokov, z ktorých každý môže byť smerovaný na ľubovoľného prijímateľa pozdĺž SOAP cesty správy.
SOAP header block (hlavičkový blok) – XML element, ktorý logicky vymedzuje samostatnú funkčnú jednotku v SOAP hlavičke.
SOAP body (telo) – súbor nula alebo viac XML elementov určených konečnému príjemcovi SOAP správy.
SOAP fault (chyba) – XML element obsahujúci informácie o chybe vygenerovanej SOAP uzlom.
SOAP sender (odosielateľ) – SOAP uzol, ktorý odosiela SOAP správy.
SOAP receiver (príjemca) – SOAP uzol, ktorý prijíma správy.
SOAP message path (cesta správy) – postupnosť uzlov, cez ktoré prechádza SOAP správa na ceste od počiatočného odosielateľa ku koncovému príjemcovi.
Initial SOAP sender (počiatočný odosielateľ) – uzol, ktorý správu vytvorí a pošle ju ako prvý.

[8] WS-I Basic Profile jeho použitie zakazuje.

SOAP intermediary (prostredník) – uzol, ktorý je súčasne príjemcom aj odosielateľom. Najprv správu prijme, po čom môže spracovať niektoré hlavičkové bloky a nakoniec správu odošle.
Ultimate SOAP receiver (konečný príjemca) – Konečná destinácia SOAP správy. Tento uzol je zodpovedný za spracovanie obsahu tela SOAP správy.

1.2.3 Model komunikácie

SOAP modeluje komunikáciu ako výmenu správ medzi uzlami. Uzly odosielajú, prijímajú a spracúvajú SOAP správy. SOAP správa je vytvorená a odoslaná počiatočným odosielateľom, správa je následne prijatá a preposlaná (možno s modifikáciami) nula alebo viac prostredníkmi a nakoniec je prijatá konečným príjemcom. Táto postupnosť uzlov definuje cestu správy (SOAP message path). Tieto základné idey komunikačného modelu SOAPu sú načrtnuté na diagrame (2).

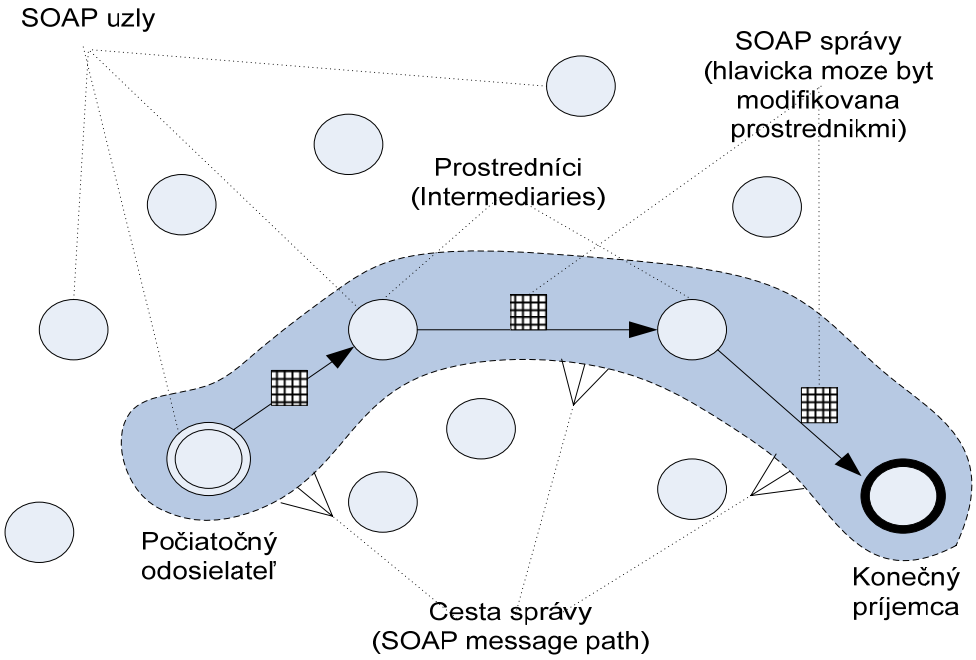


Diagram 2: Demonštrácia ideí SOAP protokolu

Predtým, než podrobnejšie rozoberieme jednotlivé prvky SOAP protokolu, skúsme si uviesť jednoduchý príklad.

Predpokladajme, že máme banku a zákazníka. Banka chce zákazníkovi poskytovať služby prostredníctvom počítačovej siete. Predpokladajme, že banka a zákazník sú v počítačovej sieti zastúpené SOAP uzlami – každý z nich jedným. Uzol zastupujúci banku nazveme server a uzol zastupujúci zákazníka nazveme klient. Banka poskytuje služby zákazníkovi tým spôsobom, že mu umožňuje prostredníctvom klienta posielať požiadavky serveru. Ten tieto požiadavky vybavuje, čím služby realizuje. Pozrime sa ako by mohla vyzerať dvojica správ vymenená medzi týmito uzlami v prípade, že zákazník chce vykonať prevod peňazí z účtu na účet.

Správa (požiadavka) odoslaná klientom môže vyzerať nasledovne:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">

   <env:Header>
      <t:TransferType xmlns:t="http://example.org/transfer"
                      env:role="http://example.org/PriorityServer"
                      env:mustUnderstand="true">
         <t:priority>High</t:priority>
      </t:TransferType>
   </env:Header>

   <env:Body>
      <t:TransferRequest xmlns:t="http://example.org/transfer">
         <t:FromAccount>3492837564/1100</t:FromAccount>
         <t:ToAccount>7480293410/900</t:ToAccount>
         <t:Amount>1000</t:Amount>
      </t:TransferRequest>
   </env:Body>

</env:Envelope>
```

Diagram 3: Príklad SOAP obálky – požiadavka klienta

Predpokladajme, že server túto správu prijal, spracoval, vykonal prevod peňazí atd. Potom môže poslať klientovi späť napríklad nasledovnú správu (odpoveď).

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Body>
      <t:TransferResponse xmlns:t="http://example.org/transfer">
         <t:Result>OK</t:Result>
      </t:TransferResponse>
   </env:Body>
</env:Envelope>
```

Diagram 4: Príklad SOAP obálky – odpoveď serveru

Tento príklad využijeme v ďalšom texte na vysvetlenie základných konceptov SOAP protokolu. Najprv si povedzme niečo o štruktúre SOAP správy.

1.2.4 Štruktúra SOAP správy

SOAP správa je definovaná ako XML dokument[9], ktorého koreňový element je SOAP Envelope (SOAP obálka). SOAP obálka môže mat jedného alebo dvoch potomkov. Prvým (a nepovinným) je hlavička (SOAP Header). Druhým (a povinným) je telo (SOAP Body). Obsah týchto elementov je závislý od konkrétneho použitia – SOAP nekladie podmienky na to, aké XML môžu/majú tieto elementy obsahovať.

V našom príklade je (v požiadavke klienta) v tele SOAP správy uvedená žiadosť o prevod peňazí (element `TransferRequest`), pričom SOAP hlavička špecifikuje typ tohto prevodu – prevod s vysokou prioritou (element `TransferType`).

[9] Presnejšie, je definovaná ako XML Infoset (vid [XMLInfoset]), ale pre naše potreby celkom postačí, ak budeme predpokladať XML dokument. Stručne a zjednodušene, XML infoset je "abstraktný model dát, ktoré sú obsiahnuté v XML dokumente". XML dokument si potom môžeme predstaviť jeden spôsob serializácie dát definovaných v konkrétnom XML infosete.

Rozdelenie SOAP obálky na hlavičku a telo nie je samoúčelné. Telo SOAP obálky obsahuje aplikačné dáta (nula alebo viac XML elementov) a je určené pre konečného príjemcu. Hlavička SOAP obálky obsahuje „kontrolné“ informácie, informácie týkajúce sa spracovania správy, bezpečnosti apod. Obsahuje nula alebo viac XML elementov – hlavičkových blokov. Každý z týchto blokov môže byť určený pre ľubovoľný uzol pozdĺž cesty správy, teda nie len pre konečného príjemcu, tak ako je to pri tele SOAP správy.

1.2.5 Viac o hlavičke a hlavičkových blokoch.

Vraveli sme, že hlavičkové bloky obsahujú kontrolné informácie. V našom príklade obsahuje hlavička len jeden blok definujúci prioritu bankového prevodu. Táto informácia by v banke mohla byť využitá napríklad takým spôsobom, že prvý SOAP uzol v banke, ktorý prijme túto správu, ju v závislosti od priority pošle na prioritné vybavenie na prioritný server (SOAP uzol) alebo na štandardné vybavenie (iný SOAP uzol).

Jednotlivé hlavičkové bloky môžu byť namierené/smerované (targeted) na konkrétne uzly pozdĺž cesty správy. Na toto je použitý atribút „role“ uvedený v top-level elemente hlavičkového bloku. Obsahuje URI, ktoré identifikuje rolu. Zároveň vieme, že každý SOAP uzol vystupuje v nejakej roli. Ak sú tieto role rovnaké, je hlavičkový blok smerovaný na tento uzol[10] a môže byť týmto uzlom spracovaný.

V našom príklade je hlavička namierená na uzly, ktoré vystupujú v roli „servera priorít” (čomu zodpovedá URI `http://example.org/PriorityServer`).

Ďalší atribút pre hlavičkové bloky, ktorý si môžeme všimnúť v hlavičkovom bloku z nášho príkladu, je atribút `mustUnderstand`. Je to štandardný atribút pre hlavičkové bloky, ktorý určuje, či uzol musí „rozumieť“ hlavičkovému bloku, v prípade že ho ide spracovať. Toto znamená, že musí jednak poznať jeho meno[11], musí rozumieť jeho sémantike a na základe nej vedieť správne konať. V našom prípade by teda uzol musel poznať element s kvalifikovaným menom `{http://example.org/transfer}TransferType`, rozumieť, že slúži na preposielanie SOAP správy na príslušný server a byť schopný toto preposlanie vykonať.

V prípade, že uzol je namierený na hlavičkový blok s atribútom mustUnderstand=“true“, je to pre tento uzol „povinný“ hlavičkový blok. To znamená, že mu musí rozumieť a musí ho správne spracovať. Ak to nedokáže, musí zlyhať a vygenerovať chybu. Atribútom mustUnderstand s hodnotou true sú spravidla označené hlavičkové bloky, ktoré sú nutné pre správne spracovanie danej SOAP správy ako celku. K týmto ideám sa ešte vrátime pri popise procesného modelu SOAPu.

Nakoniec si ešte všimnime, že v našom príklade sme nastavili atribút mustUnderstand na true, pretože pokladáme za nutné, aby bola žiadosť o prevod správne preposlaná ako prioritná/neprioritná.

1.2.6 Možnosti rozšírenia SOAPu

Ďalšie špecifikum SOAP hlavičky je, že je to miesto, kde je možné definovať a používať rozšírenia SOAPu (viď definície SOAP Features a SOAP modules). Existuje napríklad štandardné rozšírenie SOAP protokolu s názvom WS Security, ktoré poskytuje nástroje pre

[10] Uzlov s danou rolou môže byt viac, takže presnejšie, hlavičkový blok je smerovaný *aj* na tento uzol.

[11] Kvalifikovane meno (qulified name). Meno elementu „kvalifikovane“ menom menného priestoru (namespace). V našom príklade je meno elementu TransferType; názov namespace-u je http://example.org/transfer; kvalifikovane meno je {http://example.org/transfer}TransferType. Kučeravé zátvorky nemajú žiadny špeciálny význam, len oddeľujú namespace a meno elementu.

„bezpečnú“ výmenu SOAP správ. Toto rozšírenie (SOAP Feature) je realizované pomocou SOAP modulu, ktorým je element (hlavičkový blok) „Security“, ktorý môže obsahovať digitálne podpisy, autentifikačné údaje apod.
Ďalšími existujúcimi rozšíreniami sú napríklad WS Routing alebo WS Reliable Messaging. K WS Security sa ešte neskôr vrátime, keďže ho budeme používať aj v našom návrhu.

Existujú 2 mechanizmy na pridanie nových prvkov funkcionality do SOAPu. Prvým z nich je už spomenuté vytvorenie SOAP modulu a definovanie novej funkcionality pomocou neho. Druhý spôsob rozšírenia nám umožňuje SOAP Protocol Binding Framework – môžeme definovať ako sú SOAP správy odosielané a prijímané pomocou nižšieho protokolu.

V našom príklade je komunikácia medzi klientom a serverom založená na request/response (požiadavka/odpoveď) modeli. Štandardne sa v ňom, rovnako ako v našom príklade, nachádzajú 2 entity – klient a server, pričom model komunikácie je taký, že klient iba posiela požiadavky a server iba posiela odpovede na tieto požiadavky. Týmto spôsobom je špecifikované to, čo sa v SOAPe nazýva Message Exchange Pattern – MEP (vzor pre výmenu správ). MEP je špecifikovaný pomocou naviazania na nižší protokol, napr. HTTP. HTTP request/response prirodzene umožňuje SOAP request/response MEP. Typov MEP existuje niekoľko, nám však celkom postačí, keď sa obmedzíme na request/response, lebo toto bude jediný, ktorý bude v našom návrhu zahrnutý.

1.2.7 SOAP processing model (proces spracovania SOAP správy)

SOAP procesný model definuje proces spracovania správy po jej prijatí SOAP príjemcom. Vieme, že SOAP správa je vytvorená a odoslaná počiatočným odosielateľom konečnému príjemcovi cez nula alebo viac prostredníkov. Každý z prijímateľov musí správu nejakým spôsobom spracovať.

Uzol počas spracovania správy koná v (nejakej) SOAP roli. Každá rola je identifikovaná pomocou URI. Existujú tri role definované priamo SOAP špecifikáciou: next, none a ultimateReceiver. V roli next musí konať každý prostredník a konečný príjemca. V roli ultimateReceiver koná iba konečný príjemca a v roli none nesmie konať žiadny SOAP uzol. Okrem toho si môžu SOAP aplikácie definovať ľubovoľné vlastné SOAP role (my sme v našom príklade definovali rolu prioritného serveru).

Dôvod existencie rolí SOAP uzlov je zameriavanie (targeting) uzlov pozdĺž cesty správy. Ako vieme, každý SOAP hlavičkový blok môže špecifikovať rolu a prostredníctvom nej uzly, na ktoré je namierený (prípad: rola uzla a rola uvedená v hlavičkovom bloku sú rovnaké).
Takisto vieme, že niektoré hlavičkové bloky môžu byť pre daný uzol „povinné“. Je tomu tak v prípade, že blok je na daný uzol namierený a zároveň má atribút mustUnderstand nastavený na true. V tomto prípade musí uzol hlavičkový blok správne spracovať alebo zlyhať a vygenerovať chybu.

Telo SOAP obálky, na rozdiel od hlavičkových blokov, ktoré môžu byť namierené na ľubovoľné uzly pozdĺž cesty SOAP správy, je namierené vždy na konečného príjemcu.

Teraz môžeme pristúpiť k definovaniu postupnosti krokov, ktoré musí uzol vykonať po prijatí správy. Je nasledovná:
Uzol najprv skontroluje či rozumie všetkým povinným hlavičkovým blokom, ktoré sú naňho namierené. V prípade, že nerozumie, musí vygenerovať SOAP fault a ukončiť spracovanie.

V opačnom prípade ich spracuje a môže (ale nemusí) spracovať aj nepovinné hlavičkové bloky, ktoré sú naňho namierené a ktorým rozumie. V prípade, že tento uzol je konečným príjemcom, spracuje aj SOAP Body. Ak je tento uzol prostredníkom, prepošle SOAP správu – s tým, že môže niektoré z hlavičkových blokov odobrať, pridať alebo modifikovať. Presne akým spôsobom toto prostredník môže/musí robiť tu nebudeme rozoberať[12].

V prípade, že pri spracovaní správy, t.j. pri vykonávaní ľubovoľného z horeuvedenej postupnosti krokov dôjde k chybe, uzol vygeneruje SOAP fault.

1.2.8 Signalizovanie chýb (SOAP Faults)

SOAP poskytuje model ako ošetriť situácie keď nastanú chyby počas spracovania správy, pričom sa rozlišuje medzi samotným faktom vygenerovania chyby a schopnosťou signalizovať túto chybu pôvodcovi chybnej správy alebo inému uzlu. Schopnosť a spôsob signalizovania chyby závisí od transportného mechanizmu – týka sa teda naviazania na nižší protokol. V špecifikácii každého naviazania je popísané, ako sú chyby signalizované. Ďalej predpokladáme, že signalizovanie chýb je možné a pozrime sa ako vyzerá štruktúru SOAP chyby (fault).

V prípade, že pri spracovávaní správy uzlom dôjde k chybe, táto je zaznamenaná v elemente Fault. V prípade, že chceme chybu signalizovať, Fault musí byť jediným elementom obsiahnutým v SOAP Body elemente. Príklad Fault elementu je uvedený v diagrame (5). Fault element má vždy minimálne dvoch priamych potomkov. Sú nimi povinné elementy Code a Reason. Nepovinné elementy sú Node, Role a Detail.

Code obsahuje elementy Value (povinný) a Subcode (nepovinný). Hodnoty, ktoré môže nadobúdať Value sú definované SOAPom a je ich 5. Podstatné sú hodnoty Sender a Receiver. Sender indikuje chybu odosielateľa správy (napr. že v tele správy chýba niektorý povinný element). Hodnota Receiver indikuje chybu spracovania správy SOAP príjemcom, t.j. uzlom, ktorý správu spracováva. Ďalšie možné hodnoty sú VersionMismatch, DataEncodingUnknown a MustUnderstand. MustUnderstand indikuje, že niektorému povinnému hlavičkovému bloku príjemca nerozumel (a preto vygeneroval chybu). Zvyšnými dvoma sa nebudeme zaoberať.

Štruktúra Subcode elementu je podobná štruktúre Code elementu. Obsahuje povinný Value element a nepovinný Subcode element, teda má rekurzívnu štruktúru. Rozdiel oproti Code elementu je ten, že Value element môže nadobúdať ľubovoľné[13] hodnoty, nie len jednu z piatich definovaných SOAPom.

Reason, na rozdiel od Code, nie je určený na algoritmické spracovanie, ale mal by obsahovať človekom pochopiteľné vysvetlenie chyby. Reason môže obsahovať jeden alebo viac Text elementov, pričom každý popisuje chybu v nejakom jazyku (povinný atribút xml:lang).

Nepovinný element Node obsahuje URI, ktoré identifikuje uzol, ktorý vygeneroval túto chybu. Keď tento element nie je uvedený, predpokladá sa konečný príjemca. Každý okrem konečného príjemcu musí tento element v prípade vygenerovania chyby uviesť. Pozn.: nejde o URI *role* tohto uzlu ale URI *samotného uzlu*[14].

Nepovinný element Role špecifikuje rolu, v ktorej uzol vystupoval, keď vygeneroval túto chybu (fault). V prípade, že nie je uvedený, predpokladá sa rola ultimateReceiver.

---

[12] Vzhľadom na to že je to dosť komplikovane a v našom návrhu prostredníci nebudú existovať. Čitateľ v prípade záujmu môže konzultovať príslušnú časť špecifikácie.
[13] Ľubovoľné hodnoty, ale musia byt namespace qualified.
[14] Spomeňme si, že každý SOAP uzol je identifikovaný pomocou URI – viď zoznam pojmov na začiatku kapitoly. URI SOAP uzlu bude v praxi URL danej SOAP aplikácie v sieti.

Nepovinný element Detail je určený na prenos aplikačne špecifických chybových informácií a môže obsahovať 0 alebo viac ľubovoľných XML elementov popisujúcich tieto chybové informácie.

V nasledovnom diagrame máme načrtnutý príklad chyby, ktorá by mohla byť vygenerovaná v scenári nášho príkladu v prípade, že by klient poslal chybnú požiadavku – napríklad by zle špecifikoval účet prijímateľa. Element Fault/Code/Value[15] nadobúda hodnotu Sender, čo znamená, že toto bola chyba na strane odosielateľa. Fault/Code/Subcode/Value nadobúda aplikačne špecifickú hodnotu (z nášho namespace). Vo Fault/Reason je uvedený človekom pochopiteľný popis chyby. Vo Fault/Detail je uvedený aplikačne špecifický element myFaultDetails, ktorý chybu podrobnejšie popisuje. Nepovinné elementy Fault/Role a Fault/Node nie sú uvedené, čo znamená, že túto chybu vygeneroval konečný príjemca – v našom prípade bankový server vybavujúci požiadavky o bankové prevody.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:e="http://example.org/faults">
   <env:Body>
      <env:Fault>
         <env:Code>
            <env:Value>env:Sender</env:Value>
            <env:Subcode>
               <env:Value>e:BadRequest</env:Value>
            </env:Subcode>
         </env:Code>
         <env:Reason>
            <env:Text xml:lang="sk">Chybna poziadavka</env:Text>
         </env:Reason>
         <env:Detail>
            <e:myFaultDetails>
               <e:message>Ucet prijimatela neexistuje</e:message>
               <e:errorcode>999</e:errorcode>
            </e:myFaultDetails>
         </env:Detail>
      </env:Fault>
   </env:Body>
</env:Envelope>
```

Diagram 5: Štruktúra SOAP Fault elementu

1.2.9 SOAP protocol bindings

K „defacto“ výmene SOAP správ cez počítačovú sieť dochádza prostredníctvom nižšieho protokolu, resp. pomocou naviazania SOAPu na niektorý nižší protokol, ktorý plní úlohu transportného protokolu. Množina pravidiel na prenos SOAP správy pomocou nižšieho protokolu sa nazýva binding (naviazanie).

Nebudeme uvádzať všetky pravidlá, ktoré musí takýto binding obsahovať, ako musí byť vytvorený a aké špecifikácie spĺňať, ale uvedieme si rovno príklad takéhoto naviazania – bude to naviazanie na protokol HTTP (HTTP binding). Je to najpoužívanejšie naviazanie, budeme ho používať aj my v našom návrhu a väčšinu vecí, ktoré tu konkrétne popíšeme, je ideovo podobná pre všetky naviazania.

[15] Element Value, ktorý je potomok elementu Code, ktorý je potomok elementu Fault. Takto budeme aj v ďalšom texte identifikovať XML elementy v rámci štruktúry XML dokumentu.

1.2.10 HTTP binding

Ide o naviazanie SOAPu na protokol HTTP – toto naviazanie teda umožňuje výmenu SOAP správ pomocou HTTP.
SOAP HTTP binding špecifikuje dva Message Exchange Patterns – *response* a *request/response*. Pozrime sa bližšie na request/response model komunikácie prostredníctvom HTTP. Správy sa vymieňajú medzi dvojicou uzlov, pričom jeden z nich je klient (requesting SOAP Node) a posiela požiadavky, Druhý z nich je server (responding SOAP Node) a ten posiela odpovede na požiadavky. SOAP požiadavka/odpoveď je celkom prirodzeným spôsobom prenášaná v HTTP požiadavke alebo odpovedi. Príklad HTTP požiadavky a odpovede so SOAP obálkami je na diagramoch (6) a (7) .

```
POST /FundsTransfers HTTP/1.1
Host: somebank.com
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: 3247

<?XML version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  ...
</env:Envelope>
```

Diagram 6: SOAP/HTTP požiadavka

Pre request/response pattern sa používa vždy HTTP metóda POST, requestURI spolu s hlavičkou Host špecifikuje URI cieľového SOAP uzla (serveru). Content-Type hlavička s hodnotou soap+xml špecifikuje, že telo HTTP požiadavky obsahuje SOAP obálku. Content-Length špecifikuje dĺžku tela HTTP požiadavky. V tele HTTP požiadavky je už samotná SOAP obálka. Túto požiadavku server prijme a môže odpovedať napríklad takto.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: 1608

<?XML version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  ...
</env:Envelope>
```

Diagram 7: SOAP/HTTP odpoveď

V tomto prípade bola HTTP požiadavka so SOAP správou prijatá a spracovaná úspešne a server posiela odpoveď.
Na prvom riadku je uvedený HTTP response code, ktorý zodpovedá stavu odpovede, resp. stavu vybavenia požiadavky. V tomto prípade je to kód 200, ktorý naznačuje úspešné vybavenie. V tele HTTP odpovede je už samotná SOAP obálka s odpoveďou.

1.2.11 Signalizovanie chýb pomocou HTTP

Pri komunikácii medzi klientom s serverom môže dôjsť k zlyhaniam na niektorej z komunikujúcich strán. Takéto zlyhania môžu byť tiché pre ľubovoľný z uzlov (signalizovanie chyby sa nedá „zaručiť“), alebo môže byť vygenerovaná SOAP chyba (fault) alebo HTTP špecifická chyba. Čiže zlyhania môžu nastať buď pri prenose na transportnom protokole alebo pri spracovaní SOAP správy. Tomu zodpovedajú vygenerované chyby.
V prípade že nezlyhá transportný protokol, tak HTTP požiadavka aj HTTP odpoveď obsahujú SOAP obálky.

Predpokladajme, že došlo k chybe – HTTP špecifickej alebo SOAP chybe. Ako sú tieto chyby signalizovane klientovi[16]? V prípade HTTP špecifickej chyby vráti HTTP odpoveď so status kódom, ktorý zodpovedá danej chybe (4xx a 5xx), pričom SOAP obálka nie je súčasťou odpovede. 4xx kódy naznačujú chybu na strane klienta (chyby v požiadavke) a 5xx kódy naznačujú chybu/zlyhanie na strane serveru.
Druhá možnosť je, že ide o SOAP špecifickú chybu. SOAP serverom je teda vygenerovaná nejaká SOAP chyba (fault). V tomto prípade klient obdrží HTTP odpoveď, ktorá obsahuje aj SOAP obálku. HTTP status kód je v tomto prípade determinovaný typom SOAP chyby. Presne namapovanie je v tabuľke (1).

SOAP Fault	HTTP Status Code	HTTP Reason Phrase
env:VersionMismatch	500	Internal server error
env:MustUnderstand	500	Internal server error
env:Sender	400	Bad request
env:Receiver	500	Internal server error
env:DataEncodingUnknown	500	Internal server error

Tabuľka 1: Namapovanie SOAP Faults na HTTP status kódy

Čiže vo všeobecnosti, klient pošle požiadavku a čaká na odpoveď. Môže nastať niekoľko možnosti. V prípade, že sa mu vráti odpoveď s kódom 200, požiadavka bola úspešne prijatá a spracovaná a HTTP odpoveď obsahuje aj SOAP obálku. V prípade odpovede s kódmi 3xx ide o presmerovanie. Je to HTTP špecifická odpoveď a klient by sa mal pokúsiť poslať požiadavku ešte raz na novú adresu (URI) – Táto je poskytnutá v odpovedi.
V prípade, že sa mu vráti kód z triedy 4xx alebo 5xx, odpoveď môže a nemusí obsahovať SOAP obálku. 4xx a 5xx kódy sú rozobraté v predošlom odstavci.
V prípade, že klient obdrží HTTP odpoveď so status kódom, ktorého presný význam nepozná, redukuje ho na základný x00 kód danej triedy (napr. neznámy kód 487 môže zredukovať na 400).

Stručne zhrnuté, SOAP nám umožňuje výmenu ľubovoľných XML informácií vnútri SOAP Obálok medzi uzlami v sieti (SOAP uzlami) pomocou naviazania na transportný protokol siete. Definuje pravidlá na spracovanie týchto správ uzlami za cieľom ich správneho doručenia a spracovania. Pre pripadne chyby, ktoré môžu pri tomto vzniknúť, je definovaný mechanizmus ich popisu a prenosu.

V nasledujúcej časti si povieme niečo o jazyku WSDL, ktorý (v prípade použitia spolu so SOAPom) definuje, ako prostredníctvom výmeny SOAP správ realizovať operácie webovej služby.

1.3 WSDL

1.3.1 Úvod

WSDL (Web Service Description Language) je jazyk na popis webových služieb. Umožňuje nám definovať webservis ako množinu súvisiacich operácií vystavených v nejakom bode počítačovej siete, v ktorom k nemu môžu pristupovať klienti.

[16] Chyby zrejme signalizuje iba server klientovi a nie naopak.

V nasledovnom texte popisujeme verziu 1.1 tohto jazyka[17]. Vychádzame najmä z [WSDL11].

WSDL teda umožňuje definovať webové **služby**. Služba je definovaná ako kolekcia **portov** – koncových bodov v sieti. Každý port vystavuje niekoľko **operácií**. Každá operácia je definovaná ako výmena jednej alebo viacerých **správ**. Každá správa je definovaná na základe dátových **typov**. Všetky tieto prvky môžeme definovať pomocou WSDL.

Za cieľom znovupoužitia sú vo WSDL niektoré prvky (správy, operácie, porty) definované abstraktne a špecifikovaním konkrétneho protokolu pre transport a formát dát sú definované konkrétne prvky definujúce samotnú službu. WSDL umožňuje definovať nasledovné:
Types (dátové typy) – dátové typy a elementy definované spravidla pomocou XML Schémy. Tieto sú neskôr použité na definíciu správ.
Message (správa) – abstraktná definícia dát (informácií), ktoré sú prenášané.
Operation (operácia)– abstraktný popis akcie podporovanej službou. Je definovaná ako výmena niekoľkých správ.
Port Type (abstraktný port) – množina abstraktných operácií.
Binding (naviazanie) – „skonkrétnenie" abstraktného portu. Špecifikácia konkrétneho protokolu, formátu a štruktúry dát pre správy a operácie abstraktného portu.
Port – koncový bod definovaný ako dvojica (binding, sieťová adresa).
Service (služba) – množina portov.

Popis týchto prvkov a ich vzťahy sú načrtnuté na nasledovnom diagrame.

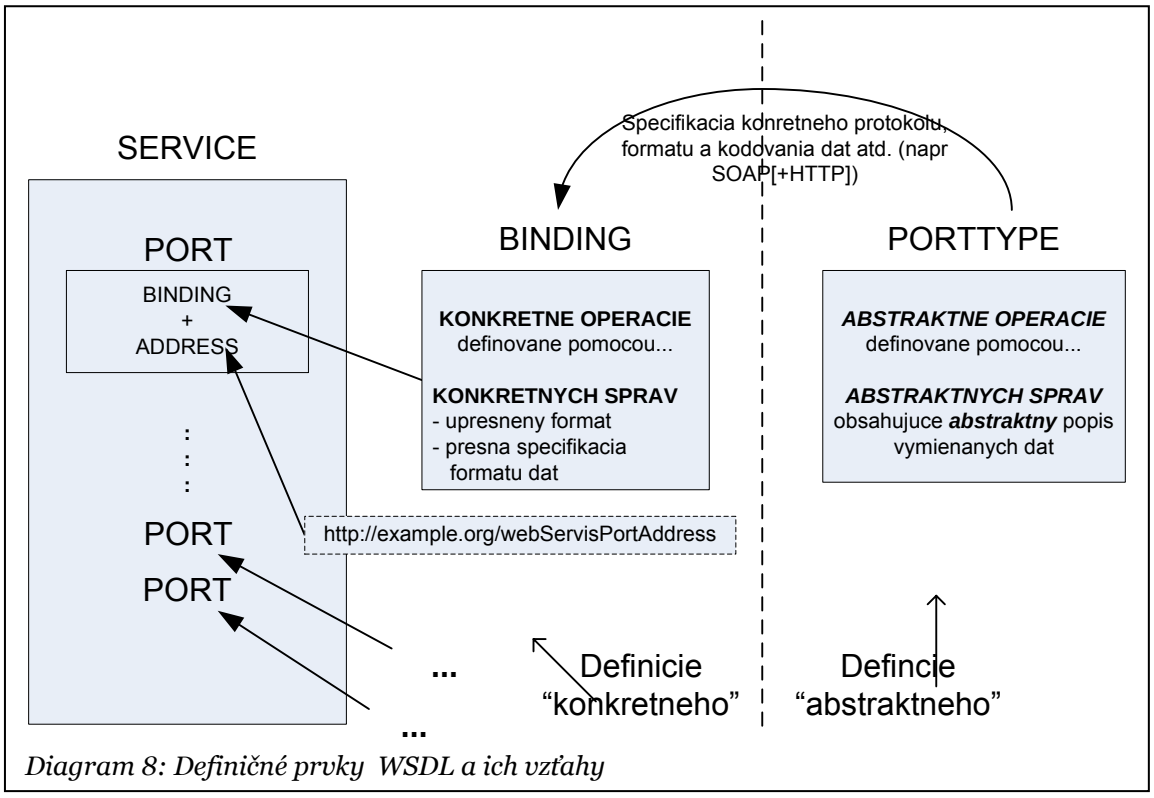


Diagram 8: Definičné prvky WSDL a ich vzťahy

[17] Pracuje sa na špecifikácii WSDL 2.0, ktorá je v momente písania tohto textu v stave Recommendation Candidate vo W3C. Viď [WSDL2].

1.3.2 Štruktúra WSDL dokumentu

WSDL dokument je dokument (XML súbor) obsahujúci hore uvedené prvky WSDL jazyka použité na definovanie nejakej konkrétnej webovej služby. V nasledujúcich odstavoch popíšeme štruktúru WSDL dokumentu všeobecne a vytvoríme WSDL dokument definujúci webservis pre náš príklad z kapitoly o SOAPe.
Globálne vyzerá štruktúra WSDL dokumentu nasledovne.

```
<?XML version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:t="http://example.org/transfer"
             targetNamespace="http://example.org/transfer">

   <!--TYPY (types)-->
   <!--SPRÁVY (message)-->
   <!--PORT TYPES (portType)-->
   <!--BINDINGS (binding)-->
   <!--SERVICES (service)-->

</definitions>
```

Diagram 9: Štruktúra WSDL dokumentu

Koreňový element `definitions` obsahuje definície niektorých alebo všetkých spomínaných prvkov (typy, správy, porty, ...). Okrem toho `definitions` element obsahuje deklarácie potrebných menných priestorov. Pozrime sa teraz, ako vyzerajú definície jednotlivých prvkov. Pre potreby popisu rozdelíme WSDL dokument na 5 imaginárnych sekcií, pričom každá bude označovať relevantné definície pre daný prvok (typy, správy, porty...).

1.3.3 Types

Typy sú definované v elemente `definitions/types`. Tento obsahuje definície dátových typov a elementov pomocou XML Schémy (xsd:simpleType a xsd:complexType; a xsd:element).

```
<types>
   <schema targetNamespace="http://example.org/transfer"
           xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
      <xsd:element name="TransferRequest">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="FromAccount" type="xsd:string"/>
               <xsd:element name="ToAccount" type="xsd:string"/>
               <xsd:element name="Amount" type="xsd:integer"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
      <xsd:element name="TransferResponse">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="Result" type="xsd:string"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
      <xsd:element name="TransferType">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="priority" type="xsd:string"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
   </schema>
</types>
```

Diagram 10: Definícia typov vo WSDL dokumente

Pre potreby nášho príkladu sú definované 3 elementy. Prvý z nich, `TransferRequest`, predstavuje žiadosť klienta o vykonanie prevodu a obsahuje tri povinné položky (elementy) identifikujúce účty, medzi ktorými sa majú peniaze previesť a množstvo peňazí pre prevod. Čísla účtov sú typu xsd:string, teda textové reťazce; množstvo je typu xsd:integer, čo je celé číslo. Element `TransferResponse` obsahuje jeden povinný element `Result` (xsd:string), ktorý bude zrejme obsahovať odpoveď serveru na požiadavku o prevod. Nakoniec je tu element `TransferType`, ktorý je použitý ako hlavičkový blok v hlavičke SOAP správy.

1.3.4 Message

Každá správa je definovaná ako element `definitions/message` s unikátnym menom (parameter `name`). Každá správa sa skladá z niekoľkých časti, čomu zodpovedajú elementy `part` vnútri `message` elementu, pričom každá časť je definovaná na základe typu (parameter `type`) alebo elementu (parameter `element`) definovaného v sekcii Typy.

```
<message name="TransferRequestMsg">
   <part name="parameter" element="t:TransferRequest"/>
</message>

<message name="TransferResponseMsg">
   <part name="parameter" element="t:TransferResponse"/>
</message>

<message name="Headers">
   <part name="transferTypeHeader" element="t:TransferType"/>
</message>
```

Diagram 11: Definícia správ vo WSDL dokumente

V našom prípade definujeme tri správy. Každá z nich sa skladá iba z jednej časti a každá z nich je definovaná pomocou elementu XML schémy. Ako uvidíme neskôr,

`TransferRequestMsg` a `TransferResponseMsg` budú slúžiť na definovanie operácie abstraktného portu a jediná časť správy `Headers` bude slúžiť ako hlavičkový blok v prenášaných SOAP správach.

1.3.5 Port type

Každý abstraktný port je definovaný pomocou elementu `portType`. Tento element obsahuje definíciu niekoľkých abstraktných operácií (elementy `portType/operation`). WSDL definuje štyri typy operácií:
One-way – klient pošle správu webservisu.
Request-response – klient pošle správu webservisu a ten pošle späť odpoveď (správu).
Solicit-response – webservis pošle správu klientovi a ten pošle späť odpoveď (správu).
Notification – Web servis pošle správu klientovi.

```
<portType name="BankingPortType">
   <operation name="doTransfer">
      <input message="t:TransferRequestMsg"/>
      <output message="t:TransferResponseMsg"/>
   </operation>
</portType>
```

Diagram 12: Definícia abstraktných portov vo WSDL dokumente

Pre request-response operácie portType definuje okrem mena (parameter `name`) vstupnú a výstupnú správu pre danú operáciu – v našom prípade už dobre známe TransferRequestMsg a TransferResponseMsg. Okrem toho tu môže byť uvedených 0 alebo viac fault elementov, ktoré definujú aplikačne špecifické chyby pri vykonávaní danej operácie. Tieto môžu byť v prípade vygenerovania chyby uvedené v SOAP `Fault/Detail` elemente (viď príslušný odsek v kapitole o SOAPe). My kvôli jednoduchosti žiadnu chybu pre nás príklad na tomto mieste nedefinujeme, aj keď sme ju v kapitole o SOAPe v našom príklade použili.

1.3.6 Binding

Binding (pre daný port type) definuje formát správ a popisuje ako sú abstraktné operácie a správy namapované na ich konkrétne ekvivalenty pre daný protokol. Pre jeden port type môže existovať ľubovoľný počet naviazaní. My sa pre naše potreby uspokojíme s popisom jedného konkrétneho typu naviazania – naviazanie na SOAP protokol. Binding pre nás webservis vyzerá nasledovne[18]:

```
<binding name="BankingSoapBinding" type="t:BankingPortType">
   <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="doTransfer">
      <soap:operation soapAction="http://example.com/doMoneyTransfer"/>
      <input>
         <soap:body use="literal"/>
         <soap:header message="t:Headers" part="transferTypeHeader" use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
</binding>
```

Diagram 13: Definícia naviazaní (bindings) vo WSLD dokumente

Binding je definovaný pomocou elementu `binding`. Atribút `name` tohto elementu definuje unikátne meno tohto naviazania a element `type` uvádza, pre ktorý abstraktný port toto naviazanie definujeme.

[18] Tu uvádzame naviazanie na SOAP 1.1.

Samotne naviazanie je definované tzv. *rozširujúcimi elementami*, ktoré uvádzame v rámci elementu `binding` a jeho podelementov. V prípade naviazania na SOAP budú tieto elementy z menného priestoru SOAPu (s prefixom `soap`), napr. `soap:binding`, `soap:body`, `soap:header` a `soap:fault`.

Ako prvý element v binding elemente je uvedený `soap:binding` element. To znamená, že tento binding definuje naviazanie na SOAP protokol a štruktúru SOAP Envelope: Header a Body. Atribút `style` nadobúda jednu z hodnôt `document`, `rpc` a určuje default hodnotu pre „štýl" operácií „naväzovanych" v rámci tohto binding elementu. RPC znamená, že správy budú obsahovať parametre a návratové hodnoty (čo zodpovedá volaniu funkcie) a document znamená, že správy budú obsahovať dokumenty. Hodnota `transport` atribútu definuje, na ktorý protokol je naviazaný samotný SOAP. V tomto prípade je to protokol HTTP, ktorý je identifikovaný pomocou URI `http://schemas.xmlsoap.org/soap/http`.

Binding ďalej obsahuje niekoľko elementov `operation`, ktoré „skonkrétňujú" operácie daného abstraktného portu. V našom prípade ide o jedinú operáciu s názvom doTransfer. Vnútri `operation` elementu je uvedený `soap:operation` element, ktorý špecifikuje `soapAction`, čo je atribút, ktorý je pre SOAP 1.1 povinný a jeho hodnota sa posiela v hlavičke HTTP požiadavky klienta pri volaní danej operácie. Atribút `style` elementu `soap:operation` určuje, či Táto operácia je v štýle rpc alebo document. My ho v našom príklade neuvádzame – použije sa default hodnota `style` atribútu zo `soap:binding`. Element `operation` môže obsahovať aj elementy `input` a `output`, ktoré definujú obsah SOAP obálky pre vstupnú a výstupnú správu operácie.

Element `soap:body` špecifikuje, ako sa jednotlivé časti správy (definovane elementami `part` v rámci elementu message) zobrazia do SOAP Body. Ak štýl operácie je rpc, každá časť správy je buď parameter (v prípade vstupnej správy) alebo návratová hodnota (v prípade výstupnej správy). Všetky časti správy (každá z nich reprezentujúca jeden parameter) sú uvedená vnútri elementu s názvom identickým názvu operácie. Tento element bude priamy potomok SOAP Body. Ak je štýl operácie document, časti správy (parts) sa zobrazia priamo v SOAP Body ako XML elementy. Povinný atribút `use` nadobúda jednu z hodnôt `literal`, `encoded`. Hodnota `literal` znamená, že každá časť (`part`) je definovaná ako konkrétny (literálny) element alebo typ pomocou XML schémy. Hodnota `encoded` znamená, že každá časť referencuje abstraktný typ (je teda použitý atribút `type`, nie `element`) a z tohto typu je potom vytvorená konkrétna časť správy pomocou kódovania uvedeného v atribúte encodingStyle. V prípade, že špecifikované kódovanie umožňuje variácie vo formáte pre daný abstraktný typ (čo je prípad aj SOAP kódovania), tak komunikujúce strany musia podporovať všetky variácie. Táto flexibilita až nejednoznačnosť spôsobuje problémy s interoperabilitou a aj preto WS-I Basic Profile (viď [WSIBP]) zakazuje použitie SOAP kódovania a ľubovoľného iného kódovania – implementácie konformne s odporúčaniami musia použiť `literal`.
Nepovinný `parts` atribút môže špecifikovať, ktoré časti správy majú byť v SOAP body uvedené. V prípade, že ho neuvedieme (náš prípad), budú v SOAP Body uvedené všetky časti správy. Okrem toho soap:body môže obsahovať ešte niekoľko atribútov, ktoré sú uvedené iba v prípade, že use="encoded", ale týmito sa nebudeme zaoberať.

Elementy `soap:header` umožňujú definovať hlavičkové bloky, ktoré budú prenesene v hlavičke SOAP obálky. Atribúty `message` a `part` určia (vyberú) časť správy, ktorá sa použije ako hlavičkový blok. Ostatne atribúty tohto elementu sú podobné atribútom `soap:body`.

Ďalej môžu byť v rámci operation definované elementy, ktoré prenášajú informácie o chybách vygenerovaných SOAP uzlami (elementy `soap:fault` a `soap:headerfault`), ale týmito sa nebudeme podrobnejšie zaoberať.

1.3.7 Service

Service definuje samotnú (webovú) službu a je definovaný pomocou elementu `definitions/service`. Je to množina súvisiacich (konkrétnych) portov – element `service` obsahuje niekoľko elementov `port`. Ako sme spomínali, port je definovaný ako dvojica (binding, adresa). Binding pre tento port je uvedený v atribúte binding a adresa je (pre SOAP Binding) uvedená v elemente `port/soap:address`.

```
<service name="BankingService">
   <port name="BankingPort" binding="t:BankingSoapBinding">
      <soap:address location="http://example.com/BankingWS"/>
   </port>
</service>
```

Diagram 14: Definícia služieb vo WSDL dokumente

V našom prípade definujeme službu s názvom BankingService, ktorá obsahuje jeden port BankingPort definovaný na základe nami definovaného naviazania. Tento port služby je prístupný na URI `http://example.com/BankingWS`.

1.4 XML Signature

Úvod

XML Signature špecifikácia (viď [XMLSig]) definuje XML syntax a pravidlá spracovania pre vytváranie a reprezentáciu digitálnych podpisov. Je špecifikovaný spôsob, ako je možné vytvoriť digitálny podpis ľubovoľného XML dokumentu, jeho časti alebo aj dát externých pre daný XML dokument a ako tento podpis reprezentovať v XML štruktúre a umožniť tak jeho vloženie do ľubovoľného XML dokumentu.

XML Signature nám umožňuje zviazať (podpisový) kľúč s (podpisovanými) dátami, čo z kryptografického hľadiska poskytuje prostriedky na zabezpečenie a autentifikácie zdroju dát, t.j. odosielateľa[19]. Digitálny podpis okrem toho zaručuje integritu podpísaných dát.

Stručne zhrnuté, podpísanie dát funguje nasledovne: Najprv sú lokalizovane dáta, ktoré sa majú podpísať. Následne je vytvorený hash týchto dát, je umiestnený do špeciálneho elementu spolu s ďalšími informáciami a výsledok je kryptograficky podpísaný.

[19] XML Signature nešpecifikuje ako sú zviazané kľúče s osobami/inštitúciami a takisto nešpecifikuje význam podpisovaných daťa. Čiže tato špecifikácia nie je postačujúcou podmienkou „bezpečnosti“, iba poskytuje prostriedky na dosiahnutie týchto cieľov.

```
<Signature ID?>
   <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
     (<Reference URI? >
        (<Transforms>)?
         <DigestMethod>
         <DigestValue>
      </Reference>)+
   </SignedInfo>
   <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
</Signature>
```

Diagram 15: Štruktúra elementu Signature

Štruktúra

XML podpisy sú reprezentované pomocou elementu `Signature`, ktorého štruktúra[20] je načrtnutá na diagrame (Diagram 15). Element `Signature/SignedInfo` je element, ktorý sa v konečnom dôsledku podpisuje a obsahuje informácie o tom, čo (elementy `Reference`) a akým spôsobom (`SignatureMethod`) je podpísané. `Signature/SignatureValue` obsahuje samotnú hodnotu digitálneho podpisu. Nepovinný `KeyInfo` môže obsahovať informácie o kľúči, pomocou ktorého môžeme tento podpis overiť. Element `Object` môže obsahovať podpisované dáta – pre prípad, že chceme aby podpisované dáta boli obsiahnuté priamo v `Signature` elemente.

Nepovinný XML ID atribút jednoznačne identifikuje element (napr. `Signature` alebo `Object`) v danom dokumente. Musí byť unikátny – t.j. žiadne dva rôzne XML elementy v danom dokumente nemôžu mat ID atribúty rovnakej hodnoty.

`SignedInfo/CanonicalizationMethod` špecifikuje metódu „kanonikalizácie“ elementu `SignedInfo`. XML elementy s významovo rovnakým obsahom totižto nemusia byť znak po znaku identické reťazce znakov (iné množstvo bieleho priestoru, iné poradie atribútov pre daný element atd.) a v prípade zhashovania a podpísania by sme mohli dostať dva rôzne podpisy rovnakých dát.
`SignedInfo/SignatureMethod` špecifikuje metódu podpísania. Napr. kombinácia SHA a RSA znamená, že dáta sa najprv zhashuju pomocou sha a výsledok je následne zašifrovaný privátnym RSA kľúčom – dostávame digitálny podpis.
`SignedInfo/Reference` pomocou atribútu `URI` identifikuje dátový objekt na podpísanie. Môže to byť napr. lokálna referencia (viď príklad nižšie) alebo URL na externý objekt. V prípade, že chceme podpísať viac dátových objektov naraz, `Reference` sa môže opakovať.
Nepovinný `Reference/Transforms` element špecifikuje, ako je referencovaný dátový objekt transformovaný predtým ako je zhashovaný. V prípade, že dátový objekt je XML fragment, veľmi pravdepodobne budeme chcieť vykonať minimálne jeho kanonikalizáciu.
`Reference/DigestMethod` špecifikuje metódu, ktorou sa (po prípadných transformáciách) zhashuje fragment, na ktorý odkazuje táto `Reference`.
`Reference/DigestValue` obsahuje hash tohto fragmentu.

Typy podpisov: enveloped, enveloping, detached

XML Signature rozoznáva tri rôzne variácie XML podpisu: zaobaľujúci (enveloping), zaobalený (enveloped) a oddelený (detached). Pri zaobaľujúcom podpise sú podpisované dáta uložené v elemente Object priamo v Signature elemente. Pri zaobalenom podpise je Signature element uložený vnútri samotného podpisovaného fragmentu. Pri oddelenom podpise sú Signature a podpisované dáta oddelené – Signature nie je potomkom podpisovaného XML fragmentu ani naopak. Posledný spomínaný typ vyzerá na prvý pohľad „najčistejší“. Používa sa napríklad aj vo WS Security, kde podpisy sú uvedené separátne v jednom hlavičkovom bloku SOAP hlavičky.

[20] ? reprezentuje 0 alebo 1 výskyt, + reprezentuje jeden a viac výskytov, * reprezentuje 0 a viac výskytov

Príklad

Pozrime sa na príklad použitia XML Signature v praxi. My v našom návrhu budeme používať dva typy podpisov, oddelený a zaobalený. Tu si uvádzame príklad zaobaleného podpisu. Podpísaný je element `TransferRequest` z nášho SOAP príkladu. Pribudol `Id` atribút pre `TransferRequest`, ktorý slúži na jeho identifikáciu v atribúte `URI` v elemente `Signature/Reference`. Okrem toho si všimnime, že v transformáciách je ako prvá uvedená transformácia pre zaobalene podpisy. Táto pri overovaní podpisu z `TransferRequest` najprv odstráni `Signature` element a až následne ho kanonikalizuje a zhashuje.

```
<t:TransferRequest xmlns:t="http://example.org/transfer" Id="TR007">
   <t:FromAccount>3492837564/1100</t:FromAccount>
   <t:ToAccount>7480293410/900</t:ToAccount>
   <t:Amount>1000</t:Amount>
   <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SignedInfo>
         <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
         <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
         <Reference URI="#TR007">
            <Transforms>
               <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
               <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            </Transforms>
            <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
         </Reference>
      </SignedInfo>
      <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
   </Signature>
</t:TransferRequest>
```

Diagram 16: Príklad zaobaleného podpisu

Proces vygenerovania podpisu

Pozrime sa teraz, ako je vygenerovaný podpis, presnejšie, samotný `Signature` element a jeho obsah.
V skratke – najprv sú identifikovane všetky referované dáta na podpis (XML fragmenty alebo externé dáta), pre každý z nich sú aplikované transformácie špecifikované pre referovaný fragment, výsledok je zhashovaný a je vytvorený `Reference` element.
Následne je vytvorený `SignedInfo` element (obsahujúci `SignatureMethod`, `CanonicalizationMethod`, a `Reference` elementy), tento je kanonikalizovaný a na základe algoritmov špecifikovaných v `SignatureMethod` je vygenerovaný podpis – hodnota `SignatureValue` elementu.
Nakoniec sa vytvorí `Signature` element obsahujúci `SignedInfo`, `SignatureValue` a pripadne `KeyInfo` a `Object` elementy.

Proces validácie podpisu

Predpokladajme, že máme XML dokument a v ňom podpis (`Signature` element).
Validácia tohto podpisu prebieha v dvoch krokoch. Prvým je validácia referencii, druhým validácia samotného podpisu.
Validácia referencii:
Najprv je element `SignedInfo` kanonikalizovaný podľa algoritmu uvedeného v `SignedInfo/CanonicalizationMethod`. Následne pre každý SignedInfo/Reference element získame dáta na podpis, aplikujeme pripadné transformácie a výsledok zhashujeme použitím metódy uvedenej v `Reference/DigestMethod`. Túto hodnotu porovnáme s hodnotou uvedenou v `DigestValue` elemente tejto referencie. Ak nie sú rovnaké, validácia zlyhá.
Validácia podpisu:

Najprv získame kľúč na overenie podpisu (buď na základe informácií uvedených v `KeyInfo`, alebo z externého zdroja, alebo ho vieme implicitne[21]). Vygenerujeme hash kanonikalizovaného `SignedInfo` a overíme, či sa výsledok zhoduje s dešifrovaným hashom (dešifrovaný obsah `SignatureValue`).

1.5 WS Security

WS Security (viď [WSSec]) je štandard, ktorého cieľom je umožniť bezpečnú výmenu SOAP správ medzi SOAP uzlami. WS Security poskytuje prostriedky na zabezpečenie integrity a dôvernosti prenášaných informácií, autentifikáciu správ a kryptografické nepopretie (non-repudiation)[22].

Na zabezpečenie integrity a autentifikáciu správ je použitý štandard XML Signature a na zabezpečenie dôvernosti sa používa XML Encryption. Okrem je možné použiť tzv. *bezpečnostné tokeny*. Token je XML element obsahujúci nejaké *tvrdenia* (claims). Napríklad `UsernameToken` špecifikovaný v tomto štandarde umožňuje odosielateľovi uviesť užívateľské meno a heslo, čiže obsahuje *autentifikačné* tvrdenia odosielateľa. Ďalšie typy tokenov sú napr. X.509 certifikát alebo Kerberos ticket.

WS Security je príkladom SOAP Feature, teda rozšírenia SOAP protokolu, ktoré je realizované pomocou SOAP Module, ktorým je SOAP hlavičkový blok Security[23] definovaný touto špecifikáciou. Security element obsahuje bezpečnostné informácie pre príjemcu – tokeny, referencie na tokeny, XML Signature elementy a elementy súvisiace s XML Encryption. Kvôli rozšíriteľnosti je možné vložiť do Security elementu aj ľubovoľný iný element. WS Security popisuje ako použiť elementy týkajúce sa bezpečnosti v rámci Security hlavičkového bloku, nezakazuje však uviesť napr. digitálne podpisy mimo neho (čo bude prípad nášho návrhu). V tomto prípade sa však nejedná o použitie podpisov v rámci WS Security štandardu, ale nezávislé použitie. Umiestnenie a štruktúra Security hlavičkového bloku je nasledovná.

```
<env:Envelope>
   <env:Header>
      <wsse:Security xmlns:wsse="...">
         <!-- tokeny -->
         <!-- XML Signature elementy -->
         <!-- XML Encryption elementy -->
      </wsse:Security>
   </env:Header>
   <env:Body>
      ...
   </env:Body>
</env:Envelope>
```

Diagram 17: WS Security hlavičkový blok

[21] V prípade komunikácie banky a zákazníka zrejme komunikujúce strany dopredu navzájom poznajú svoje verejne kľúče.
[22] WS Security explicitne spomína nepopretie ako „non-goal“ špecifikácie, ale umožňuje digitálne podpísať časti dokumentu, čo *je* prostriedok na dosiahnutie (kryptografického) nepopretia, ktoré môžeme chápať ako zviazanie podpisovaných dať s podpisovým kľúčom. V reále pre nepopretie z právneho hľadiska treba zabezpečiť ešte plejádu iných podmienok.
[23] Z príslušného namespace a s definovanou sémantikou.

My v našom návrhu budeme z WS Security používať iba mechanizmus digitálnych podpisov. XML Encryption, tokeny a iné mechanizmy používať nebudeme. Bezpečnostné ciele, ktoré tieto umožňujú dosiahnuť budeme realizovať iným spôsobom.

Digitálne podpisy

Podpis správy (pripadne jej časti) umožňuje príjemcovi overiť integritu správy a autentifikovať ju, t.j. overiť, že správa bola odoslaná držiteľom podpisového kľúča. `Security` element môže obsahovať ľubovoľný počet `Signature` elementov, pričom tieto môžu podpisovať ľubovoľné (aj prekrývajúce sa) časti SOAP správy. WS Security používa najmä oddelené (detached) podpisy. Podpisovať môžeme SOAP Body alebo jeho časti, hlavičkové bloky, ale aj elementy v rámci Security elementu (napr. tokeny). Je možné použiť aj zaobaľujúci a zaobalený podpis. WS-I Basic Security Profile však neodporúča ich použitie – zaobalený podpisu je neodporúčaný a zaobaľujúci podpis je dokonca zakázaný. Prvý z dôvodu možných konfliktov so SOAP procesným modelom[24] a Druhý vzhľadom na premenlivosť obsahu SOAP hlavičky[25].
Ako sme spomínali v texte o XML Signature, v našom návrhu budeme používať dva druhy podpisov, čomu budú zodpovedať dva odlišné scenáre, ktoré budeme v návrhu riešiť. Budeme chcieť podpisovať fragmenty tela SOAP správy a na toto použijeme zaobalené podpisy – pôjde teda o použitie podpisov *mimo* štandardu WS Security. Digitálne podpisy v rámci WS Security budeme používať iba pre druhý scenár, a to keď budeme chcieť podpísať *celé telo* SOAP správy. Príklad takéhoto použitia je uvedený na konci tejto podkapitoly.

Bezpečnostné tokeny a token referencie

WS Security špecifikuje niekoľko preddefinovaných typov tokenov a okrem toho umožňuje implementáciám použiť ľubovoľný iný token (custom token). Prvý preddefinovaný typ tokenu je `UsernameToken` slúžiaci na potreby identifikácie/autentifikácie. `BinarySecurityToken` umožňuje odosielateľovi uviesť bezpečnostný token, ktorý nie je natívne v XML formáte, napr. X.509 certifikát apod.
V prípade, že nechceme alebo nemôžeme uviesť token priamo v `Security` elemente, môžeme použiť element `SecurityTokenReference`. Tento umožňuje referencovať ľubovoľný token nachádzajúci sa v SOAP Obálke inde ako v `Security` elemente, ale aj token externý SOAP obálke (napr. identifikovaný pomocou URL).

Asociácia tokenov a podpisov za cieľom autentifikácie správy

Spoločným použitím tokenov a podpisov je možné prirodzeným spôsobom asociovať tvrdenia tokenu s obsahom správ. Predpokladajme, že Bob pomocou svojho privátneho kľúča podpísal časť správy a podpis vložil do hlavičkového bloku Security. Okrem toho v Security elemente uviedol token obsahujúci jeho X.509 certifikát (ktorý je podpísaný certifikačnou autoritou a obsahuje okrem iného Bobov verejný kľúč a informácie o Bobovi – meno, adresa, atd.). Ak sa prijímateľovi správy podarí overiť podpis pomocou Bobovho verejného kľúča a samotný certifikát je platný, vieme, že obsah správy podpísal držiteľ Bobovho privátneho kľúča, čiže pravdepodobne Bob. Takýmto spôsobom je prirodzene spojená identita Boba s obsahom podpísanej správy.
Uvedomme si však, že prítomnosť tokenov nie je vo všeobecnosti nutná na asociovanie kľúča (resp. identity jeho vlastníka) s obsahom správy. Prijímateľ môže napríklad Bobov kľúč dopredu poznať a keďže Bob je zároveň očakávaný odosielateľ tejto správy, prijímateľ vie, že má overiť podpis pomocou Bobovho kľúča. Tento prípad spomíname explicitne z toho dôvodu, že v našom návrhu bude dochádzať k autentifikácii pomocou iných mechanizmov ako WS Security tokenov a komunikujúce strany budú po autentifikácii

[24] Uzol spracovávajúci SOAP obálku by mohol mať problémy s rozoznaním hlavičkového bloku zaobaleného podpisom.
[25] Prostredníci môžu hlavičkové bloky pridávať, odoberať a modifikovať.

poznať identitu aj kľúč protistrany a teda budú môcť na základe existencie podpisu a týchto informácií vykonať autentifikáciu odosielateľa správy.

Ostatné

WS Security tiež špecifikuje ako použiť XML encryption na zašifrovanie (časti) správy a ako použiť časové pečiatky v rámci Security elementu. My však v našom návrhu XML Encryption ani časové pečiatky používať nebudeme, takže sa týmito časťami špecifikácie nebudeme zaoberať.

Chybové scenáre

Pri spracovaní bezpečnostných informácií uvedených v Security elemente môže dôjsť k chybám. V našom prípade to budú – okrem chýb všeobecného charakteru – najmä chyby týkajúce sa podpisov a ich spracovania, keďže my v našom návrhu budeme používať z WS Security iba podpisy. V prípade, že dôjde ku chybe pri spracovaní, príjemca môže o tomto fakte notifikovať odosielateľa. Nemusí to však spraviť vždy vzhľadom na to, že môže ísť o pokus o Denial of service alebo kryptograficky útok. V prípade, že sa však príjemca rozhodne o chybe odosielateľa notifikovať, musí použiť Fault mechanizmus SOAPu.

Príklad

```
<env:Envelope xmlns:env="..." xmlns:wsu="..." xmlns:ds="...">
   <env:Header>
      <wsse:Security xmlns:wsse="...">
         <ds:Signature>
            <ds:SignedInfo>
               <ds:CanonicalizationMethod
                           Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
               <ds:SignatureMethod
                           Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
               <ds:Reference URI="#SOAPBody">
                  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                  <ds:DigestValue>LyLsF0Pi4wPU...</ds:DigestValue>
               </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>DJbchm5gK...</ds:SignatureValue>
         </ds:Signature>
      </wsse:Security>
   </env:Header>
   <env:Body Id="SOAPBody">
      <!-- telo správy -->
   </env:Body>
</env:Envelope>
```

Diagram 18: Príklad použitia digitálneho podpisu v rámci WS Security

2 Elektronické bankovníctvo

The trouble with being educated is that it takes a long time; it uses up the better part of your life and when you are finished what you know is that you would have benefited more by going into banking.
-- Philip K. Dick

2.1 Základné pojmy

V tejto časti si povieme niečo o bankovníctve. Zadefinujeme základné pojmy, ktoré budeme neskôr potrebovať a načrtneme jednoduchý model bankovníctva, ktorý nám umožní skúmať charakteristiky, ktoré nás zaujímajú v kontexte nášho návrhu.

Náš zjednodušený model bankovníctva si môžeme definovať ako trojicu:

1. Entity, ktoré sa v tomto modeli vyskytujú, ich vlastnosti a vzťahy medzi jednotlivými entitami.
2. Správanie entít a interakcie medzi entitami. Správanie entít bude dané operáciami, ktoré budú môcť entity vykonávať. Keďže jednotlivé operácie budú zahŕňať aj iné entity systému ako tú, ktorá danú operáciu iniciuje, prirodzene bude dochádzať k interakciám medzi nimi.
3. Komunikácia. Pri interakcii bude dochádzať ku komunikácii (výmene informácií) medzi entitami. Budeme skúmať charakteristiky tejto komunikácie – komunikačný kanál, požiadavky kladené na prenos informácií cez tento kanál a podobne.

2.1.1 Entity - kto a čo je v modeli

Peniaze – ľubovoľné „veci“, ktoré majú nejakú hodnotu a sú použiteľné na výmenu za iné hodnotne veci alebo opäť peniaze. Konkrétne pre naše potreby: mince a bankovky; a virtuálne peniaze[26].

Banka – inštitúcia, ktorá svojim zákazníkom poskytuje bankové služby – požičiavanie peňazí, prijímanie vkladov (peňazí), vykonávanie operácií s peniazmi a podobne - toto všetko za vopred dohodnutých podmienok. Banka interaguje so zákazníkmi a ostatnými bankami; predmetom interakcie sú peniaze.

Zákazník – osoba, ktorá využíva služby poskytované bankou. Ukladá si do banky peniaze a potom s nimi podľa dohody manipuluje, alebo si peniaze od banky požičiava a podľa dohody ich spláca.

Účet (zákazníka v banke) – virtuálne miesto v banke, kde sú uložené peniaze zákazníka. Čo je dôležitejšie, účet nám poskytuje rámec pre vzťah a interakciu medzi klientom a bankou. Konkrétnejšie, keď zákazník chce využívať služby banky a banka chce tieto služby poskytovať zákazníkovi, dohodnú sa na vytvorení (otvorení) účtu, ktorý bude

[26] Pri snahe o presnú definíciu hlavne virtuálnych peňazí by sme sa dostali do zbytočných komplikácií, ktorým sa chceme vyhnúť. Okrem toho, nie je pre naše potreby nevyhnutne. Môžeme si ich predstaviť ako niečo, čo reprezentuje skutočné peniaze, ma rovnaké vlastnosti, t.j. majú hodnotu a sú akceptované ako prostriedok výmeny hodnotných veci, pričom tesne po transakcii s virtuálnymi peniazmi dôjde aj k fyzickému premiestneniu zodpovedajúcich nevirtuálnych peňazí.

pre ďalšiu interakciu, týkajúcu sa peňazí, slúžiť[27]. Na účet potom bude možné peniaze vkladať, vyberať ich odtiaľ, vykonávať s peniazmi na účte rôzne operácie apod.

Na základe definície účtu môžeme definovať dôležitý pojem disponenta a preformulovať definíciu banky.

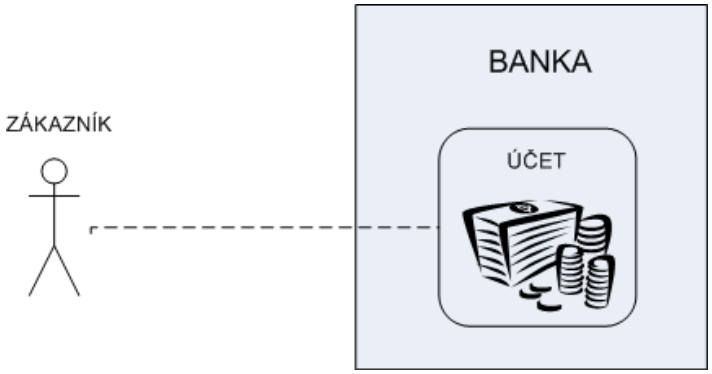
Disponent (účtu) – zákazník banky, ktorý môže vykonávať nejaké operácie na danom účte.

Vlastník účtu – zákazník banky, ktorý môže vykonávať všetky operácie povolené na účte a okrem toho môže účet zrušiť, určovať disponentov daného účtu a jednotlivé operácie, ktoré títo disponenti môžu vykonávať – čiže určovať práva jednotlivých disponentov.

Banka – inštitúcia, v ktorej sú vedené účty zákazníkov. Umožňuje disponentom vykonávať operácie nad účtom.

Poznámka: všimnime si že vlastník účtu je zároveň disponentom tohto účtu. Keďže v ďalšom texte nás budú zaujímať operácie vykonávané na účte a nie jeho rušenie a určovanie disponentov, budeme hovoriť hlavne o disponentovi a nie o vlastníkovi.

Čiže v skratke: Alica chce využívať služby banky. Preto sa s bankou dohodnú na otvorení účtu. Banka od toho dňa tento účet vedie. Alica môže pridávať a odoberať disponentov tohto účtu a určovať práva pre každého z nich. Disponenti (a teda aj Alica) vykonávajú operácie nad daným účtom – každý podľa svojich práv.



2.1.2 Správanie entít a interakcie medzi entitami.

Správanie jednotlivých aktívnych entít systému a interakcie medzi nimi môžu byť komplikované. My sa pre potreby nášho návrhu obmedzíme len na niektoré zložky správania a interakcií.

V prvom rade, zaujímať nás bude situácia: zákazník má v banke otvorený účet, na ktorom má peniaze a s týmito peniazmi a samotným účtom môže on a ostatní disponenti nejakým spôsobom manipulovať – vykonávať operácie nad účtom. Ďalej, nebudeme sa zaoberať operáciami ako zakladanie účtu, vyber a vklad peňazí. Zaujímať nás budú operácie, ktoré budú súčasťou nášho návrhu, t.j. len tie, ktoré klient môže vykonávať elektronickou formou – a aj to len niektoré z nich.

Operácia (transakcia) je teda vykonanie nejakej akcie nad účtom. Operácie môžeme rozdeliť na dva typy – aktívne a pasívne. Aktívne sú také, ktoré modifikujú stav účtu (napr. stav prostriedkov). Pasívne sú také, ktoré len zisťujú momentálny stav účtu, resp. informácií, ktoré k účtu prislúchajú (stav prostriedkov, prehľad príkazov nad účtom, prehľad obratov nad účtom atd.).
Operácia má vždy typ (aká operácia), účet, nad ktorým sa vykonáva (zdrojový účet) a zadávateľa (disponent alebo banka). Ďalšie povinné a nepovinné parametre (atribúty) operácie závisia od typu operácie. Napríklad tuzemska úhrada musí mat uvedený aj

[27] Samozrejme, v reále nie sú všetky služby viazane na vytvorenie účtu, ale pre potreby nášho návrhu to môžeme predpokladať.

cieľový účet (kam sa majú peniaze previesť), množstvo peňazí a požadovaný dátum prevodu peňazí.

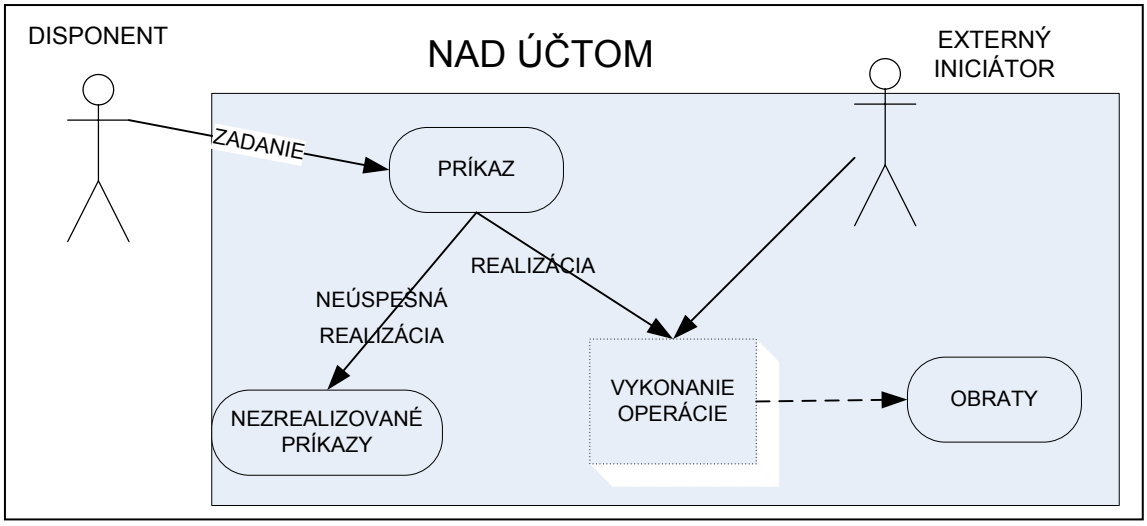
Operácie môže okrem disponentov iniciovať aj samotná banka a to spravidla na základe predošlej dohody s klientom (sťahovanie si poplatkov apod.). Okrem toho, v konečnom dôsledku všetky operácie vykonáva až banka. Klient ich len iniciuje – zadá príkaz na vykonanie operácie – a banka potom danú operáciu vykoná.

Príkaz je teda žiadosť klienta o vykonanie operácie nad účtom. Príkazu je jednoznačne priradená operácia. Príkaz v elektronickom bankovníctve slúži nielen ako žiadosť o vykonanie operácie, ale aj ako entita, pomocou ktorej budeme sledovať a usmerňovať vykonanie operácie bankou.

Životný cyklus príkazu
Príkaz môže počas spracovania bankou prechádzať rôznymi stavmi. Po úspešnom zadaní sa ocitne v počiatočnom stave. Potom prechádza rôznymi stavmi na základe toho, o aký príkaz ide a aké pripadné dodatočné kroky musia vykonať disponenti/banka na to, aby tento príkaz bol skutočne zrealizovaný. Pre vykonanie niektorých príkazov je napríklad najprv nutné, aby daný príkaz bol pred samotným zrealizovaním schválený *niekoľkými* disponentmi. Dôsledkom realizácie príkazu je vykonanie príslušnej operácie. Po zrealizovaní príkazu (alebo jeho prípadnom odmietnutí atd.) končí príkaz v jednom z koncových stavov. Na základe tohto môžeme hovoriť o životnom cykle príkazu.

Obrat na učte je záznam o operácií (transakcii) vykonanej nad týmto účtom. Môže to byť napríklad operácia vykonaná na základe príkazu zadaného disponentom. Na rozdiel od príkazu, obratu môže prislúchať aj operácia, ktorá bola iniciovaná nejakou externou entitou, čiže nie nutne len disponentom tohto účtu. Príklad takejto operácie je prevod peňazí z cudzieho účtu na náš účet. Obraty delíme na kreditné a debetné – podľa toho či zvyšujú množstvo peňazí na účte, alebo ho znižujú.

Stručne zhrnuté: disponent zadáva príkazy, v momente úspešnej realizácie príkazu sa vykoná operácia. Vykonaniu operácie prislúcha obrat – záznam o vykonanej operácií[28]. Nie každá operácia je vykonaná na základe príkazu disponenta (externí iniciátori) a nie každý príkaz konci vykonaním operácie.



[28] Pre niektore typy operacii, napr. premenovanie uctu, nemusi banka vytvorit obrat. Pre vsetky vyznamnejsie operacie sa to tak ale udeje. Pre niektore typy prikazov moze byt dokonca vygenerovanych niekolko obratov. Napr. v pripade prevodu si moze banka zauctovat poplatok, o ktoreho strhnuti tiez existuje zaznam – obrat.

2.1.3 Komunikácia

Až doteraz sme v našom modeli zanedbávali akúkoľvek potrebu komunikácie medzi entitami systému. V našom návrhu však bude mat práve komunikácia ústredné miesto. Na vykonávanie všetkých operácií, ktoré zadáva disponent, potrebujeme zabezpečiť komunikáciu medzi ním a bankou. Každá operácia je samozrejme definovaná pomocou nejakých informácií (typ operácie, parametre operácie) a tieto informácie musíme preniesť od disponenta k banke, ak chceme aby sa operácia mohla vykonať.

Pozrime sa, ako by vyzeral prenos informácií medzi disponentom a prepážkovým pracovníkom v banke v prípade, že by disponent chcel vykonať tuzemsku úhradu a skúsme vypozorovať nejaké charakteristiky komunikácie.

Disponent príde do banky, kde sa dožaduje byť obslúžený prepážkovým pracovníkom (ďalej PP). Disponent sa identifikuje pomocou občianskeho preukazu, identifikácia banky zastúpenej PP je automatická (sme v banke). Disponent povie akú operáciu chce vykonať (aký príkaz chce zadať). Ďalej pokračujeme, iba ak disponent má pravo ju vykonať. Ak je to tak, disponent písomne/ústne zadá vstupy príkazu a odovzdá ich PP. PP overí vstupné informácie a v prípade, že sú chybné, požiada disponenta aby vstupy opravil/doplnil. Toto sa opakuje až dokým vstupné informácie nemajú podobu, aká je požadovaná. Ak to operácia vyžaduje, disponent bude musieť k dátam na papieri definujúcim operáciu pridať svoj podpis, ktorý následne PP overí s podpisovým vzorom. Ak je všetko v poriadku, príkaz je úspešne zadaný a banka ho prijme na spracovanie. Disponent dostane potvrdenku o tomto fakte (s podpisom pracovníka banky a pečiatkou) a spokojne odchádza preč.

Po preskúmaní popísanej interakcie si môžeme všimnúť nasledovne charakteristiky komunikácie:

- komunikujúce strany: disponent + banka v zastúpení PP.
- kanál, cez ktorý sa prenášajú informácie: vzduch, papier.
- kódovanie informácie: slová z prirodzeného jazyka, reťazce mimo prirodzeného jazyka, čísla (symboly na papieri, zvuk).
- vzájomná autentifikácia oboch strán: občiansky preukaz a implicitná autentifikácia PP (banka v zastúpení).
- zabezpečenie integrity prenášaných informácií: v banke dostatočne ticho, takže prenos hlasu nie je rušený, papier putuje priamo od disponenta k PP a naopak.
- dôvernosť prenášaných informácií: zvukovú komunikáciu medzi PP a disponentom nikto nepočuje a písanú nikto nevidí.
- potvrdenie pre prípadnú reklamáciu: podpis klienta na papieri a potvrdenka banky s pečiatkou.
- je daná postupnosť výmeny informácií: najprv sa disponent identifikuje, potom vykoná operáciu (najprv zadá vstupy, tie sú overene, nakoniec dostane potvrdenku)
- čo sa robí pri chybných informáciách na vstupe
- a ďalšie

Takže toto sú niektoré charakteristík komunikácie a niektoré z podmienok, ktoré sú kladené na komunikáciu v prípade, že prebieha priamo v banke. Spolu tvoria komunikačný protokol, na ktorom sa potrebujú zúčastnené strany dohodnúť a dodržiavať ho, ak chcú, aby komunikácia fungovala a bola spoľahlivá a efektívna. Niektoré z týchto podmienok nie sú nutné pre samotne fungovanie komunikácie, ale zabezpečujú iné aspekty, ktoré chceme aby komunikácia mala – ide spravidla o bezpečnostné aspekty.

2.1.4 Elektronické bankovníctvo

Technologické výdobytky minulého storočia umožnili ľuďom začať komunikovať medzi sebou „na diaľku“ – najprv pomocou telefónu a neskôr prostredníctvom sofistikovanejších foriem, akými sú napríklad počítačové siete. Pri komunikácii na diaľku už nie je nutná „fyzická blízkosť“ komunikujúcich strán, ale na druhej strane pribudla nutnosť existencie komunikačného kanálu[29] medzi komunikujúcimi stranami.

Nové možnosti komunikácie sa premietli aj do sektoru bankovníctva a dnes už na vykonávanie viacerých operácií nie je nutné fyzicky prísť do banky – je možné vykonať tieto operácie na diaľku prostredníctvom elektronického komunikačného kanálu. Pri takejto forme komunikácie, resp. vykonávania operácií hovoríme o elektronickom bankovníctve.

Pozrime sa, ako sa situácia zmení, ak komunikácia bude prebiehať cez elektronicky kanál. Zjednodušený model komunikácie v elektronickom bankovníctve vidíme na nasledujúcom diagrame[30].

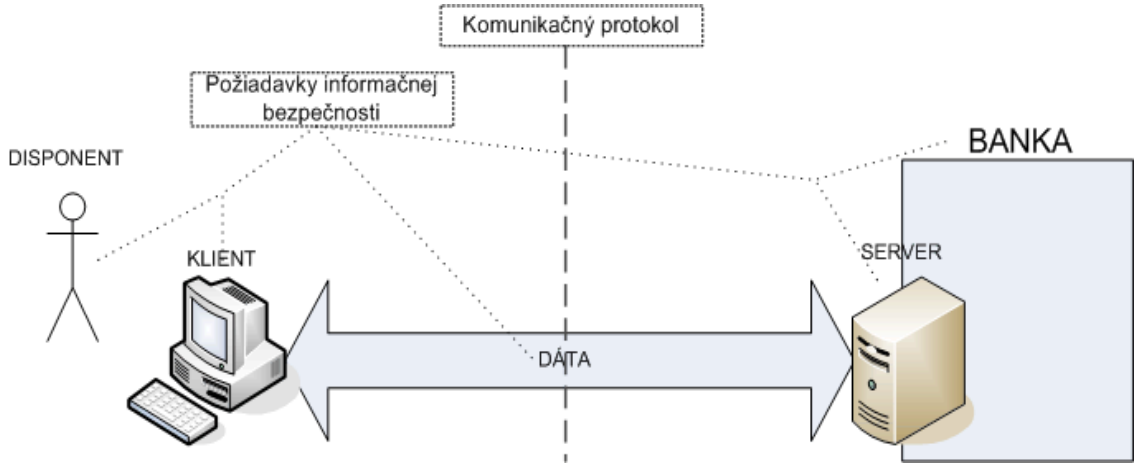


Diagram 19: Schema komunikácie v elektronickom bankovníctve

Okrem banky a disponenta tu máme komunikačný kanál (počítačová alebo iná sieť), **klienta,** čo je počítač (alebo iné zariadenie) na strane disponenta, ovládaný disponentom za účelom komunikácie s bankou a pribudol **server**, čo je počítač na strane banky, ovládaný bankou za účelom komunikácie s disponentmi a poskytovania služieb.

Komunikujúce strany teda nebudú komunikovať priamo, ale prostredníctvom počítačov (pripadne pomocou iných elektronických zariadení). Jednotlivé požiadavky na komunikáciu spomenuté hore pretrvajú ale ich implementácia bude iná. Okrem toho nám niekoľko požiadaviek ešte určite pribudne.

Skúsme si teraz stručne vymenovať požiadavky, ktoré budeme klásť na komunikáciu, resp. aké problémy budeme riešiť v scenári elektronického bankovníctva:

- Ako sú prenášané informácie (dáta) kódované.
- Aká je postupnosť výmeny takto kódovaných správ (musí byť definovaná) a ako sa pomocou výmeny správ realizujú jednotlivé operácie/služby.
- Ako dochádza k autentifikácii disponenta (klienta) a banky (serveru) na začiatku interakcie. V tomto scenári sa spôsob overenia identity zásadným spôsobom mení,

[29] Správidla *elektronickeho* komunikacneho kanalu

[30] Tento diagram nie je celkom všeobecný, klient a server nemusia byt nutne počítače (môže to byt napr. telefón a telefónny operátor/operátorka), ale nás bude zaujímať najmä scenár keď to tak bude, takže na úkor všeobecnosti predpokladáme že klient a server sú počítače.

keďže disponent a banka komunikujú len sprostredkovane. Banka potrebuje overiť, že človek, ktorý ovláda klienta, je skutočne disponent a disponent potrebuje overiť, že server je skutočne počítač ovládaný bankou.
- Ako je zabezpečená dôvernosť prenášaných informácií, t.j. že vymieňané informácie „vidia“ iba oprávnene komunikujúce strany.
- Ako je zabezpečená integrita prenášaných informácií, t.j. že dáta neboli počas prenosu (medzi korektne ovládanými počítačmi jednotlivými stranami) zmenené (poškodené apod.)
- Ako je umožnená autentifikácia zdroju dát, t.j. overenie na strane banky, že obdržané informácie vytvoril a poslal skutočne klient a naopak, že informácie obdržané klientom poslal skutočne server banky. Toto úzko súvisí s autentifikáciou strán a zabezpečením integrity prenášaných informácií.
- Čo sa robí v prípade, že príde k chybe v komunikácii, t.j. že niektoré z doteraz uvedených požiadaviek nebudú splnené. Ako sú takéto zlyhania detekované a ako sú ošetrené.
- Ako zabezpečíme, že komunikujúce strany nebudú môcť poprieť transakcie vykonané na základe vykomunikovaných informácií (v prípade že všetko išlo podľa protokolu). Predstavme si, že disponent zadá príkaz na úhradu prostriedkov a neskôr popiera, že on takúto úhradu zadal. Banka bude chcieť dokázať za dopredu určených podmienok (tretia strana, sud, zmluvne podmienky), že príkaz zadal skutočne on a je za jeho vykonanie právne zodpovedný.
- Ako banka zabezpečí dostupnosť služieb poskytovaných cez komunikačný kanál. Ide o zabezpečenia toho, že klienti budú môcť (prostredníctvom výmeny správ podľa protokolu) využívať služby v „očakávanej“ miere (24 hodín denne všetky operácie apod.)

K tomu, ako sú tieto a možno aj nejaké ďalšie požiadavky a problémy riešené, sa budeme podrobnejšie venovať v samotnom návrhu. Skúsme si teraz povedať niečo o formách elektronického bankovníctva.

2.2 Formy elektronického bankovníctva a motivácia pre použitie webservisov

V nasledujúcich odstavoch stručne popíšeme niektoré z významnejších foriem elektronického bankovníctva. Pre naše potreby spomenieme menej podstatné formy (telefónne bankovníctvo atd.) len stručne a zameriame sa na typy elektronického bankovníctva príbuznejšie tomu, ktorému sa budeme venovať v tejto diplomovej práci. Týmito to sú Internetbanking a Homebanking. Nakoniec si povieme niečo o samotnom elektronickom bankovníctve pomocou webservisov, jeho výhodách a použitiach.

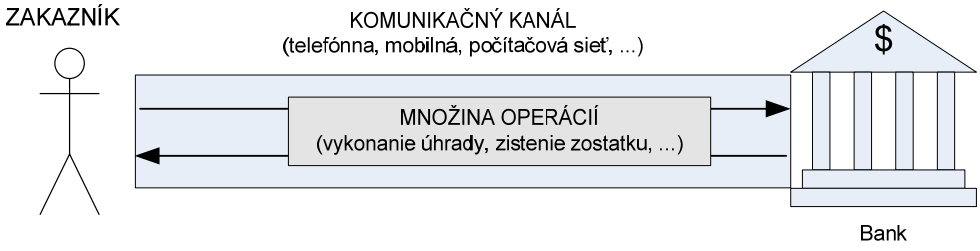


Diagram 20: Rôzne formy elektronického bankovníctva sú definovane rôznymi typmi komunikačných kanálov, príslušných protokolov a poskytovaných služieb.

Phonebanking (bankovníctvo pomocou telefónu a telefónnej siete)

Zákazník a banka komunikujú prostredníctvom telefónnej siete. Zákazník zavolá na telefónne číslo, ktoré mu poskytne banka, kde je obslúžený buď živým operátorom alebo telefónnym automatom. V druhom prípade prebieha komunikácia pomocou tonovej voľby. Množina poskytovaných služieb sa líši od banky k banke, poskytované sú pasívne a často aj aktívne operácie. Úroveň bezpečnosti je rôzna, v minulosti často stačilo, ak sa volajúci identifikoval menom a heslom. Možnosť zneužitia je v tomto prípade dosť vysoká – prípadnému útočníkovi stačí odpočuť meno a heslo. V dnešnej dobe je možné používať aj iné, bezpečnejšie formy autentifikácie (napr. generátor jednorazových hesiel), v prípade použitia ktorých je táto služba bezpečnejšia.

Mobilbanking (bankovníctvo pomocou telefónu a telefónnej siete)

V tomto prípade zákazník a banka komunikujú prostredníctvom mobilnej telefónnej siete, napr. GSM. Zákazník má v telefóne (na SIM karte) nainštalovanú aplikáciu, ktorá mu umožňuje vymieňať si šifrované správy s aplikáciou banky, ktorá funguje na vopred známom telefónnom čísle. Množina poskytovaných služieb opäť zahŕňa pasívne a spravidla aj aktívne operácie. Úroveň bezpečnosti: správy vymieňané medzi bankou a zákazníkom sú šifrované na základe symetrickej alebo asymetrickej kryptografie (kľúče sú uložené v aplikácii bezpečným spôsobom). Okrem toho, GSM sieť dodatočne šifruje všetku komunikáciu. Komunikačný kanál je tak relatívne bezpečný. Možnosť zneužitia existuje najmä v prípade, že útočník získa mobilný telefón zákazníka. V tom prípade mu stačí vedieť PIN/heslo do nainštalovanej aplikácie.

Homebanking

Homebanking je na Slovensku „služobne najstaršia“ forma elektronického bankovníctva a je určená najmä pre firmy a podnikateľov. V minulosti zákazník a banka komunikovali najmä prostredníctvom modemov cez telefónnu sieť alebo pomocou siete založenej na protokole X.25. Dnes preberá úlohu dominantného komunikačného kanálu už aj tu internet.

Zákazník má na počítači nainštalovanú aplikáciu dodanú bankou, ktorá komunikuje so serverom banky. Homebanking umožňuje relatívne efektívne, rýchlo a bezpečne vykonávať veľké množstvo operácií. Táto služba je určená najmä pre firmy a podnikateľov.

Výhody:

- Dostupné je veľké množstvo aktívnych aj pasívnych operácií.
- Efektívne vykonávanie veľkého množstva operácií (dávkové spracovanie pomocou importu súborov definujúcich operácie)
- Klientská aplikácia (klient) dokáže pracovať offline – zákazník si pripraví operácie, ktoré chce vykonať a potom ich naraz odošle.
- Aplikácia je tučný klient - môže si lokálne udržiavať informácie o stavoch účtov, vykonaných operáciách apod.
- Vysoká úroveň bezpečnosti založená na asymetrickej kryptografii a digitálnom podpise.

Nevýhody:

- Cena: niekoľko tisíc Sk zriaďovací poplatok; za samotné vykonávanie operácií sú už relatívne nízke poplatky
- Neflexibilný interface - jedna fixná klientská aplikácia dodaná bankou
- Nutnosť inštalácie aplikácie na strane klienta a jeho údržby.
- Kvôli „stavovosti“ klienta môže vzniknúť potreba synchronizovať stav klienta so serverom.

Internetbanking

Dnes najpoužívanejšia a najznámejšia forma elektronického bankovníctva. Zákazník a banka komunikujú prostredníctvom internetu. Zákazník na komunikáciu potrebuje

spravidla iba bežný webový prehliadač. Na druhej strane komunikačného kanálu je server banky, ktorý klientov obsluhuje. Táto služba je určená najmä pre fyzické osoby.
Výhody:

- Dostupné je veľké množstvo pasívnych a aktívnych operácií.
- Jednoduchosť klienta – štandardný webový prehliadač, absencia inštalácie.
- Cena (často „zadarmo“)
- Niekedy existuje možnosť dávkového spracovania podobným spôsobom ako v prípade homebankingu.

Nevýhody:

- Nutnosť online pripojenia počas celej práce s klientom
- Fixný interface (webový prehliadač)

Ostatné:

- Bezpečnosť je závislá od toho, aké bezpečnostné nástroje sú použité. Vždy sa používa minimálne SSL medzi serverom a prehliadačom. Rôzne bezpečnostné nástroje: statické heslo, GRID karta, generátory jednorazových autentifikačných kódov, asymetrická kryptografia, digitálny podpis.

Toto bol stručný prehľad pre nás zaujímavých foriem elektronického bankovníctva. Povedzme si teraz stručne niečo o tom, akú pridanú hodnotu môže poskytnúť použitie webservisov.

Elektronické bankovníctvo prostredníctvom webservisov

Základná výhoda webových služieb a rozdiel oproti všetkým horeuvedeným technológiám je, že web servis poskytuje vzdialený *programovateľný* interface pre prístup k službe elektronického bankovníctva. Voči tomuto interfejsu si môžeme naprogramovať ľubovoľného klienta, stačí ak bude konformný s definovaným rozhraním. Pre koncového používateľa to môžu byť rôzne formy grafických používateľských rozhraní, skripty vykonávajúce rôzne operácie za dopredu definovaných podmienok a pod.

Dnešný use case pre túto službu je ale hlavne pre firmy a podnikateľov, ktorí môžu integrovať služby elektronickému bankovníctva tesnejšie do svojich podnikových procesov. Môžu napríklad integrovať modul prístupu k elektronickému bankovníctvu do svojho ekonomického/účtovníckeho softvéru a vykonávať operácie elektronického bankovníctva priamo odtiaľ. Pre ilustráciu si uveďme jeden príklad použitia.

> Predpokladajme, že firma potrebuje vybavovať veľké množstvo faktúr denne. Tieto faktúry potrebuje zaevidovať do svojho ekonomického softvéru a zároveň ich uhradiť a pripadne poslať ďakovný email zákazníkovi, pretože chce budovať so zákazníkmi dobre vzťahy. Na vykonávanie týchto činností je vyhradený jeden pracovník. Čo potrebuje spraviť? Najprv zadá úhradu do účtovníckeho softvéru. Následne z účtovníckeho softvéru vygeneruje súbor definujúci úhradu pre elektronické bankovníctvo, prihlási sa cez niektorý kanál elektronického bankovníctva a vykoná úhradu importom tohto súboru. Nakoniec pošle mail zákazníkovi. V prípade, že by sa firma rozhodla integrovať prístup do elektronického bankovníctva do svojho ekonomického softvéru, možno by stačilo faktúru vložiť do systému, odtiaľ priamo ju uhradiť kliknutím na políčko uhradiť a následne by bol vygenerovaný email, ktorý by pripadne mohol pracovník poupraviť a odoslať ho priamo odtiaľ.

Aj z tohto zjednodušeného príkladu vidíme, že tento prístup šetrí čas, zvyšuje automatizáciu, (viac práce pre programy/počítač, menej pre pracovníka) a znižuje náchylnosť na chyby (tok dát na úhradu je aplikácia-aplikácia a nie aplikácia-užívateľ-aplikácia). Pre komplikovanejšie use cases by sme vedeli identifikovať aj významnejšie prínosy.

Výhody webservisu oproti Homebankingu a Internetbankingu:

- veľká flexibilita klienta
- automatizácia, zníženie náchylnosti na chyby, šetrenie času
- efektívnejšie vykonávanie veľké počtu operácií v krátkom čase aj v porovnaní s homebankingom
- množina služieb a operácií je flexibilná, definovaná protokolom

Nevýhody:

- klienta treba naprogramovať (banka môže poskytnúť nejaké knižnice)
- web servisy je stále relatívne mladá technológia, stále vo vývoji
- málo, resp. žiadne (verejné) porovnateľné implementácie v sektore bankovníctva
- bezpečnosť teoreticky vysoká (v závislosti od použitých nástrojov), ale ešte nie rozsiahlo empiricky overená (súvisí s predošlým bodom).
  Použitie modulu tretej (potenciálne nedôveryhodnej) strany a jeho integrácia do softvéru firmy, má implikácie pre bezpečnosť komunikácie medzi bankou a disponentom. Tieto rozoberieme v samotnom návrhu.

Webové služby sú dnes zaujímavé hlavne z dôvodu možnosti automatizácie vykonávania transakcií a integrácie do procesov firiem. Preto pravdepodobne budú skôr komplementárnym riešením a nie náhradou za už existujúce formy elektronického bankovníctva.

3 Postup riešenia & zdôvodnenie rozhodnutí

3.1.1 Úvod

Ako sme spomínali v úvode, cieľom tejto diplomovej práce bolo navrhnúť webservis pre elektronické bankovníctvo v slovenskom kontexte. Tento návrh mal byť založený na nejakom existujúcom otvorenom štandarde pre výmenu finančných informácií (správ). V tejto kapitole popisujeme, prečo sme vybrali štandard IFX, stručne ho popíšeme a zhrnieme úpravy a rozšírenia, ktoré sme realizovali pre dosiahnutie našich cieľov.

3.2 Výber štandardu pre finančné interakcie – IFX

Existujú dva štandardy, medzi ktorými sme sa rozhodovali. Sú to OFX (Open Financial Exchange[31]) a IFX (Interactive Financial Exchange[32]). Názvy nie sú podobne náhodné – IFX vzniklo z OFX a dnes tieto štandardy existujú paralelne. IFX ale vyzerá by perspektívnejší a z iných dôvodov vhodnejší pre naše použitie. V tejto sekcii si zdôvodníme prečo. Najprv si povieme niečo o OFX a potom ako IFX naňho nadväzuje a ako ho rozširuje.

OFX je štandard pre výmenu finančných dát a inštrukcií medzi konečnými zákazníkmi (consumers) a finančnými inštitúciami vo forme XML správ prostredníctvom HTTP protokolu, pričom všetky interakcie sú realizované ako request/response medzi klientským softvérom užívateľa a serverom banky. Výmenou dopredu definovaných správ sú realizované napr. aj bankové služby. Formát správ je v OFX dopredu a pevne definovaný – pomocou XML schémy. OFX vznikol v polovici deväťdesiatych rokov a dnes je tento štandard používaný najmä v USA a to najmä v retailovom bankovníctve[33].

IFX vznikol na začiatku tohto storočia ako modernejší a flexibilnejší pokračovateľ OFX. IFX výrazne ťaží z ideí OFX a vyberá si z neho všetky zaujímavé prvky a pridáva niekoľko nových. Popíšeme hlavne dôvody prečo sme vybrali IFX a nie OFX, nebudeme spomínať všetky rozdiely, iba tie relevantné pre naše rozhodnutie.

OFX požaduje, aby klienti boli *stateful*, teda udržiavali si stav. Náš cieľový typ klienta je však *stateless* klient. IFX podporuje aj stateful aj *stateless* klientov.

IFX explicitne špecifikuje, že je rozšíriteľný a špecifikuje ako sa dá rozšíriť, OFX mlčí. OFX je určený a používa sa hlavne v USA a množina vymieňaných správ a najmä ich „vnútro“ toto reflektuje. IFX sa snaží byť všeobecnejší pre praktiky a požiadavky vo viacerých krajinách (členovia IFX sú z USA, Európy, Ázie[34]). OFX explicitne nepodporuje použitie iných jazykov ako angličtiny. Na druhej strane IFX špecifikuje, ako sa dá internacionalizovať. IFX podporuje asynchrónne spracovanie požiadaviek (umožňuje klientom vyzdvihnúť si odpoveď neskôr).

IFX vyzerá byť perspektívnejší štandard – „deje sa okolo neho viac“. Pracuje sa na úpravách IFX štandardu pre webové služby (o tomto viac neskôr). Pre OFX už existuje, ale tento iba rozčlenil XML dokumenty do SOAP obálok. Množina služieb, ktorú IFX dokáže (resp. chce v budúcnosti) podporovať je širšia – OFX je určený iba na komunikáciu medzi zákazníkmi a finančnými inštitúciami a cieľom IFX je umožniť aj interakcie inštitúcia - inštitúcia. Toto síce nemusí byť v úzko z pohľadu nášho návrhu relevantné, ale v prípade, že sa IFX rozšíri v týchto oblastiach, môže byť znalosť jadra a jednej špecifickej časti výhodná pri snahe o adopciu iných častí.

---

[31] Viď http://www.ofx.net/. Pod OFX budeme chápať 1. štandard, 2. organizáciu.
[32] Viď http://www.ifxforum.org/. Podobne, je to štandard aj organizácia.
[33] Bankové služby určené pre konečného zákazníka. „Consumer Banking“.
[34] Viac ako 30 clenov z USA, Europy a Azie. Nielen financne institucie ale aj technologicke firmy ako napr. Microsoft, Sun Microsystems, Webmethods.

3.2.1 IFX v kocke

IFX je štandard, ktorý definuje komunikačný protokol pre výmenu finančných informácií (prevody, platby, ATM transakcie atd.). Nejde iba o pasívnu výmenu informácií. Vymieňané správy dávajú za vznik interakcii medzi finančnými inštitúciami navzájom a finančnými inštitúciami a zákazníkmi. Interakcia je založená na klient-server, request-response modeli. Klient posiela požiadavky (requests), server tieto požiadavky vybavuje a posiela späť odpovede. Táto výmena sa musí samozrejme realizovať pomocou nejakého protokolu, akým je napríklad HTTP[35]. Pozrime sa ako vyzerajú vymieňané IFX správy (pre protokol HTTP). Štruktúra požiadavky:

```
POST http://mycompany.com/IFXserverendpoint HTTP/1.0
... HTTP hlavičky ...

<?XML version="1.0" encoding="UTF-8"?>
<?ifx version="1.0.1" oldfileuid="d4bf1537-9159-4444-a306-00a0c91e6bf6"
newfileuid="3a8188de-b3f0-49f9-b052-00a0c91e6bf6"?>
<IFX>
   <SignonRq>...</SignonRq>
   <BankSvcRq>
      <XferAddRq>...</XferAddRq>
   </BankSvcRq>
   <SignoffRq>...</SignoffRq>
</IFX>
```

Požiadavky sa posielajú v rámci HTTP POST požiadaviek. Nasleduje deklarácia XML dokumentu a *IFX procesná inštrukcia*, ktorá okrem špecifikácie verzie umožňuje uviesť oldfileuid a newfileuid. K nim sa dostaneme neskôr. Potom už nasleduje samotný *IFX dokument* (element `IFX`), v prípade požiadavky ho budeme nazývať *IFX požiadavka*. Tento obsahuje požiadavku o prihlásenie (`SignonRq`), niekoľko *servisných* požiadaviek (v našom prípade len jedna: `BankSvcRq`), v rámci každej servisnej požiadavky niekoľko *individuálnych* požiadaviek (v našom prípade len jedna: XferAddRq) a nakoniec požiadavku o odhlásenie (`SignoffRq`). Server túto HTTP požiadavku prijme, spracuje a odošle odpoveď, ktorá má nasledovnú štruktúru:

```
HTTP/1.1 200 OK
... HTTP hlavičky ...

<?XML version="1.0" encoding="UTF-8"?>
<?ifx version="1.0.1" oldfileuid="d4bf1537-9159-4444-a306-00a0c91e6bf6"
newfileuid="3a8188de-b3f0-49f9-b052-00a0c91e6bf6"?>
<IFX>
   <Status>...</Status>
   <SignonRs>...</SignonRs>
   <BankSvcRs>
      <Status>...</Status>
      <XferAddRs>...</XferAddRs>
   </BankSvcRs>
   <SignoffRs>...</SignoffRs>
</IFX>
```

Štruktúra odpovede priamočiaro zodpovedá štruktúre odpovede. IFX procesná inštrukcia sa do odpovede iba zreplikuje. IFX dokument v tomto prípade obsahuje pre každú požiadavku z IFX požiadavky prislúchajúcu odpoveď v rámci analogickej štruktúry. Okrem toho pre každú požiadavku (individuálnu, servisnú, IFX, prihlásenie, odhlásenie) je v príslušnej odpovedi špecifikovaný výsledok spracovania (element `Status`). Správy, ktoré je možné vymieňať, sú rozdelene do tzv. služieb, čomu zodpovedá ich zadelenie do

[35] IFX štandard sa snaží byt všeobecný a umožniť výmenu správ pomocou rôznych transportných mechanizmov. Momentálne ale špecifikuje iba transport pomocou HTTP – Viď [IFXHTTP].

servisných požiadaviek a odpovedí. Signon a Signoff nie sú v žiadnej službe – vyskytujú sa priamo v IFX dokumente.
IFX definuje niekoľko základných dátových typov, z ktorých sú vytvorené všetky elementy, agregáty[36] a správy v IFX štandarde. Všetky dátové typy, elementy, agregáty a správy sú špecifikované pomocou XML schémy. IFX špecifikuje relatívne sebestačný komunikačný protokol (garantované doručenie, zotavenie z chyb, sessions, korelácia požiadaviek a odpovedi a detekcia duplikátov atd.). Takisto špecifikuje základné pravidlá pre spracovanie požiadaviek serverom. Nešpecifikuje však presný procesný model ako napr. SOAP, ale ponecháva flexibilitu pre rozširovateľov a implementátorov.

Spôsoby rozšírenia IFX štandardu

IFX umožňuje rozšíriť množinu elementov, agregátov a správ pre potreby danej krajiny alebo organizácie.
Rozšírenia sú realizované pomocou dodefinovania nových elementov, agregátov a správ s novými špecifickými názvami (custom names).
Špecifické meno sa skladá z prefixu, separátora a sufixu. Napríklad pre prefix „SVK“, separátor „_“ a sufix OrderAddRq získavame nove meno: „SVK_OrderAddRq“.

Rozšírenie špecifické pre krajinu umožňuje rozšíriť IFX o nove prvky, ktoré reflektujú finančné regulácie a prax v danej krajine. V tomto prípade sa ako prefix použije trojznakový prefix krajiny podľa ISO-3166; sufix je zvolený ľubovoľne ale konformne s konvenciami IFX[37]. Separátor je závislý od implementácie.
Proprietárne rozšírenia umožňujú rozšíriť IFX napr. pre potreby konkrétnej organizácie. V tomto prípade sa ako prefix nepoužíva kód krajiny, ale plne kvalifikované obrátené meno internet domény danej organizácie. Napr. pre organizáciu s doménou mycompany.sk, by prefix bol sk.mycompany. Druhá možnosť je zaregistrovať si proprietárny prefix (dlhší ako 4 znaky) s IFX. Tretia možnosť je použiť XML menný priestor, do ktorého vložíme všetky nove elementy a agregáty.

My sme v našom návrhu použili oba spôsoby rozšírenia. Niektoré elementy boli definované ako špecifické pre proprietárne rozšírenie a boli zadefinovane do špecifického XML menného priestoru a niektoré elementy boli definované ako slovenské rozšírenie IFX. Tieto dostali prefix SVK a boli definované priamo v mennom priestore spoločnom pre povodne IFX agregáty, keďže ide o (pokus o) štandardné rozšírenie pre slovenský platobný styk, resp. pre slovenské retailové bankovníctvo.

3.2.2 Ako rozšíriť/upraviť IFX pre naše potreby?

Mali sme pred sebou nasledovné úlohy.

1. Zabezpečiť, aby obsah vymieňaných správ vyhovoval potrebám slovenského platobného styku (používaných v retailovom bankovníctve) a procesom existujúcim v jednej banke (so snahou o všeobecnosť pre všetky banky). Za týmto cieľom sme potrebovali formát a obsah buď upraviť, rozšíriť, alebo vytvoriť nove správy. Ako uvidíme, realizovali sme všetky tri možnosti.
2. Upraviť IFX pre použitie v rámci webservisov. Upraviť štruktúru správ, namapovať IFX dokument do SOAP obálky a prípadne pridať nové prvky špecifické pre webservisy – takýmto bol v našom prípade iba WS Security hlavičkový blok v SOAP Header.

Poznámka: Tieto ciele boli komplementárne a niekedy sa prelínali, napr. použitie WS Security v rámci SOAP Header má dôsledky pre proces spracovania individuálnych

[36] IFX rozlišuje „elementy“ a „agregáty“. Element je XML element obsahujúci iba dáta. Agregát je XML element obsahujúci iba elementy a agregáty – teda určíte nie dáta, ani mix dať a elementov.
[37] Mena správ sú vytvorene ako zretazenie triedy, objektu, vlastnosti, metody a smeru, napr. SVK_OrderAddRq. Viac v [IFXBMS].

požiadaviek, ktoré realizujú operácie požadované klientom v rámci IFX Request a teda majú aj implikácie pre spôsob, akým sú tieto operácie realizované.

3.3 Rozšírenie IFX pre slovenský platobný styk & procesy

Keďže náš návrh bol určený pre slovenský platobný styk a žiadny z existujúcich štandardov (ani IFX) nepodporoval (pomocou existujúcich správ) realizáciu požadovaných operácií a procesov slovenského retailového bankovníctva, potrebovali sme IFX rozšíriť/upraviť pre dosiahnutie týchto cieľov.
Najprv sme sa snažili iba o minimálne modifikácie/rozšírenie existujúcich správ so snahou o zachovanie maximálnej konformnosti. Takéto rozšírenia sa však čoskoro začali javiť ako nepostačujúce a museli sme navrhnúť značné množstvo nových správ[38]. Jednoduchý model toho, čo sme potrebovali realizovať, sme už načrtli v kapitole o elektronickom bankovníctve.

Príkazy a podporne správy

V prvom rade, všetky aktívne operácie sú realizované prostredníctvom „príkazov" klienta, pričom typ príkazu definuje typ operácie, ktorá sa má v konečnom dôsledku vykonať. IFX však niečo ako príkaz nepozná a definuje iba agregáty priamo pre niektoré typy operácií. My sme potrebovali realizovať minimálne tri typy príkazov: tuzemský prevod, zahraničný prevod a hromadný prevod. IFX by nám podporu pre tieto na prvý pohľad mohla poskytnúť v správach a agregátoch, ktoré špecifikuje pre prevod: Xfer množina správ a agregátov: XferInfo, XferAdd, XferCan atd. IFX pritom predpokladá, že pomocou týchto sa budú realizovať aj zahraničné prevody. V slovenskom platobnom styku sa ale náležitosti pre tuzemský a zahraničný platobný styk výrazne líšia a tak by pri použití iba jedného druhu štruktúry v podobe Xfer množiny agregátov vznikla veľmi „nepekná" štruktúra. Okrem toho potrebujeme realizovať aj hromadne prevody, pre ktoré je štruktúra Xfer triedy agregátov úplne nepostačujúca. Z týchto dôvodov sme vytvorili novú množinu správ všeobecne pre „príkazy", pričom typ príkazu je špecifikovaný vnútri vseobecného agregátu pre príkaz a každý jednotlivý typ príkazu definuje vlastný agregát, ktorý sa potom vyskytuje vo všeobecnom agregáte pre príkaz. Keďže v budúcnosti očakávame, že množina príkazov by sa mohla rozšíriť, bola toto rozumnejšia voľba, ako nejakým komplikovaným spôsobom používať Xfer agregáty.
Xfer správy ale umožňujú realizovať niektoré operácie, ktoré potrebujeme aj pre každý príkaz všeobecne – jeho zadanie disponentom (XferAdd), jeho zmazanie (XferCan) a dotazovanie sa na príkazy (XferInq). Okrem toho, pre každý prevod zadaný na strane serveru existuje XferRec agregát, ktorý umožňuje klientovi sledovať jeho stav, ako ho banka spracováva apod. Tieto štruktúry sa ideovo dajú použiť aj pre príkazy. My teda analogicky definujeme Order, OrderAdd, OrderCan, OrderInq a ďalšie pre podporu príkazov a ich životného cyklu.

Obraty

Ďalším elementárnym prvkom požadovaným v našom návrhu je obrat. Keďže obraty sú pasívne, jediné, čo s nimi potrebujeme robiť, je získavať o nich informácie, resp. dotazovať

[38] Snahu o konformnost s IFX sme vsak zachovali, ale iba v "rozumnych" medziach. Otvorenou otazkou zostava, čo by sme získali, resp. stratili keby sme si vytvorili úplné nove správy a agregáty a existujúce náprotivky z IFX by sme použili iba ak by sa nám 100% hodili. Jeden dôsledok je strata štandardnosti obsahu správ, ale tato už aj tak nie je vysoká, na druhej strane agregáty IFX, ktoré aj momentálne v našich správach používame často štruktúru správ komplikujú. Pre prípadnou implementáciou treba zvážiť, či predsa len nezvoliť cestu minimálne konformnosti.
Tu však treba podotknúť, že čo sa týka samotnej štruktúry správ a základných prvkov, ktoré každá správa obsahuje, sme 100% konformní s IFX a teda komunikačný protokol ani procesný model sa z dôvodu návrhu nových správ nemení. Viac o štruktúre správ je povedané v kapitole venovanej návrhu.

sa na ne. IFX tieto naše požiadavky podporuje pomocou DepAcctTrnInq (dotazovanie sa na obraty) a DepAcctTrnRec (záznam o obrate na serveri). Tieto agregáty (Deposit Account Transaction agregáty) sú ale určené len pre potreby účty depozitného typu, a nie pre účty pôžičkového typu (Loan Account). Toto je ale podstatný zádrheľ, ktorý by nám neumožnil dotazovať sa na obraty na pôžičkových účtoch – čo je pre nás jedna z požiadaviek. Preto sme museli príslušné agregáty premenovať a mierne upraviť ich štruktúru pre naše potreby.

Autentifikácia

Spomeňme ešte, aspoň v skratke, ako sme upravili Signon správy. Signon správa v IFX umožňuje prihlásenie a autentifikáciu užívateľa veľkým množstvom spôsobov – pomocou ID užívateľa a hesla a pomocou rôznych iných metód[39]. Tieto sú však nevyhovujúce pre naše potreby. V slovenských podmienkach formy bankovníctva, ekvivalentne tomu nášmu, využívajú dva spôsoby prihlásenia – metódy založené na asymetrickej kryptografii (pravdepodobne challenge-response) a generátory jednorazových autentifikačných kódov[40]. Autentifikácia pomocou statického hesla je považovaná za nepostačujúcu. Metódy autentifikácie sa dajú ale v rámci Signon požiadavky jednoducho dodefinovať. Pre každý typ autentifikačného mechanizmu totižto existuje samostatný agregát v rámci SignonRq – užívateľ uvedie vždy jeden z nich a tak špecifikuje metódu prihlásenie a aj dáta pre túto formu prihlásenia. My sme teda dodefinovali agregát pre generátor autentifikačných kódov (SignonEOK) a pre challenge-response autentifikáciu pomocou asymetrickej kryptografie (SignonChalRes). Tieto sú popísané v kapitole Návrh a podrobná špecifikácia ich obsahu je zdokumentovaná v prílohe tejto práce (spolu so všetkými ostatnými správami a agregátmi používanými v našom návrhu).

Ďalšie úpravy...

Takýmto spôsobom by sme mohli pokračovať a aspoň ideovo popísať zmeny, ktoré sme vykonali pre všetky rôzne typy správ a agregátov. Pochybujeme však, že ďalšie informácie tohto typu sú pre čitateľa zaujímavé. Presné úpravy, ktoré sme vykonali sme však zaevidovali pre pripadné ďalšie rozšírenia množiny správ a agregátov alebo pre ich neskoršiu revíziu. V tejto práci ich podrobne uvádzať nebudeme.

Nakoniec však ešte jedna dôležitá poznámka o vykonaných úpravách. V prípade všetkých pôvodných agregátov definovaných v IFX sme zachovali ich *povinný* obsah a vynechali sme iba *nepovinný* obsah, pripadne sme do nich pridali naše nové prvky – špecifické či už pre slovenský platobný styk všeobecne (prefix SVK) alebo pre proprietárne rozšírenie (elementy a agregáty definované do vlastného XML namespace. Týmto sme dosiahli to, že naše úpravy a rozšírenia sú konformne s pravidlámi definovanými v IFX špecifikácii [IFXBMS].
Všetky elementy, agregáty a správy používané v našom návrhu sú definované v XML Schéme a sú priložené k tejto diplomovej práci. Okrem toho existuje kompletná dokumentácia pre všetky správy a agregáty.

Z čoho sme čerpali pri návrhu správ a procesov

Náležitosti tuzemského platobného styku sú definované najmä v opatreniach Národnej Banky Slovenska[41]. Zahraničný platobný styk slovenské banky spravidla realizujú prostredníctvom SWIFT[42]-u. Pre nás bolo ale podstatné vedieť iba položky pre tuzemské a zahraničný prevod a tieto sa dajú získať aj z iných, lepšie dostupných a lepšie čitateľných

---

[39] embedded certificate, transport certificate, Integrated Circuit Card, magnetic stripe card, PIN pad.
[40] Príklad takéhoto zariadenia je zariadenie EOK používané pre proprietárnu časť nášho návrhu – je popísané v príslušnej kapitole.
[41] http://www.nbs.sk/
[42] http://www.swift.com/

zdrojov. Týmito boli Databanking Slovenskej Sporiteľne (viď [SLSPDB]) a jednotlivé elektronické bankovníctva slovenských bank[43]. Do spoločného jadra nášho návrhu správ, sme chceli zahrnúť všetky položky pre tuzemský/zahraničný príkaz, ktoré sú aspoň trochu „štandardizované“, t.j. používali sa vo viacerých bankách.
Pochopenie pre procesy a interakcie sme získali ich analýzou v jednej konkrétnej banke, zo [SLSPDB] a z toho, čo sme vypozorovali z elektronických bankovníctiev iných bank .
Požiadavky kladené na bezpečnosť a nástroje na jej dosiahnutie sme skúmali opäť v elektronických bankovníctvach slovenských bank.
Návrhu správ a operácií v našom návrhu sme sa na základe vypozorovaného snažili urobiť dostatočne všeobecný tak, aby mohol byť (po prípadných drobných úpravách) použitý aj v iných slovenských bankách (okrem jednej konkrétnej, pre ktorú sa tento návrh primárne robí).

3.4 Úprava IFX pre Webservisy

V skratke, upravenie IFX štandardu pre webservisy znamenalo dotvoriť procesný model IFX tak, aby zapadol do procesného modelu SOAPu, upraviť štruktúru IFX dokumentu pre použitie v rámci SOAP obálky a vytvoriť WSDL dokument pre výmenu IFX správ. Skúsme sa stručne zamyslieť nad týmito úpravami.

WSDL

Keďže klient a server si vymieňajú IFX dokumenty (resp. IFX požiadavky a IFX odpovede), celkom postačí ak celý web servis bude realizovaný jednou request/response operáciou.
Ďalej, pre IFX je prirodzene používať document/literal web servisy. Voľba štýlu *document* (a nie RPC) je zjavná z toho hľadiska, že sa vymieňajú dopredu definované dokumenty a teda nie sú volané vzdialene procedúry/operácie v doslovnom zmysle. Voľba kódovania *literal* (a nie encoded) je tiež zrejmá, keďže nás postup riešenia nie je “existujúca aplikácia & aplikačne dáta → XML správa” ale máme dopredu definované XML správy, ich presný formát a dátové typy. Okrem toho, ako sme už spomínali v kapitole o webservisoch, štýl encoded bol zakázaný v odporúčaní WS-I Basic [WSIBP].

Procesný model

IFX nešpecifikuje “procesný model” explicitným spôsobom tak, ako to robí napríklad SOAP. Existujú len fragmenty, ktoré sú roztrúsené po [IFXBMS] špecifikácii. Tieto spolu nešpecifikujú kompletný model spracovania IFX požiadavky. Predpokladáme preto, že IFX ponecháva voľné ruky implementáciám (serverom) na spracovanie požiadaviek podľa “im vyhovujúcich rozumných pravidiel”.
My v našom návrhu budeme využívať SOAP v naviazaní na HTTP. Týmto bude definované, ako sa pomocou HTTP protokolu požiadavky posielajú, ako sa po prijatí spracovávajú (v rámci štruktúry SOAP Obálky) a ako sú vygenerované odpovede vracané späť klientovi. Takisto je presne špecifikovaná štruktúra chýb (v SOAP Fault) a spôsob ich signalizovania klientovi (HTTP odpoveď s príslušným HTTP status code). IFX procesný model, ktorý my potrebujeme dodefinovať, sa teda bude týkať najmä toho, ako budeme IFX požiadavku a jej obsah (SignonRq, servisné požiadavky, individuálne požiadavky v rámci servisných požiadaviek, SignoffRq) spracovávať v rámci SOAP processing modelu. Takisto definujeme, ako sa rôzne chyby, ktoré môžu vzniknúť v rámci spracovania IFX požiadavky, prejavia do štruktúry IFX odpovede a SOAP obálky (v ktorých prípadoch chyby budeme vracať v rámci SOAP Fault a kedy nie) .

[43] Skúmali sme elektronické bankovníctva nasledovných bank: Slovenska sporiteľňa, Ľudová banka, Tatrabanka, OTP Banka, Dexia, CSOB, Unibanka, VUB, Istrobanka.

Úprava IFX dokumentu pre SOAP obálku

Ďalej sa musíme zamyslieť ako namapovať IFX dokument do SOAP obálky. V prvom rade, SOAP obálka nesmie obsahovať žiadne *XML processing instructions*, čiže IFX procesnú inštrukciu[44] nemôžeme priamo použiť. Prirodzeným spôsobom z nej ale môžeme spraviť agregát, ktorý nazveme IFXHeader a pridáme mu troch potomkov: version, NewFileUID a OldFileUID. Jeho funkcia zostane nezmenená. IFXHeader je určený pre konečného príjemcu takže ho môžeme umiestniť do SOAP Body ako prvý element.
Samotný IFX dokument umiestnime do SOAP Body hneď za IFXHeader. IFX používa názov IFX pre názov elementu IFX požiadavky aj odpovede. My túto konvenciu pre uľahčenie návrhu XML schémy a následnú pracú s inštanciami tejto schémy zmeníme. Element požiadavky bude mat názov IFXRequest a element odpovede IFXResponse. Samotný obsah zostane nezmenený.
V SOAP obálke bude možné nepovinne uviesť WS Security hlavičkový blok a v ňom Signature element pre potreby digitálneho podpísania *celého* SOAP Body a teda celej IFX požiadavky. Okrem toho v našom návrhu umožníme podpisovanie jednotlivých požiadaviek pomocou zaobaleného typu podpisu v rámci individuálnych požiadaviek – podpisovať sa budú individuálne požiadavky.

Budúcnosť pre IFX a Webservisy

My sme pre potreby nášho návrhu realizovali len minimálnu množinu zmien nutnú na použitie IFX v rámci webservisov – tak, aby sme mohli realizovať naše požiadavky.
Je však možné urobiť aj zásadnejšie zmeny, ktoré by zosúladili IFX s "filozofiou" webservisov, resp. by využívali všetky technologické výhody, ktoré web servisy dnes a v skorej budúcnosti poskytnú. Hovoríme najmä o štandardných rozšíreniach WS-*, akými sú napr. WS-Addressing, WS-Trust, WS-SecureConversation, WS-ReliableMessaging, WS-Coordination, WS-AtomicTransaction, and WS-Policy.
O úplnejšie úpravy sa už značnú dobu snaží samotne IFX[45]. Záujemcovia sa o týchto snahách môžu dozvedieť napríklad v [IFXWS] a [IFXWS2], ale dostupných informácií je len minimum.
V svetle týchto skutočnosti sa ukázala naša cesta neveľkých uprav ako celkom rozumná. Je totižto zbytočné snažiť sa o úpravy paralelne s niekým, koho zmeny budú aj tak – a aj pre nás – referenčné.

---

[44] Vo formáte `<?ifx version="1.0.1" oldfileuid="00000000-0000-0000-0000-000000000000" newfileuid="00000000-0000-0000-0000-000000000000" ?>`

[45] Pracuje na ňom IFX Web Services Work Group – už vyše dva roky. Výstupom majú byt „modularizované schémy, wsdl a referenčné implementácie pre Javu a .NET". Ich zverejnenie bolo už niekoľkokrát ohlásené, ale stále sa odkladá (už vyše roka), čo naznačuje, že kompletnejšie úpravy asi nie sú triviálne. V každom prípade bude zaujímavé vidieť – aj pre budúcnosť nášho návrhu – čo IFX vytvorí. Posledne informácie, ktoré sa nám podarilo získať od IFX WS Work Group, boli, že tieto výstupy budú k dispozícii začiatkom mája 2006 (po výročnom meetingu IFX).

4 Návrh

4.1 Úvod

V tejto podkapitole načrtneme cieľovú architektúru pre náš návrh, pokúsime sa o zhrnutie základných čŕt návrhu a uvedieme príklad interakcie medzi klientom a serverom, ktorý nám tieto základné črty pomôže ozrejmiť.

4.1.1 Architektúra

Cieľom nášho návrhu je umožniť banke poskytovať (bankové) služby svojim zákazníkom. Zákazník a banka nekomunikujú priamo, ale prostredníctvom klienta a serveru. Bankové služby sú realizované prostredníctvom operácií, ktoré môže zákazník prostredníctvom klienta vzdialene vykonávať. Každá operácia je realizovaná ako dvojica request/response – klient pošle požiadavku (čo chce vykonať) a server pošle späť výsledok v odpovedi. Obsah správ, operácie a služby sú v našom návrhu definované pomocou XML Schémy a jazyka WSDL. Správy sa vymieňajú pomocou protokolu HTTPS v SOAP obálkach. Správy alebo ich časti môžu byť digitálne podpísané. Cieľová architektúra systému je nasledovná.

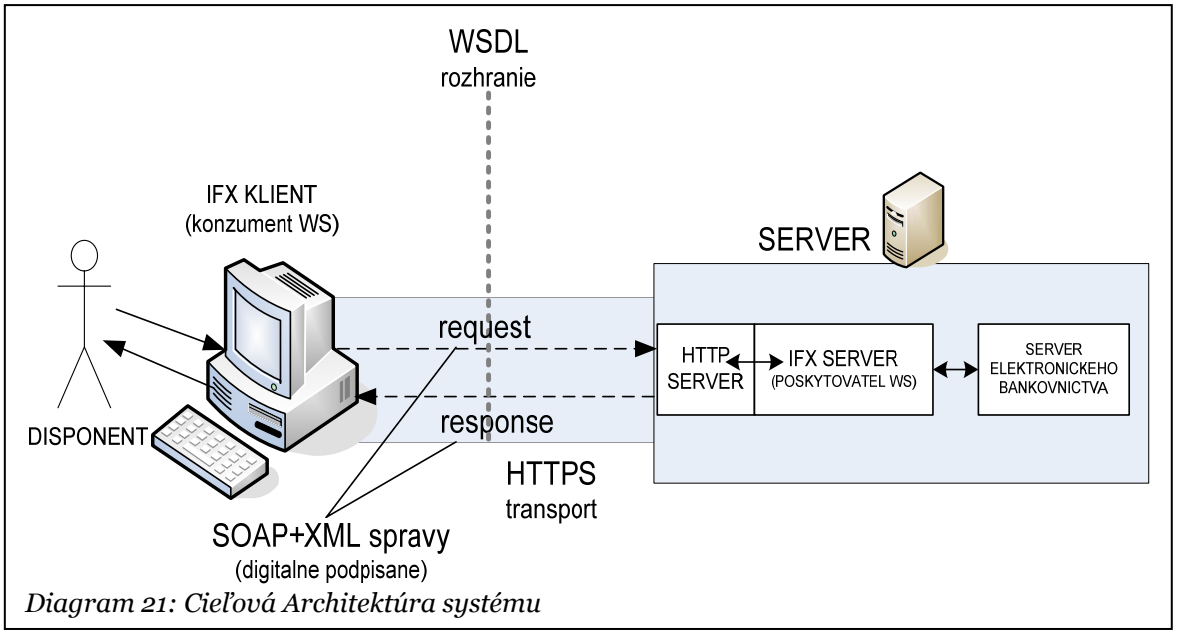


Diagram 21: Cieľová Architektúra systému

Skúsme si presnejšie definovať aktérov načrtnutých na tomto diagrame.
IFX klient (alebo jednoducho **klient**) je program ktorý dokáže komunikovať s IFX serverom podľa protokolu definovaného v tomto návrhu, t.j. posielať požiadavky a prijímať odpovede v správnom formáte a správnym spôsobom.
IFX server je program, ktorý dokáže komunikovať s IFX klientami podľa protokolu definovaného v tomto návrhu.
Poznámka: Z hľadiska technológie webových služieb je IFX klient konzumentom webových služieb a IFX server poskytovateľom webových služieb.
Server – všeobecne je to nositeľ funkcionality, ktorú využíva klient. Buď IFX server alebo HTTP server alebo server elektronického bankovníctva. Jednotlivé funkcionality týchto serverov často nie je potrebné rozlišovať, alebo v danom kontexte splývajú do jedného celku.

Užívateľ – je osoba, ktorá používa a ovláda klienta s cieľom využívania služieb, ktoré poskytuje banka prostredníctvom serveru. Pravdepodobne disponent[46].
Disponent – osoba, ktorá vykonáva operácie nad účtom v banke – v tomto kontexte prostredníctvom IFX klienta, ktorý si vymieňa správy so serverom.
Banka – finančná inštitúcia poskytujúca bankové služby svojim zákazníkom – disponentom.

4.1.2 Úvodný príklad

Skúsme si teraz uviesť príklad na ktorom si ozrejmíme základné črty nášho návrhu. Ako sme spomínali, banka poskytuje prostredníctvom serveru bankové služby svojim zákazníkom – disponentom, pričom služby predstavujú množinu operácií. Jednou zo základných poskytovaných operácií je úhrada prostriedkov z účtu disponenta na iný účet. Pozrime sa, ako by mohla vyzerať výmena požiadavka/odpoveď pre realizáciu tejto operácie. Najprv zjednodušená požiadavka:

```
POST http://mycompany.com/IFXserverendpoint HTTP/1.0
... HTTP hlavičky ...

<?XML version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="...">
   <soapenv:Body>

      <IFXHeader xmlns="...">
         <Version>1.7</Version>
         <NewFileUID>3a8188de-b3f0-49f9-b052-00a0c91e6bf6</NewFileUID>
      </IFXHeader>

      <IFXRequest xmlns="...">
         <SignonRq>
            <SessKey>as3d5f7oi45uq493ai456sopu305qru90qddk0dk</SessKey>
         </SignonRq>
         <BankSvcRq>
            <RqUID>3dd1ce96-292e-4cbf-9bde-00a0c91e6bf6</RqUID>
            <SVK_OrderAddRq Id="bd54493a-71c3-4ca1-8ee8-00a0c91e6bf6">
               <RqUID>bd54493a-71c3-4ca1-8ee8-00a0c91e6bf6</RqUID>
               <SVK_DomXferInfo>
                  <DepAcctIdFrom>SK8409000000000179661406</DepAcctIdFrom>
                  <DepAcctIdTo>SK4231000000000478682567</DepAcctIdTo>
                  <CurAmt>1000SK</CurAmt>
               </SVK_DomXferInfo>
               <ClientDt>2006-04-05T21:32:14.367352+01:00</ClientDt>
               <Signature xmlns="...">
                  <SignedInfo>
                     ...
                     <Reference URI="#bd54493a-71c3-4ca1-8ee8-00a0c91e6bf6">
                        ...
                     </Reference>
                  </SignedInfo>
                  <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
               </Signature>
            </SVK_OrderAddRq>
         </BankSvcRq>
      </IFXRequest>

   </soapenv:Body>
</soapenv:Envelope>
```

Prvé čo si všimneme je, že požiadavka sa posiela ako HTTP POST požiadavka, pričom je identifikované URL koncového bodu poskytovateľa webovej služby. Ďalšie hlavičky pre potreby tohto úvodného príkladu ignorujeme.
Telo HTTP požiadavky tvorí SOAP obálka, ktorá v tomto prípade obsahuje iba SOAP Body. V tomto sú uvedené dva elementy: IFXHeader a IFXRequest.
IFXHeader pomocou NewFileUID jednoznačne identifikuje túto IFX požiadavku. Okrem toho je v IFXHeader uvedena verzia IFX použitá pre túto požiadavku. Potom je tu už samotný IFXRequest, ktorý definuje, čo klient od serveru žiada. IFXRequest obsahuje SignonRq, čo je požiadavka o autentifikáciu klienta a jeho prihlásenie do systému. Správna

[46] Môžu existovať aj iní užívatelia systému, aj keď týchto v našom návrhu explicitne nespomíname.

autentifikácia je predpokladom spracovania zvyšku IFX požiadavky. V tomto prípade sa klient identifikuje pomocou tzv. SessionKey, ktorý bol vygenerovaný na základe niektorej z predošlých požiadaviek klienta, kde sa autentifikoval „plným“ spôsobom, napr. pomocou challenge-response autentifikácie.

IFXRequest môže okrem SignonRq obsahovať niekoľko *servisných* požiadaviek. Každá servisná požiadavka je identifikovaná univerzálne unikátnym ID[47] (element RqUID) a agreguje v sebe niekoľko individuálnych požiadaviek pre daný servis (službu). V našom prípade je uvedená jediná servisná požiadavka – BankSvcRq, ktorá obsahuje jedinú individuálnu požiadavku SVK_OrderAddRq. Táto definuje žiadosť klienta o zadanie nového príkazu, ktorým je v tomto prípade tuzemský prevod (element SVK_DomXferInfo). SVK_OrderAddRq okrem definície tuzemského prevodu obsahuje RqUID – každá individuálna požiadavka musí byť tiež unikátne identifikovaná. Ďalej je tu uvedený ešte datetime klienta (element ClientDt), ktorý slúži pre posledný element obsiahnutý v SVK_OrderAddRq, ktorým je Signature. Signature je v tomto prípade digitálny podpis zaobaleného typu a podpisuje individuálnu požiadavku, ktorá ho obsahuje – SVK_OrderAddRq. Tento podpis je potvrdenkou pre server o tejto požiadavke klienta a slúži pre potreby riešenia možných sporov medzi disponentom a bankou. Všimnime si ešte, že na individuálne požiadavky sa môžeme z elementu Signature/Reference odkazovať pomocou RqUID danej požiadavky – použitie RqUID pre hodnoty Id atribútov prirodzeným spôsobom zaručuje unikátnosť hodnoty tohto atribútu v IFX požiadavke.

Skúsme si teraz niektoré z pojmov spomenutých v príklade definovať presnejšie:

Požiadavka (Request) – Požiadavku môžeme chápať niekoľkými spôsobmi:

a) Individuálna (jednoduchá, jednotlivá) požiadavka je žiadosť o vykonanie nejakej operácie poskytovanej serverom. Napr. SVK_OrderAddRq je žiadosť o zadanie príkazu.
b) Servisná požiadavka (Service Request) je množina individuálnych požiadaviek pre nejakú službu. Napr. BankSvcRq (Banking Service Request) zaobaľuje niekoľko individuálnych požiadaviek pre službu bankovníctva.
c) IFX požiadavka je agregácia niekoľkých servisných požiadaviek.
d) HTTP požiadavka posielaná klientom. Obsahuje IFX požiadavku.

Z kontextu sa dá spravidla ľahko zistiť, o ktorý typ požiadavky ide.
Odpoveď (Response) definujeme analogicky k požiadavke: odpoveď typu X je odpoveďou na požiadavku typu X (Individuálna, servisná, IFX, HTTP odpoveď).
Správa (Message) – je buď požiadavka (request message) alebo odpoveď (response message).
Operácia – dvojica správ (požiadavka-odpoveď) realizujúca nejakú ucelenú jednotku funkcionality. Napr. dvojica správ SVK_OrderAddRq (požiadavka), SVK_OrderAddRs (odpoveď) realizuje zadanie príkazu.
Služba (Servis) – množina súvisiacich operácií, ktoré umožňuje poskytovateľ služieb (banka) vykonávať svojim zákazníkom.

Teraz sa môžeme vrátiť späť k nášmu príkladu. Predpokladajme, že klient úspešne odoslal danú požiadavku a server ju prijal. Ako postupuje server ďalej? Najprv skontroluje, či je požiadavka správna po syntaktickej stránke a či sedia dátové typy (je vykonaná validácia podľa XML Schémy). Následne server spracuje požiadavku podľa procesného modelu. Na základe výsledku spracovania vygeneruje odpoveď a odošle ju späť klientovi. Pre potreby nášho príkladu budeme predpokladať, že požiadavka bola vybavená v poriadku a server posiela klientovi späť odpoveď, kde ho o tomto fakte informuje. Tá môže vyzerať napríklad nasledovne:

[47] Ktoré generuje a zadáva klient.

```
HTTP/1.1 200 OK
... HTTP hlavičky ...

<?XML version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="...">
   <soapenv:Body>
      <IFXHeader xmlns="...">
         <!-- obsah rovnaky ako v poziadavke -->
      </IFXHeader>
      <IFXResponse xmlns="...">
         <SignonRs>
            <Status>OK</Status>
            <Language>SK</Language>
         </SignonRs>
         <BankSvcRs>
            <RqUID>3dd1ce96-292e-4cbf-9bde-00a0c91e6bf6</RqUID>
            <SVK_OrderAddRs Id="bd54493a-71c3-4ca1-8ee8-00a0c91e6bf6">
               <Status>OK</Status>
               <RqUID>bd54493a-71c3-4ca1-8ee8-00a0c91e6bf6</RqUID>
               <SVK_DomXferInfo>
                  <!-- obsah rovnaky ako v poziadavke -->
               </SVK_DomXferInfo>
               <ClientDt>2006-04-05T21:32:14.367352+01:00</ClientDt>
               <ServerDt>2006-04-05T21:32:15.198663+01:00</ServerDt>
               <Signature xmlns="...">
                  ...
                  <SignedInfo>
                     <Reference URI="#bd54493a-71c3-4ca1-8ee8-00a0c91e6bf6">
                        ...
                     </Reference>
                  </SignedInfo>
                  <SignatureValue>G84CRFrfL0tR9i=...</SignatureValue>
               </Signature>
            </SVK_OrderAddRs>
         </BankSvcRs>
      </IFXResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Prvý riadok HTTP odpovede nás informuje o tom, že požiadavka ako celok bola vybavená úspešne. Obsah odpovede je analogický požiadavke: V tele HTTP odpovede je SOAP obálka obsahujúca iba SOAP Body a v ňom IFXHeader (zreplikovaný z požiadavky) a IFXResponse element.
Obsah IFXResponse zodpovedá obsahu IFXRequest-u. Je tu uvedená odpoveď na požiadavku klienta o prihlásenie (prebehlo úspešne a jazyk pre komunikáciu bude SK - slovenčina). Ďalej je tu uvedená odpoveď (BankSvcRs) na príslušnú servisnú požiadavku a v rámci nej odpoveď (SVK_OrderAddRs) na príslušnú individuálnu požiadavku. Všimnime si najprv, že RqUID sú pre všetky požiadavky zreplikovane do príslušnej odpovede. Každá odpoveď okrem toho môže obsahovať Status agregát (v prípade jeho neprítomnosti sa predpokladá, že všetko prebehlo OK). Replikácia, ktorú sme spomenuli je zámerná – je to prvok komunikačného protokolu IFX. Okrem už spomínaného Status elementu je tu uvedený datetime spracovania požiadavky serverom (element ServerDt). Tento slúži pre potreby vymedzenia času odpovede najmä z dôvodu, že táto je podpísaná (opäť pomocou Signature) – tento krát serverom a podpis slúži ako potvrdenka pre klienta o tom, že ju server kladne vybavil.

Vo zvyšných podkapitolách Návrhu si podrobnejšie popíšeme štruktúru vymieňaných správ, komunikačný protokol, proces spracovania požiadaviek a ošetrenie chýb a bezpečnosť. Nakoniec stručne popíšeme operácie a správy v návrhu.

4.2 Štruktúra vymieňaných správ

Služby, ktoré bude server poskytovať klientom sú realizované pomocou operácií a operácie sú realizované prostredníctvom výmeny správ. V nasledovnom texte sa pozrieme ako tieto správy vyzerajú – aká je ich štruktúra. Začneme od stavebných blokov, ktorými sú elementy definované na základe IFX dátových typov, pokračovať budeme agregátmi, ktorú tvoria štruktúru z elementov a ďalších agregátov. Následne si popíšeme štruktúru *jednotlivých* požiadaviek a odpovedí, *servisných* požiadaviek a odpovedí a nakoniec *IFX* požiadaviek a odpovedí. Nakoniec popíšeme, ako sa IFX požiadavky a odpovede budú prenášať v rámci SOAP obálky a uvedieme WSDL dokument pre nás webservis, ktorý umožní výmenu takýchto SOAP obálok.

4.2.1 Dátové typy

IFX definuje niekoľko základných dátových typov (Tabuľka 2). Všetky elementy a agregáty sú definované pomocou týchto dátových typov, resp. ako štruktúry nad týmito dátovými typmi. V tabuľke sú zvýraznené dátové typy, ktoré my využívame v našom návrhu. Ostatne nevyužívame buď z dôvodu, že ich v momentálne nevyužíva ani IFX (aj keď ich definuje - YrMon), alebo kvôli tomu, že elementy a agregáty v našom návrhu nie sú definované pomocou týchto typov.

Character	**Closed Enum**
Narrow Character	**Open Enum**
Binary	**Long**
Boolean	**Identifier**
~~YrMon~~, **Date**, Time, **DateTime**, Timestamp	**Phone Number**
Decimal	**Universally Unique Identifier (UUID)**
Currency Amount	URL
HexBin	

Tabuľka 2: Dátové typy IFX. Tučným sú vyznačené dátové typy, ktoré my používame v našom návrhu.

Tieto dátové typy sú v našom návrhu definované pomocou XML Schémy. Každý dátový typ je definovaný ako simpleType alebo complexType XML schémy s prípadnými reštrikciami aby daná definícia presne zodpovedala IFX dátovým typom.
Kvôli nedostatku miesta a nie prílišnej zaujímavosti jednotlivé dátové typy podrobne rozoberať nebudeme. Čitateľ ich vyčerpávajúci popis nájde v [IFXBMS].

Character – dátový typ reprezentujúci reťazce znakov. Označenie: **C-n** = znakový reťazec dĺžky maximálne n. **C** = znakový reťazec bez obmedzenia dĺžky. V XML Schéme: xsd:string.
Narrow Character - ako *Character* s dodatočným obmedzením, že každý znak musí byť zo znakovej sady ISO Latin-1. Označenie: **NC-n** a **NC**. V XML Schéme: xsd:string.
Boolean – reprezentuje logickú hodnotu *true*, alebo *false*. Booleovská hodnota je reprezentovaná ako NC-1, t.j. znakový reťazec dĺžky jedna, pričom "1" reprezentuje *true* a "0" *false*. Defaultná hodnota elementu typu boolean v prípade jeho absencie v agregáte je *false*. Označenie: Boolean.
Enum – je definovaný ako NC-80. Element typu Enum môže nadobúdať jednu z definovaných (vymenovaných) hodnôt. má dva podtypy, ClosedEnum a OpenEnum. Element typu Closed Enum môže nadobúdať iba hodnoty, ktoré sú presne definované v tomto návrhu (resp. v IFX špecifikácii), iné hodnoty sú zakázané. Open Enum môže nadobúdať aj ľubovoľné dodefinované hodnoty, na ktorých použití sa komunikujúce strany dohodnú. Označenie: OpenEnum a ClosedEnum.

Binary – dátový typ reprezentujúci binárne dáta zakódované napr. pomocou base64 kódovania. Označenie: Binary. V XML schéme: complexType s troma elementami: samotne binárne dáta, typ kódovania, dĺžka kódovaného reťazca pred zakódovaním.
Date a **DateTime** – dátové typy reprezentujúce dátum a (dátum a čas). DateTime sa vždy uvádza v UTC čase. Označenie: Date a DateTime.
Decimal – dátový typ reprezentujúci desatinné čísla. Označenie: Decimal.
Long - dátový typ reprezentujúci celé čísla v rozsahu [–2147483648, 2147483647]. Označenie: Long.
Currency Amount – komplexný dátový typ reprezentujúci *množstvo peňazí v mene*. Napr. 100 Sk. Môže obsahovať aj kurz na prepočet tohto množstva do inej meny. Označenie: CurrencyAmount
Identifier – dátový typ reprezentujúci jednoznačný identifikátor. Definovaný ako NC-36. Označenie: Identifier.
Universally Unique Identifier (UUID) – Identifikátor, ktorý je (resp. cieľ je aby bol) unikátny v priestore a čase (univerzálne unikátny). Existuje niekoľko spôsobov ako takýto identifikátor vygenerovať. V IFX sa používajú UUID vygenerovane na základe 1. IEEE 802 adresy sieťovej karty počítača, 2. aktuálneho času, 3. stavu procesora.
Príklad: f81d4fae-7dec-11d0-a765-00a0c91e6bf6. Označenie: UUID.
URL – dátový typ definovaný ako NC-1024 reprezentujúci URL.

4.2.2 (IFX) Element

XML element s obmedzením, že môže obsahovať iba dáta, teda nie iné XML elementy, pričom nesmie byť prázdny a nesmie obsahovať iba whitespace. Každý element v IFX je definovaný na základe nejakého IFX základného dátového typu – budeme hovoriť že element „je typu X“. Príklad: element Memo je typu C-255, čiže obsahuje reťazec maximálne 255 znakov. V tabuľke to budeme zapisovať nasledovne:

Názov	*Typ*	*Výskyt*	*Popis*
Memo	C-255		Textový záznam.

4.2.3 Agregát

Agregát je XML element, ktorý môže obsahovať IFX elementy a opäť agregáty. Nesmie obsahovať dáta alebo kombináciu dát a elementov. Môže byť aj prázdny. Ako príklad sa pozrime na štruktúru Status agregátu, ktorý sa používa v odpovedi na každý požiadavku klienta.

Status

Názov	*Typ*	*Výskyt*	*Popis*
StatusCode	Long	**Povinný**	Status kód.
ServerStatusCode	C-20	Nepovinný	Server Status kód. Táto hodnota môže byť zobrazená užívateľovi. Ten môže túto hodnotu napríklad nahlásiť poskytovateľovi služieb napríklad pre potreby debuggovania.
Severity	Closed Enum	**Povinný**	Vážnosť stavu. Povolene hodnoty: Info, Warn, Error.
StatusDesc	C-255	Nepovinný	Popis stavu. Vysvetľujúci/popisný text.
AdditionalStatus	Agregát	Nepovinný opakujúci sa	Dodatočné Status agregáty. Horeuvedený StatusCode musí obsahovať primárny status kód, pričom tento agregát môže byť použitý pre zahrnutie každého dodatočného stavu, ktorý sa server rozhodne poskytnúť. Príklad: Ak aj číslo účtu aj dátum obsahujú neplatné hodnoty, prvý Statuscode môže obsahovať jednu z týchto chýb a tento agregát môže obsahovať tú druhú.

Pomocou takejto tabuľky budeme popisovať agregáty, ak budeme chcieť popísať ich štruktúru a obsah. Horeuvedená tabuľka definuje agregát s názvom Status, ktorý môže obsahovať niekoľko potomkov. Pre každého z nich máme uvedené 1. Názov – jeho meno, 2. Typ – dátový typ tohto elementu v prípade, že ide o element alebo je uvedené, že ide o agregát, 3. Výskyt – špecifikácia výskytu potomka v rámci agregátu, či je povinný, nepovinný, pripadne iné typy výskytu, 4. Popis – popis tejto položky: čo obsahuje a na čo sa používa v rámci štruktúry daného agregátu.

4.2.4 Správa (Message)

Agregát, ktorý reprezentuje ucelenú jednotku funkcionality. Správa je buď požiadavkou (request) alebo odpoveďou (response). Prvú posiela klient serveru a druhú server klientovi. Napr. SignonRq je žiadosť klienta o prihlásenie do systému, resp. autentifikáciu. SignonRs je odpoveďou serveru na túto požiadavku. Signon správy reprezentujú ucelenú jednotku funkcionality, ktorou je v tomto prípade prihlásenie klienta do systému.

4.2.5 Typy správ

IFX definuje niekoľko typov správ. Každá správa okrem Signon a Signoff správ je niektorého z nasledujúcich typov.

a) **Add** správy (xxxAddRq a xxxAddRs). Add požiadavka sa používa na vytvorenie nového objektu xxx na serveri. Napr. SVK_OrderAddRq vytvorí nový bankový príkaz.
b) **Delete** a **Cancel** správy (xxxCanRq a xxxCanRs, resp. xxxDelRq a xxxDelRq) sa používajú na zmazanie nejakého objektu (Delete) alebo zrušenie nejakej operácie (Cancel).
c) **Inquiry** správy (xxxInqRq a xxxInqRs) sa používajú na zistenie informácií o objektoch a operáciách existujúcich na strane serveru a stave týchto objektov a operácií. Klient zadá niekoľko z možných *filtračných kritérií* a server na základe týchto filtračných kritérií nájde na serveri objekty, ktoré týmto kritériám *vyhovujú* a pošle ich (resp. informácie o nich) klientovi v odpovedi.

My si v našom návrhu vystačíme s horeuvedeným typmi správ. IFX ich definuje ešte niekoľko. Tu uvádzame ich stručný prehľad. Detaily si záujemca môže prečítať v [IFXBMS].

d) **Modify** správy slúžia na modifikáciu objektov existujúcich na strane serveru.
e) **Audit** správy. Umožňujú audit objektov. Server podporujúci audit si zaznamenáva všetky správy, ktoré vytvoria alebo modifikujú objekty na serveri. V prípade potreby môže klient poslať Audit požiadavku týkajúcu sa konkrétneho objektu a vrátia sa mu všetky správy, ktoré tento objekt modifikovali (vytvorili, modifikovali alebo zmazali) počas celej jeho existencie od vytvorenia až doteraz. Prípadne môže klient použiť dodatočné filtračné kritéria, ktoré vrátia užšiu množinu správ pre daný objekt. Audit teda môžeme použiť, ak chceme zistiť históriu niektorého objektu/objektov, pripadne pre účely synchronizácie pre stavových klientov[48].
f) **Synchronize** správy. Umožňujú synchronizovať lokálny stav stavových klientov s aktuálnym stavom na serveri. Synchronizácia má význam hlavne v prostredí, kde užívateľ alebo užívatelia používajú viac klientov na vykonávanie aktívnych operácii nad jedným užívateľským kontom. Takto sa môže stať, že stavový klient si pamätá stav o nejakom objekte, ktorý však bol medzitým modifikovaný druhým klientom. V tomto momente je prvý klient rozsynchronizovaný so stavom na serveri a potrebuje sa nejakým spôsobom synchronizovať. Spravidla sa na to používa takzvaný token, ktorý identifikuje poslednú odpoveď, ktorú tento klient obdržal. Sever potom klientovi môže poslať odpovede na všetky modifikujúce požiadavky od

[48] stavový klient je taký, ktorý si lokálne udržiava informácie o stave objektov, o vykonaných operáciách a podobne.

momentu, ktorý definuje token. Toto umožní klientovi aktualizovať svoj stav. V našom návrhu nie sú zahrnuté správy, ktoré by umožňovali klientom vykonávať synchronizáciu. Cieľovým typom klienta pre náš návrh je *stateless* klient, t.j. taký, ktorý si stav lokálne neudržiava, pripadne udržiava, ale nepožaduje synchronizáciu.

g) **Advise** správy. Umožňuje „odporučiť" vykonanie nejakej operácie, alebo informovať aktéra (klienta, server) o stave objektu. Viac v [IFXBMS].
h) **Reversal** správy. Umožňuje vykonať „undo" nejakej požiadavky, ktorú klient poslal. Reversal správa je podobná Delete a Cancel typom správ, ale má silnejšiu sémantiku. Požaduje totiž nielen aby bola príslušná operácia zrušená alebo objekt zmazaný, ale aby všetky akcie na strane serveru, ktoré sa na základe pôvodnej požiadavky vykonali, boli zvrátené, t.j. aby sa všetko vrátilo do pôvodného stavu.

4.2.6 Služba

Jednotlivé správy sú podľa funkcie, ktorú vykonávajú (Bankovníctvo apod.) zaradené do jednotlivých tried správ, ktoré nazývame Služby (Services). My v našom návrhu budeme používať dve služby, Banking Service a Base Service, pričom prvá sa týka striktne bankových služieb a druhá obsahuje operácie, ktoré sú spoločné pre rôzne typy finančných interakcii – aj pre bankové interakcie.

Jednotlivé požiadavky a odpovede sa medzi klientom a serverom vymieňajú v rámci servisných správ. Požiadavky pre danú službu sú koncentrovane v servisnej požiadavke a zodpovedajúce odpovede sú odoslane späť v rámci servisnej odpovede. Štruktúra servisných agregátov je nasledovná.

Servisná požiadavka <xxx**SvcRq**>

Názov	*Typ*	*Výskyt*	*Popis*
RqUID	UUID	**Povinný**	Identifikátor tejto požiadavky.
***xxx*Rq**	Správa	Nepovinný opakujúci sa	Niekoľko individuálnych požiadaviek zo služby xxx.

Servisná odpoveď <xxxSvcRs>

Názov	*Typ*	*Výskyt*	*Popis*
Status	Agregát	Nepovinný	Stav odpovede.
RqUID	UUID	REPLIKA **Povinný**	Identifikátor požiadavky, ku ktorej táto odpoveď prislúcha.
***xxx*Rs**	Správa	Nepovinný opakujúci sa	Množina individuálnych odpovedí zodpovedajúca množine individuálnych požiadaviek zo servisnej požiadavky.

Čiže klient pošle v servisnej požiadavke nula alebo viac *jednotlivých* požiadaviek z danej služby, tieto požiadavky sú spracovane serverom, ktorý pošle späť servisnú odpoveď, ktorá obsahuje zodpovedajúce jednotlivé odpovede. Významu a použitiu agregátov Status a RQUID sa onedlho vrátime.

4.2.7 IFX Dokument

Jednotka komunikácie medzi klientom a serverom. IFX dokument obsahuje nula alebo viac servisných požiadaviek alebo odpovedi. Klient posiela serveru IFX požiadavku a Server na ňu odpovedá IFX odpoveďou.
IFX požiadavka povinne obsahuje požiadavku na prihlásenie (SignonRq), nula alebo viac servisných požiadaviek a môže obsahovať požiadavku na odhlásenie (Signoff).

IFX požiadavka <IFXRequest>

Názov	*Typ*	*Výskyt*	*Popis*
SignonRq	Správa	**Povinný**	Požiadavka o prihlásenie.
***xxx*SvcRq**	Servisná správa	Nepovinný opakujúci sa	Servisná požiadavka.
SignoffRq	Správa	Nepovinný	Požiadavka o odhlásenie.

IFX odpoveď <IFXResponse>

Názov	*Typ*	*Výskyt*	*Popis*
Status	Agregát	Nepovinný	Stav odpovede.
SignonRs	Správa	Nepovinný *Ale pozri popis*	Odpoveď na požiadavku o prihlásenie. Musí byť uvedená v prípade, že IFX požiadavka bola validná.
***xxx*SvcRs**	Servisná správa	Nepovinný opakujúci sa *Ale pozri popis*	Servisná odpoveď. Pre každú servisnú požiadavku z IFX požiadavky tu musí byť uvedená servisná odpoveď.
SignoffRq	Správa	Nepovinný *Ale pozri popis*	Odpoveď na požiadavku o odhlásenie. Musí byť uvedená v prípade, že IFX požiadavka bola validná a klient sa dobre autentifikoval v SignonRq.

Čiže klient pošle serveru IFX požiadavku ktorá obsahuje žiadosť o prihlásenie, niekoľko servisných požiadaviek a možno odhlásenie. Server odošle späť IFX odpoveď, ktorá reflektuje obsah a validitu IFX požiadavky.

4.2.8 SOAP Obálka

IFX dokumenty sa vymieňajú v rámci SOAP obálok. Na nasledujúcom diagrame (Diagram 23) vidíme ako by mohla vyzerať SOAP obálka v prípade, že posielame IFXRequest.
V hlavičke SOAP obálky môžeme uviesť (nepovinne) WS Security element s podpisom celého tela SOAP obálky. V samotnom tele SOAP obálky je potom IFXHeader (ktorého obsah popíšeme neskôr) a IFXRequest. Štruktúra SOAP obálky pre odpoveď je identická až na to, že IFXRequest je nahradený príslušnou odpoveďou – IFXResponse. V prípade, že dôjde k vygenerovaniu chyby, SOAP Obálka v prípade odpovede môže mať značne odlišnú štruktúru, ale tú si rozoberieme až pri procesnom modeli.

```
<env:Envelope>
   <env:Header>
      <wsse:Security>
         <ds:Signature>
            <!--podpis SOAP Body-->
         </ds:Signature>
      </wsse:Security>
   </env:Header>
   <env:Body Id="SOAPBody">
      <ifx:IFXHeader>
         ...
      </ifx:IFXHeader>
      <ifx:IFXRequest>
         ...
      </ifx:IFXRequest>
   </env:Body>
</env:Envelope>
```

Diagram 22: Štruktúra SOAP Obálky s IFX požiadavkou

Čiže, v sumáre. Klient a server si vymieňajú SOAP obálky s IFX správami – klient posiela IFX požiadavku, na ktorú server odpovie IFX odpoveďou. IFX požiadavka obsahuje prihlásenie a (možno aj) odhlásenie užívateľa a niekoľko servisných požiadaviek. IFX odpoveď obsahuje odpovede na jednotlivé požiadavky odoslane v IFX požiadavke. IFX servisná požiadavka obsahuje niekoľko jednotlivých požiadaviek pre danú službu (napr. Bankovníctvo), IFX servisná odpoveď obsahuje jednotlivé odpovede na jednotlivé požiadavky z IFX servisnej požiadavky. Jednotlivé požiadavky a odpovede (=správy) sú agregáty, ktoré obsahujú niekoľko elementov a agregátov. Agregát obsahuje elementy a iné agregáty. Element obsahuje iba dáta. Každý element je určitého typu – jedného zo základných dátových typov IFX.

4.2.9 WSDL dokument

Webová služba pre výmenu IFX správ je definovaná nasledovne:

```
<?XML version="1.0" encoding="UTF-8"?>
<definitions name="IFXWS" xmlns:soap="..." xmlns:ifxws="..." xmlns="..." xmlns:ifx="..."
             targetNamespace="..." >
   <types>
      ...
   </types>
   <message name="IFXRequest">
      <part name="header" element="ifx:IFXHeader"/>
      <part name="reqeuest" element="ifx:IFXRequest"/>
   </message>
   <message name="IFXResponse">
      <part name="header" element="ifx:IFXHeader"/>
      <part name="response" element="ifx:IFXResponse"/>
   </message>
   <portType name="IFXWebServicePortType">
      <operation name="doIFX">
         <input message="ifxws:IFXRequest"/>
         <output message="ifxws:IFXResponse"/>
      </operation>
   </portType>
   <binding name="IFXWebServiceBinding" type="ifxws:IFXWebServicePortType">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="doIFX">
         <soap:operation soapAction="http://michal.rajniak.net/SKFX/doIFX"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
      </operation>
   </binding>
   <service name="IFXWebService">
      <port name="IFXWebServicePort" binding="ifxws:IFXWebServiceBinding">
         <soap:address location="https://localhost:1234/axis/services/IFX"/>
      </port>
   </service>
</definitions>
```

Diagram 23: WSDL pre výmenu IFX správ.

Vidíme, že WSDL dokument je veľmi jednoduchý a umožňuje priamočiaro výmenu IFX správ ako parametrov jedinej operácie – `doIFX`.

4.3 Niektoré prvky komunikačného protokolu

V tejto podkapitole si povieme niečo o základných prvkoch komunikačného protokolu IFX a elementoch/agregátoch pomocou ktorých sú tieto prvky realizované.

4.3.1 Identifikovanie požiadaviek, korelácia, kontrola duplikátov – RqUID

Klient používa RqUID na identifikáciu požiadaviek, ktoré posiela serveru. Je to povinné pole pre všetky jednotlivé požiadavky (okrem SignonRq a SignoffRq) a pre všetky servisné požiadavky. Server v každej (jednotlivej alebo servisnej) odpovedi *zreplikuje* príslušné RqUID, aby klient mohol spárovať požiadavky a odpovede.
RqUID sa používa na koreláciu, t.j. spárovanie požiadaviek a príslušných odpovedí v rámci IFX požiadaviek a odpovedí a takisto sa používa na detekciu duplikátov – jedna RqUID hodnota totižto môže byť použitá (globálne v priestore a čase) práve raz. Server si všetky použité hodnoty (pre daného užívateľa) archivuje a požiadavky s RqUID, ktoré už bolo použité, sú odmietnuté. Treba podotknúť, že RqUID nemusia nutne tvoriť usporiadanie, t.j. server nemôže na základe jednotlivých hodnôt RqUID predpokladať, že niektoré zodpovedá požiadavke vygenerovanej skôr alebo neskôr.

Názov	*Typ*	*Výskyt*	*Popis*
RqUID	UUID		Jednoznačný identifikátor požiadavky klienta. Používa sa na koreláciu požiadaviek a odpovedi a na detekciu duplikátov.

4.3.2 Informácie o výsledku spracovania požiadavky – Status agregát

Status agregát môže poslať server v každej odpovedi – v odpovedi na jednotlivú požiadavku, v servisnej odpovedi, alebo v IFX odpovedi. Server v ňom informuje klienta o stave vybavenia požiadavky. V prípade, že Status agregát nie je uvedený, predpokladá sa StatusCode rovný 0, t.j. bezproblémové vybavenie.

Výsledky vybavenia požiadavky môžeme rozdeliť do troch tried „vážnosti“ (severity): Info, Warn, Error.
Info znamená, že serveru požiadavku (jednotlivú, servisnú, dokument) prijal a spracoval a posiela odpoveď.
Warn znamená, že server požiadavku prijal a spracoval a posiela odpoveď, ale nevykonal požiadavku v plnom rozsahu alebo v plnej kvalite, alebo chce klienta upozorniť na nejakú dôležitú skutočnosť, ktorá nastala pri vykonávaní požiadavky. Napr.: Server nefiltroval podľa niektorých filtračných kritérií uvedených v Inquiry požiadavke, pretože ich nepodporuje.
Error znamená, že server odmietol správu vybaviť buď z dôvodu, že bola invalidná, alebo vznikla chyba na strane serveru pri jej spracovaní.
V prípade že Severity (vážnosť) je Info alebo Warn, server musí vygenerovať kompletný response. Keď Severity = Error, server je povinný v odpovedi uviesť iba Status agregát (ktorý je v tomto prípade povinný) a zreplikovať RqUID z požiadavky, ak mu v tom daná chyba nezabráni.

Štruktúra Status agregátu je nasledovná.

Meno	*Typ*	*Výskyt*	*Popis*
StatusCode	Long	**Povinný**	Status kód. Povolene hodnoty závisia od kontextu (o aký typ požiadavky ide...)
ServerStatusCode	C-20	Nepovinný	Server Status kód. Táto hodnota môže byť zobrazená užívateľovi. Ten môže túto hodnotu napríklad nahlásiť poskytovateľovi služieb pre potreby debuggovania.
Severity	ClosedEnum	**Povinný**	Vážnosť stavu. Povolené hodnoty: Info, Warn, Error.
StatusDesc	C-255	Nepovinný	Popis stavu. Vysvetľujúci/popisný text.
AdditionalStatus	Agregát	Nepovinný opakujúci sa	Dodatočné Status agregáty. Horeuvedený StatusCode musí obsahovať primárny status kód, pričom tento agregát môže byť použitý pre

			zahrnutie každého dodatočného stavu, ktorý sa server rozhodne poskytnúť. Príklad: Ak aj číslo účtu aj dátum obsahujú neplatné hodnoty, prvý Statuscode môže obsahovať jednu z týchto chýb a tento agregát môže obsahovať tú druhú.

Povolené hodnoty Status kódov (element StatusCode) závisia od kontextu. Napríklad Status kód 1740: „Authentication failed" má zrejme význam uvádzať iba v odpovedi na požiadavku klienta o autentifikáciu (SignonRs). Povolene hodnoty sú uvedené v jednom z dodatkov [IFXBMS][49].

4.3.3 Zotavenie z chýb na báze súborov[50]

Predstavme si, že klient pošle požiadavku serveru a kvôli nejakej chybe – či už na strane klienta alebo serveru alebo na komunikačnom kanále – neobdrží odpoveď na túto požiadavku. V tomto prípade klient nevie, či bola požiadavka vybavená kladne, záporne, alebo či bola vôbec prijatá[51].
Zotavenie z chýb na báze súborov (ďalej FBER) umožňuje klientovi riešiť túto a podobne situácie. Za týmto cieľom sa používajú polia NewFileUID a OldFileUID v IFXHeader agregáte, ktorý sa posiela s každou IFX požiadavkou/odpoveďou. Jeho obsah je nasledovný.

Meno	*Typ*	*Výskyt*	*Popis*
Version	NC-20	**Povinný**	Verzia IFX, ktorú používa klient. Server túto hodnotu v odpovedi iba replikuje.
NewFileUID	UUID	Nepovinný	Identifikátor tejto IFX požiadavky. Pri odpovedi: Identifikátor IFX požiadavky, na ktorú server posiela odpoveď. Používa sa na zotavenie z chýb na báze súborov.
OldFileUID	UUID	Nepovinný	Identifikátor poslednej IFX požiadavky, ktorú klient od serveru úspešné obdržal a spracoval. Server túto informáciu môže použiť na manažment archivovaných odpovedi.

V nasledovných odstavcoch si stručne popíšeme, ako je FBER realizovaná. Kompletnejší popis nájde čitateľ napríklad v [IFXHTTP].

Klient k IFX požiadavke priloží NewFileUID. Ak server obdrží požiadavku, ktorá má uvedené NewFileUID a odpovie na ňu a archivuje si príslušnú (kompletnú) IFX odpoveď pre prípad, že by ju klient neobdržal.
V prípade, že klient postráda odpoveď na niektorú požiadavku[52], môže kompletnú kópiu tejto požiadavky poslať serveru znova. V prípade, že ju server prvýkrát úspešné prijal, spracoval a aj na ňu odpovedal, ale odpoveď klientovi neprišla, server iba odošle uloženú odpoveď[53]. V prípade, že ide o nove NewFileUID (samotná požiadavka nedorazila), server ju spracuje ako novu, pošle odpoveď a uloží si ju pre prípad, že by ju klient v budúcnosti požadoval.
Takýmto spôsobom môžeme jednoducho riešiť prípad, ktorý sme uviedli na začiatku: Ak klient neobdrží odpoveď, pošle požiadavku ešte raz a obdrží odpoveď teraz snáď už v poriadku.
V prípade, že klient pošle požiadavku bez NewFileUID, server si pre ňu odpoveď nearchivuje a klient prirodzene nemá ako explicitne žiadať o odpoveď.

[49] A je ich *veľa*.
[50] File based error recovery
[51] Môže si síce prezerať zadané príkazy na serveri a iné informácie, tieto však už nemusia obsahovať všetky informácie uvedené v prípadnej odpovedi.
[52] ku ktorej prilozil NewFileUID
[53] Odpoveď nemusí, dokonca by nemal poslať, v prípade, že obsah pôvodnej požiadavky nie je identicky obsahu tejto požiadavky

Klient okrem NewFileUID môže poslať aj OldFileUID, čo je identifikátor poslednej IFX požiadavky, ktorú klient od serveru úspešné obdržal a spracoval. Server môže na základe tohto napríklad príslušnú archivovanú odpoveď zmazať.

Otázka je, koľko odpovedi si server potrebuje archivovať. Existuje niekoľko možnosti: buď si bude pamätať len poslednú odpoveď, niekoľko (N) posledných odpovedi, alebo si bude pamätať všetky odpovede za nejakú dobu (napr. posledne 2 mesiace). Pripadne môže tieto možnosti kombinovať s informáciami od užívateľa vo forme OldFileUID. Analýza týchto možnosti je zdĺhavá a nie príliš zaujímavá. Pre všetky praktické potreby stačí, ak si server pamätá všetky požiadavky za nejakú dobu (napr. 2 mesiace dozadu). Pre ostatné možnosti a ich kombinácie treba vziať do úvahy počet klientov, ktoré chce užívateľ (súčasne) používať a akým spôsobom (či chce používať FBER), či server bude brat do úvahy OldFileUID a akým spôsobom atd.

4.3.4 Sedenia (Sessions)

Vieme, že klient sa musí v *každej* IFX požiadavke autentifikovať pomocou SignonRq. Neustále posielanie kompletných autentifikačných údajov môže byť pre klienta nevýhodné a preto server umožňuje klientovi požiadať pri autentifikácii o vygenerovanie session kľúča[54].

Meno	*Typ*	*Výskyt*	*Popis*
SessKey	NC-64		Session kľúč. Slúži na autentifikáciu klienta a definuje session (sedenie).

Súčasťou SignonRs odpovede pri vygenerovaní session kľúča je aj čas vypršania jeho platnosti[55]. Týmto spôsobom je prirodzene definované sedenie, v rámci ktorého sa môže klient autentifikovať v rámci SignonRq pomocou tohto kľúča.

V kontexte použitia viacerých klientov jedného užívateľa súčasne (jedno užívateľské konto) vyvstáva otázka, či povoliť vygenerovanie viacerých session kľúčov pre týchto klientov, alebo im umožniť zdieľať jeden session kľúč (takto to defaultne predpokladá IFX), alebo úplne zamedziť možnosť prístupu viacerých klientov pre jedného užívateľa súčasne. Tieto možnosti majú svoje implikácie pre použiteľnosť, jednoduchosť implementácie a pre bezpečnosť a treba ich zvážiť pri implementácii.

4.3.5 Asynchrónne vyzdvihnutie odpovede

V prípade, že server prijal od klienta požiadavku, na ktorú mu nedokáže odpovedať hneď, ale bude mu to trvať určitý čas, môže klienta notifikovať o tomto fakte a odporučiť mu vyzdvihnutie si výsledku neskôr. IFX podporuje túto funkcionalitu tým spôsobom, že v okamžitej odpovedi, ktorú klientovi na danú požiadavku pošle, uvedie tzv. asynchrónny unikátny identifikátor požiadavky (AsyncRqUID) a časový rozsah, v ktorom bude odpoveď k dispozícii. Klient v definovanom časovom rozsahu potom môže poslať požiadavku v ktorej uvedie RqUID pôvodnej požiadavky ale aj AsyncRqUID a server pošle späť odpoveď na pôvodnú požiadavku (v prípade, že už je pripravená na vyzdvihnutie). My túto funkcionalitu v našom návrhu momentálne nebudeme podporovať, ale v prípade potreby môže byť ľahko pridaná v budúcnosti.

4.3.6 Replikácia v odpovediach

Obsah každej požiadavky sa v IFX defaultne *replikuje*[56] do príslušnej odpovede. Predpokladajme, že pošleme jednoduchú Inquiry požiadavku, ktorá vyhľadá všetky platby,

---

[54] Presnejší názov by bol session cookie (koláčik). Termín *kľúč* môže nabádať čitateľa k domnienke, že session key sa používa na kryptografické účely, ale nie je tomu tak. Ide skutočné o koláčik.
[55] Element `SignonRs/ExpDt`.
[56] Podľa IFX: *echoing in responses*; replikovane položky sa nazývajú *echoed* fields, values atd.

v ktorých sa zaplatilo 1000Sk. Požiadavka teda obsahuje: RqUID a dve filtračné kritéria: suma(1000) a mena(SK). V odpovedi sú defaultne[57] zreplikovane RqUID aj tieto dve filtračné kritéria s pôvodnými hodnotami a okrem toho je tu samozrejme výsledok požiadavky – Status agregát a záznamy o vyhovujúcich platbách.
Podotýkame, že niektoré položky (napr. pri autentifikácii) sa nereplikujú do odpovede z bezpečnostných alebo iných dôvodov.
Pozn: Replikácia sa uplatňuje aj na IFXHeader – ten sa vždy (okrem prípadu závažnej chyby) kompletne zreplikuje do odpovede, presnejšie, do SOAP Body pred IFXResponse.

4.4 Proces spracovania požiadaviek

V tomto momente máme dosť informácií na to, aby sme popísali spôsob, ako budú IFX požiadavky prijímané a spracovávané serverom, aké chyby môže pri spracovaní nastať, ako sa tieto chyby zobrazia do štruktúru IFX odpovede (a pripadne do štruktúry SOAP Fault) a ako sú v závislosti od tohto odpovede (korektné alebo informujúce o chybe) odosielané späť klientovi.

Zopakujme si najprv stručne niektoré základné idey, ktoré budeme využívať v tejto kapitole.
V našom návrhu vystupujú iba dva typy SOAP uzlov: klient a server. Na ceste SOAP správy nemáme žiadnych prostredníkov. SOAP správy sa vymieňajú cez sieť pomocou HTTP protokolu – používame štandardné naviazanie SOAPu na HTTP z druhej časti SOAP špecifikácie, pričom používame iba request/response MEP[58].
Pri request/response najprv klient pošle SOAP obálku (a v nej IFXRequest) v rámci HTTP POST požiadavky. Jej štruktúra je nasledovná (neformálne):

```
<env:Envelope>
   <env:Header>
      <wsse:Security>
         <ds:Signature><!--podpis SOAP Body--></ds:Signature>
      </wsse:Security>
   </env:Header>
   <env:Body Id="SOAPBody">
      <ifx:IFXHeader><!--Version, NewFileUID, OldFileUID--></ ifx:IFXHeader>
      <ifx:IFXRequest>...</ifx:IFXRequest>
   </env:Body>
</env:Envelope>
```

Z kapitoly o štruktúre vieme, že IFXRequest obsahuje niekoľko servisných požiadaviek, pričom každá z nich obsahuje niekoľko individuálnych požiadaviek.

Takáto SOAP obálka je prijatá serverom a následné je spracovaná. Prvý je spracovaný hlavičkový blok Security a následné telo SOAP správy, ktoré obsahuje IFX požiadavku. Na základe priebehu a výsledku spracovania IFX požiadavky je vygenerovaná IFXResponse a je odoslaná klientovi. Pripomeňme si, že štruktúra IFXResponse je nasledovná.

[57] Dá sa požiadať o nereplikovanie – toto sa robí globálne pre celu IFX požiadavku v SignonRq.
[58] Pre pripomenutie: Message exchange pattern.

```
<IFXResponse>
   <Status>
      ...
      <Severity>jeden z {Error, Info, Warn}</Severity>
      ...
   </Status>
   <xxxSvcRs>
      <Status>...</Status>
      <RqUID>3dd1ce96-292e-4cbf-9bde-00a0c91e6bf6</RqUID>
      <SomeIndividualRs>
         <Status>...</Status>
         <RqUID>bd54493a-71c3-4ca1-8ee8-00a0c91e6bf6</RqUID>
         <!-- obsah individualnej odpovede -->
      </SomeIndividualRs>
      <!-- dalsie individuálne požiadavky -->
   </xxxSvcRs>
   <!-- dalsie servisne požiadavky -->
</IFXResponse>
```

Všimnime si, že Status agregát sa nachádza na všetkých úrovniach odpovedí: v IFXResponse, v servisných požiadavkách aj v individuálnych požiadavkách. `IFXResponse/Status` budeme nazývať *globálny status*. Podotýkame ešte, že použitie Status agregátu je nepovinné v prípade, že spracovanie prebehlo v poriadku. V prípade chyby (na príslušnej úrovni) musí byť Status na príslušnej úrovni uvedený.

4.4.1 Procesný model IFX+SOAP

Predpokladajme teda, že IFX server práve prijal požiadavku (SOAP obálku), ktorá môže byť korektná ale aj chybná. Bude spracovaná nasledovným spôsobom.

Spracovanie SOAP Header

Podľa SOAP procesného modelu vieme, že najprv sú spracované SOAP hlavičkové bloky, ktoré sú *nasmerované* pre daný uzol. Týmto uzlom je v našom návrhu vždy server a teda všetky hlavičkové bloky sú určené preňho. SOAP Header môže v našom prípade obsahovať jediný hlavičkový blok, a to element Security, ktorý – ak sa vyskytne – obsahuje práve jeden podpis – podpis celého tela SOAP obálky. Server teda (v prípade výskytu) najprv spracuje tento hlavičkový blok a overí podpis. Ak nesedí, je vygenerovaný SOAP Fault. Čo presne bude obsahovať, si povieme v ďalšej sekcii tejto podkapitoly (sekcia Štruktúra odpovede).

Spracovanie SOAP Body a IFX požiadavky

Ako vieme, po hlavičkových blokoch je spracovane konečným príjemcom telo SOAP správy. Toto v našom prípade povinne obsahuje dva elementy: IFXHeader a IFXRequest. Pri ich spracovaní môže dôjsť k rôznym chybám. Môžeme si ich rozdeliť do dvoch tried:
Chyba typu 1: závažná chyba globálneho charakteru, ktorá zamedzí serveru spracovať IFX požiadavku ako celok. V prípade, že dôjde k chybe tohto typu, priamo v IFXResponse je uvedený iba Status agregát s vážnosťou stavu Error (`IFXResponse/Status/Severity`=Error), pričom servisné a individuálne odpovede nemusia byť uvedené. Chybou typu 1 je napríklad chyba štruktúry alebo dátových typov IFX požiadavky.
Chyba typu 2: chyba v rámci servisnej alebo individuálnej požiadavky. Tiež môže byť závažná, ale nezamedzí spracovaniu ostatných nesúvisiacich požiadaviek. V tomto prípade nadobúda `IFXResponse/Status/Severity` hodnotu Info alebo Warn a servisné a jednotlivé odpovede sú v IFXResponse uvedené spravidla pre všetky požiadavky z IFXRequest.

Teraz už môžeme pristúpiť k presnému popisu postupu spracovania.

1) Zvaliduje sa štruktúra a obsah IFXHeader elementu a skontroluje sa verzia, ktorú používa klient (`IFXHeader/Version`). V prípade, že zlyhá validácia alebo server nedokáže komunikovať s klientom používajúcim danú verziu IFX, je vygenerovaná chyba typu 1.
2) Je skontrolovaná štruktúra IFXRequest.
   Server skontroluje, čí prijatý IFXRequest má správnu štruktúru a či sedia dátové typy – t.j. je vykonaná validácia dokumentu podľa XML schémy, prípadne je skontrolované splnenie iných podmienok kladených na štruktúru a dátové typy, na vyjadrenie a automatickú kontrolu ktorých ale Schéma nemá dostatočné výrazové prostriedky.
   V prípade, že Táto validácia zlyhá, IFX požiadavka a ani žiadna jej časť nie je spracovaná a je vygenerovaná chyba typu 1.
   V prípade, že štruktúra je správna, ale niektoré elementy nie sú správneho dátového typu, odporúčame, aby server vrátil IFX odpoveď s nasledovnou štruktúrou. Na úrovni IFXResponse bude obsahovať Status agregát so Status kódom 200 – general data error. Zo servisných a individuálnych odpovedí budú zahrnuté iba tie, ktorých zodpovedajúce požiadavky obsahovali element s dátovou chybou. Tieto odpovede budú obsahovať iba repliku RqUID (ak toto samotne nebolo chybné) a Status agregát, v ktorom bude presne špecifikovaná daná dátová chyba.
3) Následné je spracovaný SignonRq, t.j. vykoná sa autentifikácia klienta a prihlásenie do systému. V prípade, že zlyhá autentifikácia (zlé autentifikačné hodnoty), IFX response bude obsahovať iba SignonRs a v ňom Status agregát so Status kódom 1740 – Authentication failed. Ide o chybu typu 1.
4) Následne je spracovaný zvyšok IFXHeader – zrejme iba v prípade, že je uvedený aspoň jeden z NewFileUID, OldFileUID.
   V prípade, že klient posiela NewFileUID, server skontroluje, čí eviduje v archíve odpoveď pre takéto UID. V prípade, že nie, ide o novú IFX požiadavku a server pokračuje v jej spracovaní a odpoveď, ktorá eventuálne vznikne, si archivuje.
   V prípade, že server dané UID v archíve eviduje, ide o pokus o zotavenie z chyby. V tomto prípade server skontroluje, čí má táto IFX požiadavka rovnaký obsah ako ta pôvodná. V prípade, že áno, môže spracovanie skončiť v tomto bode a server pošle späť archivovanú odpoveď. V prípade, že sa obsah požiadaviek nezhoduje, server vygeneruje chybu typu 1.
   V prípade, že je uvedené OldFileUID, server ho (v závislosti od implementácie) spracuje. V prípade, že jeho spracovanie vygeneruje chybu, prirodzene pôjde o chybu typu 1.
5) Zvyšok IFXRequest je spracovaný nasledovne:
   - Sú spracované *servisné* požiadavky v ľubovoľnom poradí (t.j. nie nutne v takom poradí ako sú uvedené v IFX Request), pričom v rámci servisnej požiadavky sú *individuálne* požiadavky spracované v sekvenčnom poradí – tak ako boli zadané klientom.
     Ak dôjde ku chybe pri spracovaní servisnej alebo individuálnej požiadavky, globálny status IFX odpovede v tomto prípade už nebude obsahovať status kód so závažnosťou Error, ale Info alebo Warn.
     Pre každú servisnú a individuálnu požiadavku je uvedená v IFXResponse servisná/individuálna odpoveď. Jedine v prípade, že dôjde ku chybe na úrovni servisnej požiadavky[59], jednotlivé odpovede v rámci nej zrejme nemá význam spracovávať a ani pre ne generovať odpovede.
   - Nakoniec je spracovaný SignoffRq.
     Jediná chyba, ktorá môže nastať v prípade Signoff požiadavky, je prípad, keď sa klient pokúsi odhlásiť, pričom žiadna session momentálne neprebieha. Toto ale nie je problém, keďže klient vlastne požaduje stav, ktorý už momentálne existuje (ukončenie session vs. momentálna

[59] Jedine čo nás napadá, je duplikátne RQUID servisnej požiadavky.

neexistencia session). Toto nedopatrenie na strane klienta mu môžeme oznámiť v rámci globálneho statusu so závažnosťou Warn.

Počas spracovania môžu byť vygenerované nielen chyby z dôvodu nesprávneho obsahu požiadavky, čo je chyba klienta, ale môže zlyhať aj server. Prípady, kedy sa toto môže stať, rozoberať nebudeme, pretože to jednak nezávisí iba od štruktúry správy, ale najmä od toho, ako je IFX server naimplementovaný. Odporúčame však, aby chyby v tomto prípade boli generované a premietali sa do IFX odpovede "rozumným" spôsobom. Napríklad ak zlyhá iba jedna individuálna požiadavka, pretože pri jej spracovaní zlyhá jedna konkrétna databázová query, pričom ostatné požiadavky sa vybavia v poriadku, bolo by "rozumné" nevygenerovať globálnu chybu a "zahodiť" tak všetky ostatné korektne spracované požiadavky.

Pozrime sa teraz, ako bude v závislosti od priebehu a výsledku spracovania vyzerať odpoveď – SOAP obálka a IFXResponse v nej.

4.4.2 Štruktúra odpovede

Štruktúra SOAP obálky odpovede je nasledovná.

(1) V prípade, že IFXResponse a IFXHeader boli spracované bezchybne, alebo došlo iba k menej závažnej chybe na strane klienta (typ 2) alebo na strane serveru a teda *globálny Status nadobúda vážnosť Info alebo Warn*, je IFXResponse vrátená priamo v SOAP obálke. IFXHeader je štandardné zreplikovaný pred IFXResponse:

```
<env:Envelope>
   <env:Header>
      <wsse:Security>
         <ds:Signature><!--podpis SOAP Body--></ds:Signature>
      </wsse:Security>
   </env:Header>
   <env:Body Id="SOAPBody">
      <ifx:IFXHeader><!--replika--></ifx:IFXHeader>
      <ifx:IFXResponse>...</ifx:IFXResponse>
   </env:Body>
</env:Envelope>
```

SOAP obálka okrem toho môže obsahovať aj podpis tela správy serverom (čomu zodpovedá `Security/Signature` element v SOAP hlavičke).

(2) V prípade, že pri spracovaní došlo k závažnej chybe, či už na strane klienta (typ 1) alebo na strane serveru, sú IFXResponse a pripadne aj zreplikovaný IFXHeader vrátené v rámci SOAP fault – v SOAP `Fault/Detail` elemente. `Fault/Code` obsahuje jednu z hodnôt `sender` alebo `receiver` v závislosti od toho, či ide o chybu klienta alebo serveru.

```
<env:Envelope>
   <env:Body>
      <env:Fault>
         <env:Code>
            <env:Value>env:Sender</env:Value>
         </env:Code>
         <env:Reason>
            <env:Text xml:lang="sk">Chybna požiadavka</env:Text>
         </env:Reason>
         <env:Detail>
            <ifx:IFXHeader><!--replika--></ifx:IFXHeader>
            <ifx:IFXResponse><!--zavazna chyba--></ifx:IFXResponse>
         </env:Detail>
      </env:Fault>
   </env:Body>
</env:Envelope>
```

(3) V prípade, že došlo k SOAP špecifickej chybe, akou je napríklad v našom prípade zlyhanie overenia podpisu v Security hlavičkovom bloku, je vygenerovaný SOAP Fault, ktorý *neobsahuje* IFXResponse vo `Fault/Detail` elemente (ani nikde inde).

4.4.3 Signalizácia odpovedi pomocou naviazania na HTTP

SOAP naviazanie na HTTP (viď kapitola o SOAPe) jednoznačne definuje ako sa horeuvedené prípady pretavia do HTTP response status kódov HTTP odpovedí, ktoré budú prenášať SOAP obálku s odpoveďou. V prípade (1) pôjde o status kód 200 (OK). V druhom prípade pôjde buď o status kód 400 alebo 500. V prípade (3) to bude status kód 400 (alebo 500 v prípade, že overenie podpisu zlyhalo kvôli chybe serveru).
Z kapitoly o SOAPe vieme, že výpočet chyb, ktoré môžu nastať, ešte nie je kompletný. Okrem horespomenutých môže dôjsť ešte k HTTP špecifickej chybe – a v tomto prípade nie je v odpovedi uvedená ani SOAP obálka.

Okrem toho môže dôjsť aj k vygenerovaniu inej SOAP špecifickej chyby z iného dôvodu ako z dôvodu neoverenia podpisu. Uvedomme si totiž, že v praxi nebudeme písať klienta ani server "from scratch", ale veľmi pravdepodobne vygenerujeme časť kódu serveru automaticky, čo nám uľahčí implementáciu. Okrem toho, náš IFX server bude pravdepodobne bežať v rámci nejakého aplikačného serveru, akým je napríklad Apache Axis[60], ktorý sa sám môže starať o časti zodpovednosti IFX serveru (napr. HTTP komunikácia). Keby sme chceli získať úplnú kontrolu nad tým, za akých okolnosti sa môže odoslať SOAP Fault a za akých nie, mali by sme pred sebou značne obtiažnu úlohu. Preto musí byť klient pripravený prijímať a vedieť spracovať a konať na základe aj iných, možno celkom štandardných, chyb, ktoré môže SOAP uzol od iného SOAP uzlu očakávať. V reále by toto nemala byť náročná úloha – klient jednoducho prezrie SOAP Fault a v prípade, že obsahuje IFXResponse, vie ako postupovať a v prípade, že tam IFXResponse nie je, vie že ide o SOAP špecifickú chybu a musí sa zariadiť podľa nej.

4.5 Bezpečnosť

V tejto podkapitole si povieme niečo o bezpečnosti nášho návrhu. Najprv si definujeme bezpečnostné požiadavky, ktoré budeme chcieť dosiahnuť. Následné si stručne popíšeme nástroje na ich dosiahnutie a ako ich použijeme. Podotýkame, že naším cieľom bola v prvom rade aplikácia už existujúcich bezpečnostných nástrojov a riešení pre potreby nášho návrhu a nie ich rigorózny popis a ani návrh nových nástrojov a riešení. Podobne, v tejto kapitole budeme skúmať nástroje na umožnenie bezpečnej *výmeny správ*. Pri použití návrhu pre reálnu implementáciu pravdepodobne pribudnú ďalšie požiadavky, ktoré bude nutné riešiť.
Na nasledovnom diagrame máme načrtnuté, v akom kontexte budeme bezpečnostné požiadavky definovať a takisto sú načrtnuté už aj niektoré riešenia.

[60] Viď http://ws.apache.org/axis2/

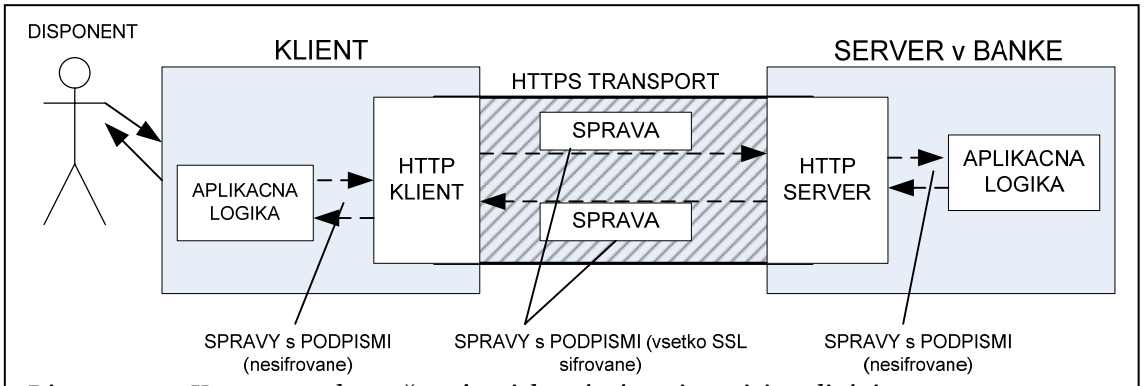


Diagram 24: Kontext pre bezpečnosť a niektoré nástroje na jej realizáciu

Diagram 24: Pri stanovovaní bezpečnostných požiadaviek a návrhu ich riešení, budeme predpokladať, že informácie sa budú prenášať vo forme ucelených správ s definovanou štruktúrou. Máme tu stručne načrtnuté aj niektoré z použitých riešení. Správy sa budú prenášať medzi klientom a serverom pomocou HTTPS (HTTP komunikácia s transport layer security). Správy budú môcť byť digitálne podpísané a podpis priložený ku správe.

4.5.1 Bezpečnostné požiadavky (ciele)

Vzájomná autentifikácia

Banka chce overiť, že osoba, ktorá ovláda klienta a posiela správy je skutočne oprávnená osoba – disponent. Disponent chce overiť, že server je skutočne server banky. K overeniu identity komunikujúcej strany nedochádza priamo, ale tým spôsobom, že táto preukaze, že má alebo vie niečo, čo by mala vedieť len ona – užívateľské meno, heslo, súkromný kľúč atd.

Dôvernosť

Ide o zabezpečenie toho, že prenášané správy môžu (v nezašifrovanej podobe) čítať len oprávnené komunikujúce strany, čo v našom prípade znamená disponent a banka, resp. klient disponenta a server banky. Z hľadiska nášho návrhu sa budeme zaoberať spôsobmi zabezpečenia dôvernosti počas samotnej výmeny správ, nie po jej ukončení.

Autentifikácia správy

Autentifikácia správy[61] je overenie toho, že správu vytvoril a odoslal oprávnený a očakávaný odosielateľ – v našom prípade klient kontrolovaný disponentom na jednej strane a server banky na druhej strane. Ide v podstate o zviazanie identity odosielateľa v obsahom správy.

Integrita

Ide o zabezpečenie toho, že správy neboli počas prenosu komunikačným kanálom medzi autentifikovanými stranami zmenene. K zmene môže dôjsť kvôli chybám pri prenose, pripadne môže ísť o zámernú modifikáciu správy útočníkom. Zabezpečenie integrity úzko súvisí s autentifikáciou správy – v konečnom dôsledku totižto chceme overiť, že prijatá správa bola súčasne (1) prijatá od oprávneného a očakávaného odosielateľa a (2) nebola počas prenosu zmenená.

Nepopretie

Nepopretie je schopnosť jednej strany dokázať pomocou kryptografických a iných metód, že druhá strana niečo vykonala, resp. schopnosť zabrániť popretiu vykonania nejakej akcie

[61] Alebo autentifikácia zdroju dát

druhou stranou. V našom prípade ide o schopnosť dokázať odoslanie správy druhou stranou.
Predpokladajme napríklad, že osoba identifikovaná ako disponent D pošle banke správu, na základe ktorej banka vykoná úhradu prostriedkov na účte disponenta D. Disponent D ale neskôr odmieta, že on zadal príkaz na vykonanie takejto operácie, t.j. že on správu, na základe ktorej bola operácia vykonaná, neposlal. Nepopretie v tomto kontexte znamená, že banka je schopná dokázať, že správu poslal skutočne on.
Z hľadiska kryptografie ide o dokázanie toho, že správa bola poslaná osobou, ktorá vedela/mala niečo čo mala vedieť/mat iba oprávnená osoba – napr. zviazanie odoslanej správy s privátnym kľúčom oprávnenej osoby.
Z praktického (a teda právneho) hľadiska je nepopretie schopnosť súdnou cestou dokázať zodpovednosť druhej strany za vykonanie akcie X. Toto predpokladá existenciu zmluvne dohodnutých a/alebo zákonom daných podmienok, ktoré ak nastanú, tak druhá strana je právne zodpovedná za vykonanie akcie X.
My sa v našom návrhu budeme snažiť poskytnúť kryptografické prostriedky pre nepopretie, rozoberať tento problém z právneho a iných hľadísk je nad rámec tohto návrhu.

Dostupnosť

Ide o zabezpečenie prístupu k službe v očakávanej forme oprávnenými osobami. Presnejšie, ide o zabezpečenia toho, že disponenti budú môcť (prostredníctvom výmeny správ podľa protokolu) využívať služby v „očakávanej“ miere (24 hodín denne všetky operácie apod.)

4.5.2 Nástroje na dosiahnutie bezpečnostných požiadaviek

SSL/HTTPS

SSL (Secure Sockets Layer) je kryptografický protokol umožňujúci bezpečnú komunikáciu medzi bodmi v sieti komunikujúcimi pomocou TCP/IP protokolu. HTTPS je „naviazanie“ HTTP protokolu na SSL protokol – SSL predstavuje transport pre HTTP požiadavky a odpovede.
SSL/HTTPS umožňuje autentifikáciu serveru pomocou SSL certifikátu serveru (certifikát s verejným kľúčom serveru, identitou, názvom domény atd) a zabezpečuje dôvernosť a integritu prenášaných správ pomocou šifrovania a autentifikačných kódov správ[62].
Spôsobom fungovania HTTPS resp. SSL sa nebudeme podrobne zaoberať. Čitateľ nájde viac informácií v [RFC4346]. Podotkneme len, že skutočná bezpečnosť a dosiahnutie deklarovaných bezpečnostných služieb (autentifikácia, dôvernosť, integrita) závisí od toho, ako bude SSL implementovaný v konečnom riešení.

PKI[63]

Infraštruktúra verejného kľúča je založená na kryptografii verejného kľúča[64]. V nej pre každú entitu A existujú dva kľúče, verejný a súkromný. K súkromnému má prístup a môže ho používať iba entita A – jeho vlastník. Verejný kľúč je určený pre všetky entity, ktoré chcú bezpečným spôsobom komunikovať s entitou A. Medzi súkromným a verejným kľúčom existuje matematický vzťah, ktorý, veľmi zjednodušene, hovorí nasledovné: to, čo sa zašifruje súkromným kľúčom, sa dá dešifrovať iba príslušným verejným kľúčom a naopak: to, čo je zašifrované verejným kľúčom, môže byť dešifrované iba príslušným súkromným kľúčom. Na základe tohto môžeme jednoducho dosiahnuť dôvernosť, autentifikáciu (správy, odosielateľa) a poskytnúť prostriedky pre nepopretie.

---

[62] Message Authentication Codes
[63] Public Key Infrastructure, pre prehľad Viď napr. http://en.wikipedia.org/wiki/Public_key_infrastructure
[64] Asymetrickej kryptografii

Dôvernosť: správu zašifrovanú verejným kľúčom môže dešifrovať iba vlastník príslušného súkromného kľúču.
V prípade, že nejaká správa je zašifrovaný súkromným kľúčom a následné ho „úspešne“ dešifrujeme vereným kľúčom, vieme, že pôvodná správa bola odoslaná držiteľom privátneho kľúča, čiže pravdepodobne vlastníkom – takto môžeme vykonať autentifikáciu odosielateľa a správy. Túto (súkromným kľúčom) zašifrovanú správu si môžeme odložiť a neskôr použiť ako dokaz toho, že túto správu poslal držiteľ súkromného kľúča, čo nám poskytuje prostriedok pre nepopretie – vlastník súkromného kľúča by hypoteticky (ak kľúč nebol odcudzený/zneužitý) nemal byť schopný poprieť, že danú správu skutočne poslal on.
Čiže v scenári, kde chce komunikovať N entít, stačí, ak každá entita má svoj súkromný kľúč a drží ho v tajnosti a jej verejný kľúč je distribuovaný ostatným entitám. Otázka je, ako distribuovať kľúče tak, aby si entita X bola istá, že verejný kľúč, ktorý dostala, skutočne patrí deklarovanému vlastníkovi – entite Y. V prípade, že takáto asociácia medzi verejnými kľúčmi a entitami nie je zabezpečená, sú bezpečnostné služby realizované pomocou asymetrickej kryptografie náchylné na viacero útokov[65] a nemožno ich teda považovať za funkčné.
V praxi sa problém asociácie identít s verejnými kľúčmi rieši pomocou tzv. *certifikačných autorít*. Na (okrem iného) jej existencii je založená aj už spomínaná infraštruktúra verejného kľúča (PKI). Certifikačná autorita registruje certifikáty užívateľov (entít). Certifikát obsahuje identitu užívateľa, jeho verejný kľúč a prípadne ďalšie informácie[66]. Tieto informácie sú digitálne podpísané certifikačnou autoritou (jej súkromným kľúčom) a podpis je súčasťou certifikátu. Takto sú identita a verejný kľúč užívateľa bezpečne zviazané. Toto zviazanie sa dá hocikedy overiť pomocou verejného kľúča certifikačnej autority[67].
Certifikačná autorita alebo iná dôveryhodná tretia strana plniaca podobnú funkciu je dôležitá pre dosiahnutie nepopretia. Predpokladajme, že Alica pošle Bobovi správu *S* zašifrovanú svojim súkromným kľúčom a neskôr odoslanie správy popiera. Bob môže teraz predložiť zašifrovanú formu správy, a keďže ju dokáže dešifrovať Aliciným verejným kľúčom, tvrdiť, že správu skutočne poslala Alica. Ta však môže odmietnuť, že daný verejný kľúč (a teda aj súkromný) patri jej. V takomto prípade sa nám zíde certifikačná autorita – skontrolujeme, či v databáze eviduje certifikát, ktorý zväzuje Alicinu identitu s daným verejným (a teda aj súkromným) kľúčom.
V našom návrhu predpokladáme, že banka a disponent poznajú navzájom – a to ešte pred začiatkom komunikácie – svoje verejne kľúče a netreba ich teda distribuovať cez komunikačný kanál.
PKI budeme používať na challenge-reponse autentifikáciu klienta a na digitálne podpisovanie požiadaviek a odpovedi s cieľom dosiahnutia zodpovednosti komunikujúcich strán (nepopretie).

EOK

Elektronický osobný kľúč je zariadenie, ktoré sa bude využívať v proprietárnej časti návrhu. Po jeho nainštalovaní na strane klienta ako aj na strane serveru umožní autentifikáciu klienta pomocou jednorazových hesiel a nepriamym spôsobom umožní odoslane správy autentifikovať pomocou tzv. EOK kódov[68] a vytvorí tak podporu pre nepopretie (ale iba zo strany klienta[69])
Z technologického hľadiska nebudeme EOK rozoberať do hĺbky, len načrtneme, na základe akých princípov funguje.
Fungovanie EOK je založené na symetrickej kryptografii. Klient (resp. disponent) a server zdieľajú tajný kľúč, pomocou ktorého jednak disponent môže generovať jednorazové hesla

[65] Napr. man in the middle attack.
[66] Napr. v prípade SSL certifikátu je to doménová adresa serveru.
[67] Zviazaniu (identita certifikačnej autority, verejný kľúč certifikačnej autority) dôverujeme.
[68] EOK kód je obdoba message authentication kódu (MAC).
[69] Prípad keby klient chcel poprieť nejakú akciu.

a autentifikačné kódy a server ich na druhej strane môže overovať. V prípade, že dochádza aj k autentifikácii serveru, funguje to aj inverzne – server odosiela a klient overuje.
Tajný kľúč je na strane klienta bezpečným spôsobom uložený v tzv. EOK kalkulátore, prístup do ktorého je chránený heslom. Klient po zadaní hesla môže kalkulátor používať na vygenerovanie jednorazového hesla alebo zadávať vstupne údaje a na základe nich generovať autentifikačné kódy pre dane údaje.
Na serveri je nainštalované zariadenie, ktoré si pre každý zaregistrovaný EOK kalkulátor pamätá zdieľaný tajný kľúč a dôkaze overovať autentifikačné kódy a heslá prijate od klienta.
Okrem zdieľaného tajného kľúču[70] sú zariadenia na strane klienta a serveru zosynchronizované časovo a pomocou počítadla. V prípade generovania jednorazového hesla sa aktuálny čas a hodnota počítadla spolu s tajným kľúčom použije na jeho vygenerovanie pomocou nejakého algoritmu. Následne sa inkrementuje počítadlo a vygeneruje nový tajný kľúč. Toto sa deje deterministickým spôsobom tak, aby ten istý kľúč mohol vygenerovať aj server na druhej strane. Generovanie autentifikačného kódu pre dáta je podobné, s tým rozdielom, že vstupom sú aj samotné dáta, ktoré sa majú autentifikovať.
Podotknime ešte, že napriek tomu, že ide o systém založený na symetrickej kryptografii, je možné pomocou neho dosiahnuť službu nepopretia. Server by totiž za normálnych okolnosti nemohol predložiť podpis ako dokaz, keďže kľúč je zdieľaný a mohol by si vygenerovať podpis podľa ľubovôle. Preto je infraštruktúra na strane serveru navrhnutá takým spôsobom, aby toto nebolo možné. Viac informácií o fungovaní EOK je dostupných v [EOK] [71].

4.5.3 Model pre zodpovednosť a nepopretie

Prečo je vlastne nutné zaručiť zodpovednosť strán a zabezpečiť nepopretie?
Predpokladajme, že disponent pošle banke požiadavku, v ktorej žiada o vykonanie prevodu peňazí. Ak banka tento prevod vykoná, obe strany sú spokojné. Ak ale banka prevod nevykoná, toto bude disponentovi pravdepodobne vadiť a bude sa s bankou sporiť. Tento a iné spory, ktoré môžu vzniknúť medzi bankou a disponentom, je nutné vedieť vyriešiť, resp. aspoň poskytnúť prostriedky na ich vyriešenie.

V našom návrhu disponent posiela IFX požiadavky, v ktorých sú uvedené *individuálne požiadavky*. Na základe nich banka vykonáva pasívne alebo aktívne operácie na účte disponenta.
Je potrebné zabezpečiť aby disponent bol zodpovedný za všetky požiadavky, ktoré pošle banke a na základe ktorých ona následné koná a naopak, je potrebné zabezpečiť, aby banka konala práve tak ako jej to ukladajú požiadavky od disponenta.
V našom návrhu sa pokúsime dosiahnuť tieto ciele prostredníctvom digitálnych podpisov[72]. Bude možné, alebo dokonca povinné, individuálne požiadavky a odpovede digitálne podpisovať. Schéma je nasledovná:

[70] Tých kľúčov je dokonca niekoľko – pre každú operáciu (autentifikácia, generovanie autentifikačných kódov) sa používa samostatný kľúč.
[71] Referencia obsahuje najmä používateľské informácie – ako správne použiť EOK kalkulátor na strane klienta. Informácie však nie sú dostupne bez zakúpenia príslušného zariadenia.
O infraštruktúre na strane serveru sú informácie obmedzene podobne.
[72] A EOK kódov, o nich neskôr.

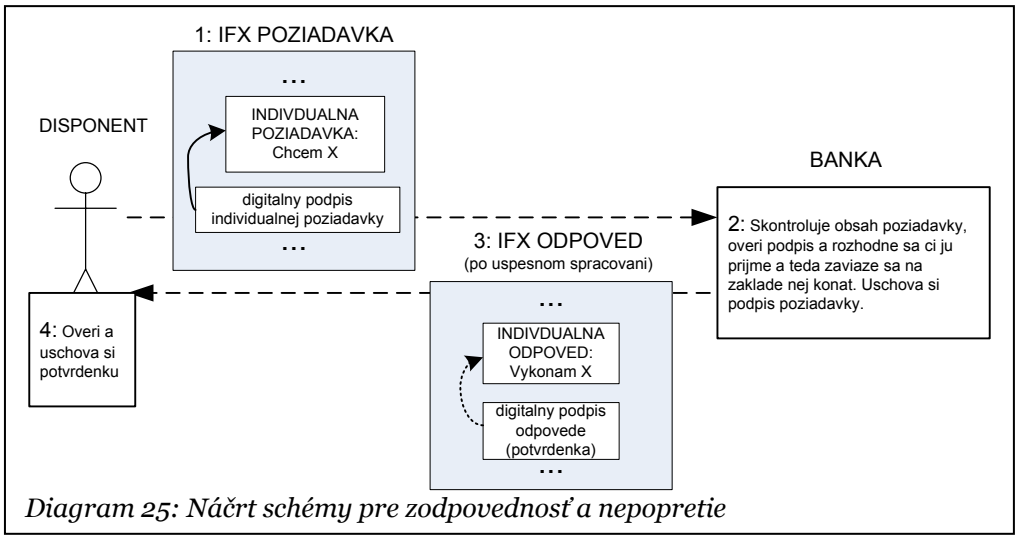


Diagram 25: Náčrt schémy pre zodpovednosť a nepopretie

Predpokladajme, že disponent chce vykonať nejaké nejakú operáciu nad účtom. Vygeneruje preto príslušnú (individuálnu) požiadavku, ku každej priloží jej podpis a odošle ju banke. Banka (individuálnu) požiadavku spracuje a v prípade, že požiadavka je v poriadku a aj podpis sedí, ju môže prijať. Podpis požiadavky si odloží ako dokaz o tom, že disponent takúto požiadavku poslal (a chcel vykonať príslušnú operáciu). O pozitívnom prijatí požiadavky informuje klienta v odpovedi, ku ktorej priloží aj digitálny podpis tejto odpovede[73]. Táto bude pre zmenu slúžiť ako potvrdenka pre disponenta o tom, že banka požiadavku prijala a zaviazala sa na jej základe konať. Môže sa stať, že banka potvrdenku v odpovedi nepošle, alebo bude potvrdenka chybná (podpis sa neoverí). Toto si ale disponent hneď všimne a môže banke tento fakt hneď reklamovať.
Podotýkame, že pre banku je na vykonanie príslušnej operácie postačujúca správne zadaná požiadavka so správnym podpisom – t.j. bez ohľadu na to, či disponent obdrží správnu potvrdenku, môže byť operácia vykonaná.

Takýmto spôsobom má banka pre neskoršie spory k dispozícii potvrdenie od klienta a naopak, disponent bude mat potvrdenku od banky, že požiadavku prijala a zaviazala sa ju vykonať. Na efektívne riešenie sporov sa banka a disponent ešte musia dopredu dohodnúť, *čo presne je postačujúce podpísať*, aby sa na základe podpísaného mohol spor jednoznačné vyriešiť. Predpokladajme, že na tomto sa banka a disponent dokážu dohodnúť a správili tak.
Aké spory môžu vzniknúť a ako ich môžeme pomocou navrhnutých nástrojov riešiť? Intuitívne, keď disponent podpíše všetko, na základe čoho sa môže vykonať aktívna operácia na účte a banka pošle potvrdenku o všetkom, čo sa zaviaže na základe tohto vykonať, všetko „aktívne“, čo sa na účte udeje, je zdokladovateľné jednou alebo druhou stranou.

Konkrétne, disponent môže banku obviniť, že 1. nejakú požadovanú operáciu nevykonala, 2. vykonala nejakú operáciu ktorú disponent nepožadoval, 3. „zle“ vykonala nejakú operáciu[74].
V prvom prípade bude musieť disponent predložiť potvrdenku od banky (príslušná odpoveď a jej podpis), kde sa banka zaväzuje konať. V prípade, že takéto dôkazy predloží, je banka za nekonanie zodpovedná. V opačnom prípade disponent nemá ako dokázať zodpovednosť banky.

[73] Je otázka či podpisovať aj negatívne odpovede: banka môže klienta informovať, že požiadavka je nesprávna, ale aj tak ju realizovať. Náš návrh to (podpísanie) bude umožňovať, ale v praxi toto ale nie je pravdepodobný scenár.
[74] Napr. sa prevedie 1000SK namiesto 5000Sk.

V prípadoch 2 a 3 disponent banku obviní, že banka vykonala operáciu, ktorú disponent nepožadoval (prípad 2), alebo vykoná požadovanú operáciu „zlým“ spôsobom. Banka v tomto prípade bude musieť predložiť podpisy od disponenta, ktoré ju k týmto operáciám oprávňujú. Ak to nedokáže, je zodpovedná. Ináč zodpovedná nie je.
Pre kompletnosť si uvedomme, že banka nemá dôvod *iniciovať* spor s klientom, keďže všetky operácie vykonáva ona – všetko čo sa udeje má pod kontrolou. Banka iba reaguje na obvinenia klienta.

4.5.4 Možnosti podpísania správ v našom návrhu

My v našom návrhu predpokladáme nasledovné možnosti podpísania správy.

Inline podpis

Prvá možnosť je podpísanie celej *individuálnej* požiadavky. Podpis je vložený ako element Signature do elementu danej požiadavky – ide teda o zaobalený typ XML podpisu.
V prípade inline podpisu musia byť dáta, ktoré sú v požiadavke uvedené a podpísané, postačujúce na riešenie sporov. V našom návrhu bude možnosť podpisovať takýmto spôsobom všetky individuálne požiadavky, ktoré môžu mať za následok vykonanie nejakej aktívnej operácie alebo inej akcie zo strany banky. Je samozrejme možné podpisovať aj individuálne požiadavky pre pasívne operácie aj keď pre toto momentálne nevidíme zmysel.
Príklad tohto typu podpisu je uvedený v úvodnom príklade Návrhu.

Globálny podpis

Druhá možnosť je podpísať správu ako celok, presnejšie podpísať celé SOAP Body so všetkými individuálnymi požiadavkami. V tomto prípade je podpis uložený v rámci WS Security hlavičkového bloku. Banka a disponent sa musia dohodnúť, že predloženie celého SOAP Body a jeho podpisu je postačujúce pre riešenie sporov. V tele totižto môže byť veľa informácií, ktoré budú pre spor nezaujímavé. Globálny podpis vyzerá elegantne, ale má svoje implikácie napr. na dávkové spracovanie. Štandardné sú individuálne požiadavky spracovane v rámci IFX požiadavky nezávislé – zlyhanie overenia podpisu jednej individuálnej požiadavky neovplyvni spracovanie inej (nezávislej) individuálnej požiadavky. V prípade jedného globálneho podpisu jeho zlyhanie znamená nespracovanie ani jednej individuálnej požiadavky.

Súčasne použitie globálneho a inline podpisov

Podotýkame, že tieto dva druhy podpisov je možné kombinovať, t.j. podpísať jednotlivé požiadavky a potom podpísať ešte celé správu ako celok alebo naopak. V tomto prípade sa treba zamyslieť, či skôr vytvoríme globálny podpis a až potom podpíšeme jednotlivé požiadavky alebo naopak. V prípade, že toto chceme robiť, treba sa zamyslieť, aké to bude mat implikácie na vytváranie a overovanie globálneho podpisu – globálne budeme chcieť pravdepodobne podpísať obsah SOAP Body *bez* jednotlivých inline podpisov a pri overovaní ich najprv transformáciou z tela odstrániť. [75]

Čo podpisovať?

Na tom, čo presne by mala obsahovať individuálna požiadavka (okrem samotného obsahu), aby jej inline podpis bol postačujúci ako dôkazový materiál pre riešenie sporov, sa musia v konečnom dôsledku dohodnúť banka a disponent. Skúsme sa zamyslieť, čo by mohlo byť postačujúce.

Požiadavka:
1) RqUID
2) Vlastné parametre požiadavky, ktoré jednoznačne definujú, čo sa má vykonať.

[75] Koniec koncov nám však nič nebráni globálne podpísať aj samotné inline podpisy.

3) *Dátum a čas odoslania (resp. vygenerovania) požiadavky klientom s dostatočnou presnosťou (minimálne sekundy[76]).*

Čiže, stačí pridať dátum a čas klienta, lebo RqUID a parametre musí požiadavka obsahovať aj tak. Keď sa takáto požiadavka podpíše, RqUID zaručuje globálnu jednoznačnosť požiadavky (aj podpisu), jej obsah jednoznačne definuje požadovanú akciu a takisto je presne dané, kedy ju klient poslal.

Odpoveď:

1) *Stav vybavenia požiadavky (Status agregát)*
2) Replika RqUID
3) Replika všetkých parametrov z požiadavky definujúcich čo sa má vykonať.
4) Replika dátumu a času odoslania požiadavky klientom
5) *Dátum a čas akceptácie požiadavky serverom*

V odpovedi máme teda zreplikovanú celú požiadavku (okrem samotného podpisu), okrem toho tu máme Status agregát, ktorý oznamuje, že banka požiadavku prijala a zaviazala sa na jej základe konať[77] a takisto je tu dátum a čas akceptácie požiadavky serverom. Takáto požiadavka spolu s podpisom by mohla byť postačujúca.

Zodpovednosť a nepopretie pomocou EOK

EOK tak, ako je momentálne v našej konkrétnej banke používaný, umožňuje zaručiť iba zodpovednosť disponenta.
EOK totiž momentálne umožňuje, aby autentifikačné kódy posielal len klient a aj to iba pri zadávaní príkazu alebo pri jeho dodatočnom podpisovaní[78]. Neumožňuje podpisovať zrušenie príkazu ani odvolanie podpisu[79].
Stále ale platí, že pomocou EOK je možné zabezpečiť nepopretie klientom. Procesy na strane serveru sú totiž navrhnuté tak, aby to aj s obmedzenými možnosťami EOK umožnili. Odvolávanie podpisu nie je nutne podpisovať vôbec a zrušenie príkazu je realizované tak, že sa vygeneruje novy príkaz, ktorý odkazuje na pôvodný príkaz a tento „príkaz na zrušenie príkazu“ je podpísaný. Následné môže byť zrušený pôvodný príkaz. Takto je zabezpečené, že banka má potvrdenie o tom, že disponent chcel príkaz odvolať.

EOK neumožňuje zaručenie zodpovednosti serveru, keďže server EOK autentifikačné kódy posielať nevie. Jedna možnosť ako pridať zodpovednosť banky je čiastočné použitie PKI. Certifikát a verejný a súkromný kľúč by v tomto prípade mala iba banka a mohla by posielať potvrdenky v podstate rovnakým spôsobom ako je to popísané v našom modeli. V modeli totižto nie je predpokladom generovania a odosielania potvrdeniek banky spôsob, ako sú požiadavky podpísané (EOK, súkromný kľúč, ...). Takto celkom lacným spôsobom môžeme (pre proprietárny návrh) rozšíriť model o zodpovednosť banky pričom overené firemne procesy pre EOK zostanú zachovane.

4.5.5 Realizácia bezpečnostných požiadaviek

Z doteraz napísaného je jasné, ako sa budeme snažiť dosiahnuť nepopretie (PKI a EOK), ako bude zaručená dôvernosť a integrita správ a autentifikácia serveru (HTTPS).
O challenge-response autentifikácii klienta pomocou PKI a autentifikácii pomocou jednorazového EOK hesla budeme hovoriť v poslednej kapitole tejto práce. Ostáva nám povedať niečo o zaručení dostupnosti.

---

[76] A možno presnejšie, ak napríklad na základe vygenerovaného času budeme chcieť požiadavky klienta jednoznačne a tak ich vybavovat a pripadne sa neskôr sporiť sa aj o presne poradie. Toto ale v našom návrhu neriešime.

[77] V pripade, ze doslo pri spracovani k chybe, nema vyznam odpoved podpisovat, kedze ziadna operacia sa vykonavat nebude. V pripade, ze banke nedoverujeme natolko, ze ju podozrievame, ze nas sice informuje o chybe, ale prislusnu operaciu aj tak v skutocnosti vykona, bolo by potrebne podpisovat aj negativne odpovede. My vsak toto predpokladat nebudeme.

[78] O tomto si viac povieme v poslednej podkapitole tejto práce.

[79] Tieto sú tiež popísané v poslednej podkapitole tejto práce.

Dostupnosť

Zabezpečenie dostupnosti sa nášho návrhu (výmena správ) týka len okrajovo, preto ho spomenieme len stručne. Jeden aspekt dostupnosti, resp. jedna trieda útokov na narušenie dostupnosti, ktorá sa nášho návrhu týka, sú Denial of service útoky. Zjednodušene ide o to, že útočníci zabezpečia odosielanie veľkého množstva packetov, prípadne kompletných správ na náš server. V prípade nesúrodých packetov je potrebné zabezpečiť ich elimináciu už na úrovni routeru prípadne firewallu – ak to dokážu. Toto je ale tiež mimo rámca nášho návrhu.
Predpokladajme teraz, že náš server je zaplavený správami, ktoré vyzerajú na prvý pohľad celkom „rozumne“ (správne štruktúrované IFX správy) a tak dorazia na náš HTTP server. Zhrňme si idey, ako túto situáciu riešiť:

- Pri začiatku komunikácie požadovať triviálnu HTTP autentifikáciu pomocou mena a hesla, ktoré sú pridelene oprávneným užívateľom. Neautentifikovane správy ignorujeme. V prípade že začneme dostavať veľa správ autentifikovaných jedným heslom (a možno s rôznymi IP adresami), môžeme tohto užívateľa zablokovať.
- Stanoviť maximum pre veľkosť správ a ich frekvenciu od jedného užívateľa.

IFX klient ako nedôveryhodný modul

Spomeňme ešte jeden bezpečnostný problém, značne odlišný od tých, ktoré sme riešili doteraz.
IFX klient môže na základe verejného interfejsu implementovať hocikto. Nepredpokladáme ale, že by si každý užívateľ, pripadne organizácia, chceli implementovať vlastného klienta. V tom prípade musia použiť klienta (alebo softvérový modul) vytvoreného niekým iným. Tento modul môže poskytnúť banka, alebo to môže byť modul implementovaný treťou stranou. V prípade klienta tretej strany budeme chcieť záruku od tejto strany, že za to, čo sa vykomunikuje zle z dôvodu (zámerne) zlej funkčnosti klienta, bude niesť dodávateľ zodpovednosť. Druhá možnosť je certifikácia klientov bankou s tým, že zodpovednosť za prípadne následky by niesla banka. Predpokladajme ale nezaručeného IFX klienta tretej strany[80]. Použitie takéhoto klienta má podstatne implikácie pre bezpečnosť komunikácie samotného disponenta s bankou, pretože nedôveryhodný IFX klient je prostredníkom v komunikácii[81].
Skúsme sa len veľmi stručne zamyslieť, čo „zlé“ sa môže stať a ako by sa vzniknutá zloba dala riešiť. Schematicky si situáciu môžeme znázorniť nasledovne:

[80] Toto v praxi môže byt reálny use case.
[81] Vyvstáva možno otázka, prečo by niekto chcel používať softvér, ktorému nie celkom dôveruje. V praxi ale sotva existuje softvér, ktorému niekto dôveruje na 100%. Ide skôr o to, mat v prípade, že softvér správi niečo, čo by nemal, niekoho, kto za to bude zodpovedný a odškodní ma.

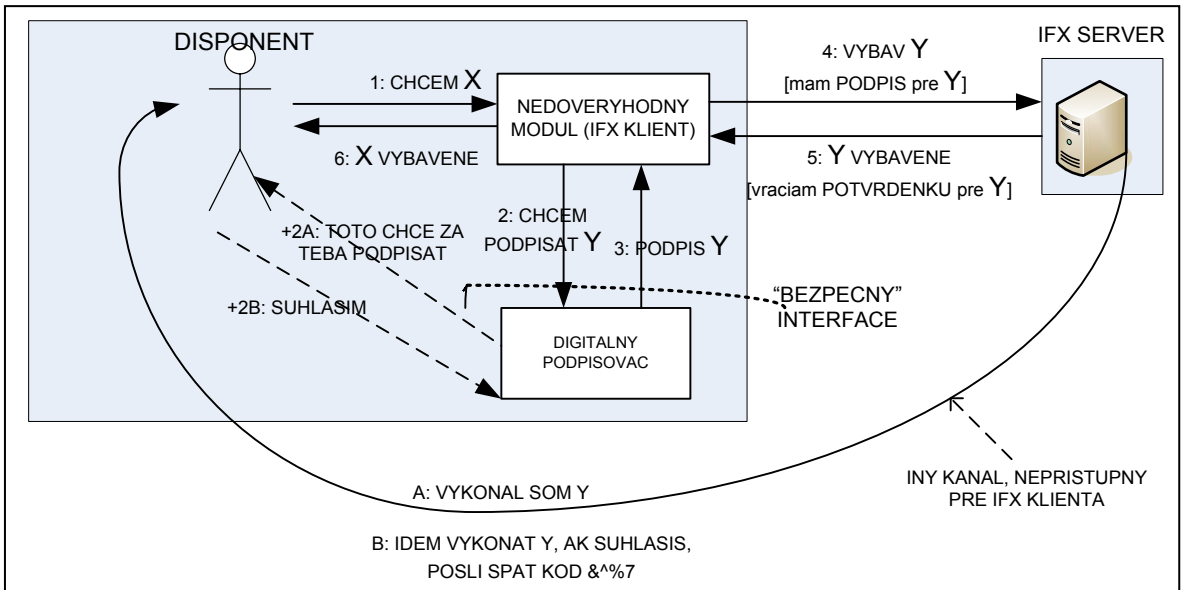


Diagram 26: IFX klient ako nedôveryhodný modul

„Podpisovač" z diagramu je softvérový alebo hardvérový modul, ktorý dokáže podpisovať dáta súkromným kľúčom disponenta. Môže to byť ale aj iné analogické zariadenie ako napr. EOK. Idey pre iné nástroje sú podobné, ako tie, ktoré spomenieme teraz pre digitálny podpis.

V prípade, že komunikácia ide disponent – IFX klient – podpisovač – IFX klient – server – IFX klient – disponent, môže IFX klient vykomunikovať zo serverom čo chce a súčasne disponentovi oznamovať, že všetko je OK. Ako toto riešiť? Jedna možnosť je nezávislým kanálom posielať potvrdenky (digitálne podpísané dáta) o všetkom, čo banka vykonala (možnosť A cez nezávislý kanál). Tieto dáta by sa overili s tými zadanými cez klienta a ak by nesedeli, disponent by mohol promptne reklamovať túto skutočnosť. Ďalšia možnosť (B cez nezávislý kanál) je vykonávať všetky aktívne operácie v dvoch krokoch, s tým, že v prvej odpovedi pošle server disponentovi nezávislým kanálom informácie o tom, čo chce vykonať spolu s nejakým potvrdzovacím kódom, ktorý musí klient v druhom kroku poslať banke (môže aj cez IFX klienta) – iba v prípade že ho táto dostane, vykoná požadovanú operáciu. IFX klient nemá ako zneužiť tieto informácie, nemá k nim prístup, jediné čo môže urobiť, je neposlať kód zadaný disponentom a v tom prípade sa žiadna operácia nevykoná.

Tretia možnosť je, aby disponent schvaľoval všetko, čo podpisuje podpisovač (kroky +2A a +2B). V tomto prípade ide komunikácia nasledovne: disponent – IFX klient – podpisovač – disponent – podpisovač – IFX klient – server – IFX klient – disponent. Podpisovač napríklad zobrazí disponentovi dáta, o ktorých podpísanie ho žiada IFX klient a iba v prípade, že tieto dáta sú správne (zhodujú sa s tými, čo zadal disponent klientovi), disponent dovolí tieto dáta podpísať a odovzdať IFX klientovi.

Na záver treba povedať, že výskyt nedôveryhodného softvéru na strane disponenta je problém, ktorý sa nedá hneď a elegantne vyriešiť. Nebrali sme totižto do úvahy, aké iné akcie okrem miskomunikácie môže zlý klient na počítači disponenta iniciovať. Každé z ideovo načrtnutých riešení okrem toho (mierne) znižuje automatizáciu (ktorá je cieľom) a komplikuje samotnú implementáciu a použitie IFX+WS riešenia.

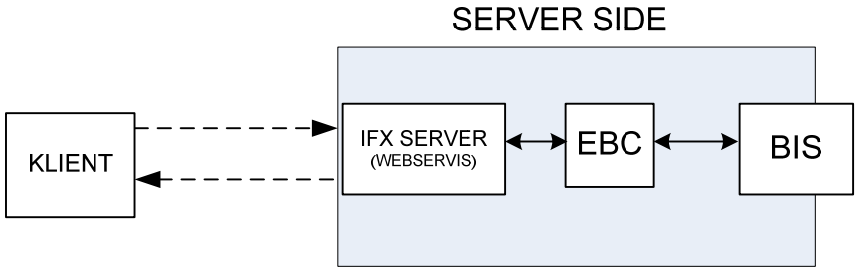
4.6 Operácie a procesy pre webservis elektronického bankovníctva

V tejto kapitole si povieme niečo o operáciách, ktoré budeme chcieť prostredníctvom webservisu vykonávať a príslušných správach, ktoré sa za týmto cieľom budú vymieňať.

Kvôli nedostatku miesta tu nebudeme do detailu popisovať obsah jednotlivých správ a agregátov v nich. Ich presná definícia je uvedená v XML schémach priložených k tejto práci a ich podrobná dokumentácia sa nachádza v ďalšej prílohe tejto práce[82]. Čitateľovi odporúčame konzultovať tieto dve prílohy počas čítania časti tejto kapitoly, pretože tu sa snažíme byť struční na úkor presného popisu.

Ako sme už spomínali, náš návrh sa skladá z dvoch častí, spoločnej a proprietárnej. Proprietárna časť oproti všeobecnej špecifikuje niekoľko málo dodatočných správ a agregátov, ktoré sa môžu posielať medzi klientom a serverom a ktoré podporujú špecifické procesy pre daný banku, resp. pre jej systém elektronického bankovníctva.

Pre potreby podrobnejšieho popisu backend procesov v jednej konkrétnej banke ešte načrtneme architektúru, aká existuje v tejto banke. Informačný systém podporujúci fungovanie elektronického bankovníctva sa skladá z dvoch častí (vrstiev). Sú nimi Jadro elektronického bankovníctva (EBC[83]) a Bankový informačný systém (BIS).

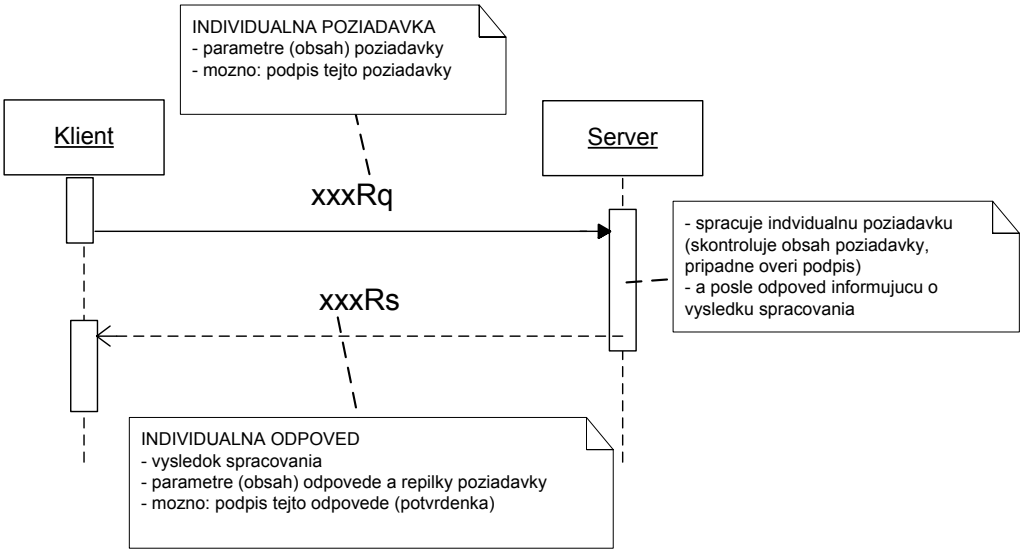


EBC – jadro elektronického bankovníctva slúži pre rôzne formy elektronického bankovníctva (Internetbanking, Homebanking, Phonebanking...). Je to vrstva, ktorá sa stará o interakciu s klientom (autentifikácia, autorizácia príkazov, udržiavanie ich stavu atd) a tak predpripravuje všetky náležitosti pre BIS.
BIS – bankový informačný systém, sú ním prepojené všetky pobočky banky a čo je pre náš podstatné, pomocou neho, resp. v ňom, sa defacto vykonajú významné bankové operácie (prevody, platby atd). EBC prostredníctvom neho ako nižšej vrstvy realizuje jednotlivé operácie.

Teraz už môžeme pristúpiť k popisu individuálnych správ – požiadaviek a odpovedí. Najprv si popíšeme správy týkajúce sa autentifikácie (SignonRq a SignonRs). Potom popíšeme jednotlivé pary správ týkajúce samotnej realizácie bankových operácii. Vieme, že v IFX je každá správa zaradená do nejakej služby (service) a individuálne požiadavky a odpovede z nejakej služby sa vymieňajú v rámci servisných požiadaviek a odpovedí. Všetky správy okrem SvcAcctInqRq a SvcAcctInqRs sú zo služby Banking. Ostatné dve spomenuté patria do služby Base (základné služby), ktorá obsahuje správy spoločné pre rôzne druhy finančných interakcii, nielen pre bankovníctvo. Toto zadelenie definuje IFX a pre nás neexistuje dôvod ho meniť – ide len o formalitu.

Pre lepšie zostručnenie a lepšie pochopenie nasledovného textu si zopakujme, že individuálne požiadavky sa odosielajú a spracovávajú v rámci IFXRequest a po spracovaní sa príslušná individuálna odpoveď pošle späť v rámci IFXResponse.

[82] Viď kapitola Prílohy.
[83] Electronic banking core



Individuálna požiadavka sa skladá v podstate dvoch separátnych časti, jedna je jej samotný obsah a druhá (a nepovinná) je inline podpis tejto požiadavky. Server požiadavku spracuje a prípadne overí podpis a zodpovedajúcu odpoveď následne pošle klientovi a informuje ho takto o výsledku.

4.6.1 Autentifikácia – SignonRq & SignonRs

Podporujeme dva spôsoby autentifikácie klienta. Jedným je challenge-response autentifikácia založená na PKI a druhým je autentifikácia klienta pomocou jednorazového EOK hesla.
SignonRq obsahuje nasledovné informácie (vyberáme len tie najpodstatnejšie):

- Identifikátor užívateľa
- EOK jednorazové heslo alebo žiadosť o dáta na podpis (1. krok challenge response autentifikácie) alebo podpísané dáta (2. krok challenge response autentifikácie) - v závislosti od typu prihlásenia
- Žiadosť o vygenerovanie session kľúča – v prípade úspešnej autentifikácie jedným z dvoch horeuvedených mechanizmov pošle server v odpovedi Session kľúč, ktorý užívateľ môže (určitý čas) používať na prihlasovanie.

SignonRs obsahuje nasledovné informácie (vyberáme len tie najpodstatnejšie):

- Status agregát – či autentifikácia a prihlásenie prebehlo úspešne
- Náhodné dáta na podpis pre klienta – ak sme v prvom kroku challenge response autentifikácie[84]
- Session kľúč, v prípade úspešnej autentifikácie a žiadosti užívateľa

Pozrime sa teraz dva možné typy prihlásenia, ktoré náš návrh umožňuje.

Prihlásenie pomocou EOK autentifikačného kódu

Ide o jednoduchý request-response, v požiadavke pošle klient EOK heslo a v odpovedi obdrží odpoveď o výsledku prihlásenia.

Challenge response autentifikácia pomocou PKI

Vykonáva sa v dvoch krokoch. V prvom kroku klient požiada o zaslanie náhodných dát[85] na podpis. Tieto mu server pošle. V druhom kroku klient tieto dáta podpíše svojim

[84] v tomto prípade Status nehovorí o výsledku autentifikácie ale o výsledku vygenerovania náhodných dať

súkromným kľúčom a odošle ich v rámci požiadavky na prihlásenie. Server overí správnosť podpisu a v prípade úspechu je klient úspešné autentifikovaný a pravdepodobne je vrátený aj session kľúč.
Jednoduchým rozšírením tohto modelu je možné dosiahnuť aj autentifikáciu serveru[86].

Na nasledovných stranách si popíšeme správy, ktoré budú umožňovať pracovať s príkazmi – umožnia ich zadávať, rušiť, dodatočne ich podpisovať a tieto podpisy odvolávať.

4.6.2 Zadanie príkazu – OrderAddRq & OrderAddRs

Pomocou OrderAddRq klient zadáva bankové príkazy (tuzemský prevod, zahraničný prevod apod.), teda žiada o vykonanie nejakej aktívnej operácie nad účtom. Spôsob zadania je rovnaký pre všetky typy príkazov – OrderAddRq.
OrderAddRq obsahuje nasledovne informácie (vyberáme len tie podstatné):

- RqUID
- Definícia príkazu (príslušný agregát definujúci tuzemský, zahraničný alebo hromadný prevod)
- dátum a čas (datetime) odoslania požiadavky a inline podpis požiadavky (ale iba v prípade, že chceme požiadavku podpísať alebo to server požaduje) alebo EOK autentifikačný kód pre daný príkaz (autentifikujú sa dáta definujúce príkaz, nie celá požiadavka)

Server požiadavku štandardným spôsobom spracuje a pošle späť odpoveď. V prípade úspešného spracovania je príkaz bankou prijatý.
OrderAddRs obsahuje nasledovne informácie (uvádzame len tie podstatné):

- Status agregát – výsledok spracovania požiadavky: odmietnutie alebo prijatie príkazu.
- replika RqUID, definície príkazu a klientovho datetime (ak bol v požiadavke uvedený)
- v prípade úspešného spracovania, t.j. v prípade prijatia príkazu serverom, je uvedený záznam o tomto príkaze (OrderRecord), ktorý sa po jeho prijatí vytvorí a server pomocou neho môže ďalej sledovať[87] ako sa príkaz v systéme spracováva
- prípadne datetime prijatia požiadavky serverom a potvrdenka serveru (inline podpis tejto odpovede)

4.6.3 Dodatočné podpísanie príkazu

Niekedy bude chcieť disponent podpísať príkaz dodatočne. Toto môže byť jednoducho z dôvodu, že to banka dovoľuje alebo požaduje (najprv sa príkaz zadá pomocou OrderAddRq bez podpisu a až dodatočne sa podpíše), Druhý dôvod je použitie EOK, pri použití ktorého sa spravidla podpisujú zadané príkazy dodatočne. Tretia možnosť je tá, že banka požaduje, aby príkaz bol podpísaný *niekoľkými* disponentmi predtým ako ho banka vybaví – k tomuto sa ešte dostaneme. Popíšme si ale teraz už samotný spôsob dodatočného podpisovania príkazov. Vykonáva sa v dvoch krokoch. V prvom si klient najprv od serveru vyžiada „dáta na podpis“. Tieto následné podpíše a podpis pošle serveru. Server overí či je podpis správny. Ak je správny, dodatočné podpísanie prebehlo úspešne, ináč je klient informovaný o chybe.

---

[85] Ako majú náhodné dáta vyzerať aby ich podpísanie umožňovalo bezpečnú autentifikáciu, teraz neriešime. Predpokladáme ale, že také dáta sa dajú určíte zvoliť.
[86] Klient v požiadavke druhého kroku pošle spolu so svojim podpisom náhodných dať aj (iné a nové) náhodné dáta, ktoré má tentoraz podpísať server – ten tieto dáta pošle v odpovedi druhého kroku podpísané (toto ale zrejme urobí iba v prípade úspešnej autentifikácie klienta)
[87] Pomocou OrderInqRq, o ktorom budeme hovoriť neskôr.

Krok 1: Vyžiadanie dát na dodatočné podpísanie – OrderAuthInfoInqRq & OrderAuthInfoInqRs

Klient v požiadavke identifikuje príkaz[88], pre ktorý žiada informácie o autorizácii a uvedie nastroj pre aký žiada dáta na podpis – pre EOK a PKI sú totižto tie dáta úplné odlišné. V (úspešnej) odpovedi server pošle okrem jednak informácie o tom, ktorí disponenti už daný príkaz podpísali a takisto pošle „dáta na podpis“, ktoré disponent môže použiť druhom kroku na dodatočne podpísanie príkazu. Je zrejme rozumne očakávať, že tieto dáta sa týkajú daného príkazu a nejakým spôsobom ho jednoznačne definujú alebo identifikujú. V triviálnom prípade týmito dátami môže byť napríklad kompletný OrderAddRq daného príkazu tak ako bol odoslaný pôvodným zadávateľom (ak bol odoslaný s podpisom, tak ten samorejme teraz nebude).

Krok 2: Dodatočné podpísanie príkazu – OrderAuthAddRq

Klient príkaz podpíše (požiada server o „pridanie autorizácie“ – Order Authorization Add Request). Klient najprv pre dané dáta vygeneruje podpis / autentifikačný kód a pošle ho serveru. Ten overí, či je podpis správny. Na základe toho klienta informuje či pridanie podpisu bolo úspešné.
OrderAuthAddRq obsahuje nasledovné informácie:

- RqUID, bankové ID príkazu
- V prípade EOK dodatočne: autentifikačný kód pre daný príkaz vygenerovaný na základe dát získaných v predošlom kroku (krok 1)
- V prípade inline podpisu: „dáta na podpis“ z kroku 1, datetime odoslania požiadavky klientom a samotný podpis celej požiadavky.

Server požiadavku štandardným spôsobom spracuje a pošle späť odpoveď. V prípade úspešného spracovania je príkaz dodatočne podpísaný disponentom – dodatočný podpis je prijatý.
OrderAuthAddRs obsahuje nasledovné informácie:

- Status agregát – výsledok spracovania požiadavky: odmietnutie alebo prijatie dodatočného podpisu.
- replika RqUID, bankového ID príkazu definície príkazu a klientovho datetime (ak bol v požiadavke uvedený)
- v prípade, že server posiela potvrdenku pre túto odpoveď, odpoveď obsahuje aj datetime prijatia požiadavky serverom, repliku dát na podpis, a samotnú potvrdenku (inline podpis tejto odpovede)

4.6.4 Autorizácia príkazu

Vieme, že pomocou OrderAddRq môžeme príkaz (Order) zadať do systému. Po úspešnom zadaní je príkaz bankou prijatý, ale ešte nemusí byť vykonaný. Banka môže požadovať, aby bol tento príkaz najprv podpísaný buď samotným zadávateľom a to buď súčasne so zadaním (inline podpis OrderAddRq) alebo dodatočne (OrderAuthAddRq). Niekedy banka môže požadovať, aby príkazy boli najprv podpísané *niekoľkými* disponentmi a az potom pristúpi k ich realizácii.
Vo všeobecnosti, banka môže požadovať, aby bol príkaz najprv *autorizovaný* a až potom pristúpi k jeho vykonaniu. Autorizáciu môžeme chápať jednak ako *schválenie príkazu niekoľkými disponentmi* (1 a viac) a na druhej strane môžeme autorizáciu chápať ako *splnenie dopredu definovanej podmienky typu*: Pre daný príkaz P s danými parametrami A každý disponent Di z nejakej množiny disponentov D podpísal príkaz P(A) bezpečnostným nástrojom X. Ak takáto podmienka nastane, budeme hovoriť že príkaz bol autorizovaný (disponentami).

[88] Vyvstáva otázka ako sa klient dozvedel ID daného príkazu. Odpoveď: pomocou OrderInq správ. Tieto sú zdokumentované neskôr v tejto kapitole.

Takéto autorizačné podmienky môžu byť dopredu určené pre rôzne typy príkazov a ich parametre. Môže byť napríklad dohodnuté, že všetky tuzemské prevody viac ako 100 tisíc Sk musí podpísať aspoň polovica disponentov daného účtu.
Disponenti a banka sú dopredu zmluvne dohodnutí, že banka sa pokúsi o realizáciu príkazu P(A) práve vtedy, ak sú splnene autorizačné podmienky.
Autorizácia sa používa hlavne kvôli zdieľaniu účtov a bezpečnosti. Bezpečnostný aspekt je zjavný – je potrebné aby príkaz podpísalo niekoľko disponentov. Druhá výhoda je možnosť pre disponentov dopredu definovaným spôsobom zdieľať účet a vykonávať operácie na ňom. Predstavme si, že máme firmu, ktorá je riadená 5 partnermi a má v banke otvorený účet. Všetci piati partneri sú disponentmi na tomto učte. Po vzájomnej dohode s bankou si teraz môžu napríklad stanoviť pravidlá typu: Na každú úhradu nad 100 tisíc Sk musí byť táto autorizovaná aspoň troma disponentmi a platby nad 1 milión Sk musia autorizovať všetci disponenti.
Prípadne môžu byť na účte vytvorené tzv. „podpisové skupiny“ s niekoľkými disponentmi, ktoré môžu udávať rámec pre autorizáciu niektorých typov príkazov.

4.6.5 Odvolanie podpisu – OrderAuthCanRq & OrderAuthCanRs

Umožňuje disponentovi zrušiť *svoj* podpis daného príkazu (odvolať svoj podpis).
Klient pošle požiadavku, kde identifikuje príkaz, pre ktorý chce podpis odvolať. Naspäť dostane odpoveď či odvolanie podpisu prebehlo úspešne. Server môže, ale nemusí požadovať inline podpis[89] tejto požiadavky.
Pre naše proprietárne procesy to nie je nutne. Životný cyklus príkazu je taký, že v momente keď sa nazbiera dosť podpisov, začína banka príkaz realizovať a už nie je možné podpisy odvolávať. V prípade, že ešte neexistuje dosť podpisov a teda príkaz ešte nie je autorizovaný, je podpisy možné odvolávať (a pridávať) bez potvrdeniek, preto banka realizáciou príkazu začne zaoberať až v momente keď je príkaz autorizovaný (podpísaný dostatočným počtom disponentov). Viac o tomto si povieme pri popise životného cyklu príkazu pre proprietárne procesy v nasej banke.

4.6.6 Zrušenie (odvolanie) príkazu

Umožňuje disponentovi odvolať príkaz, ktorý je zadaný v informačnom systéme banky.
Klient pošle požiadavku, kde identifikuje príkaz, ktorý chce odvolať. Naspäť dostane odpoveď o výsledku svojej požiadavky. Server môže, ale nemusí, požadovať inline podpis tejto požiadavky.
Vieme, že pomocou EOK sa nedá odvolanie príkazu priamo podpísať – dokonca vieme, že EOK dokáže podpísať len pridanie príkazu (OrderAddRq) alebo dodatočne podpísať zadaný príkaz (OrderAuthAddRq). Keďže odvolanie príkazu je dôležitá aktívna operácia, aj v prípade našich procesov server bude vyžadovať podpis od klienta. Ako je to realizované? Po žiadosti disponenta o odvolanie príkazu sa vygeneruje nový „príkaz na odvolanie príkazu“. Tento je potom možné dodatočne podpísať pomocou EOK, čím server získa požadované potvrdenie od klienta. Niekedy nie je nutné rušenie príkazu podpisovať ani takýmto spôsobom. Totižto, pokiaľ nie je príkaz autorizovaný disponentmi, je ho možné zrušiť aj bez podpísania príkazu o odvolanie príkazu. Tomu, za akých podmienok môže byť príkaz odvolaný, za ktorých nie a akým spôsobom odvolávanie prebieha, sa budeme venovať v nasledovnom odstavmi.

4.6.7 Životný cyklus príkazu

Tu si popíšeme, aký je životný cyklus jednoduchého[90] príkazu pre backend procesy v jednej banke. Životným cyklom príkazu rozumieme množinu stavov, v ktorých sa príkaz môže nachádzať, spolu s pravidlámi na prechod medzi týmito stavmi.

---

[89] Pomocou EOK sa odvolanie nedá podpísať.
[90] Nie „hromadného“ príkazu, o ňom neskôr.

Pozrime sa najprv na množinu možných stavov a prechody medzi nimi.

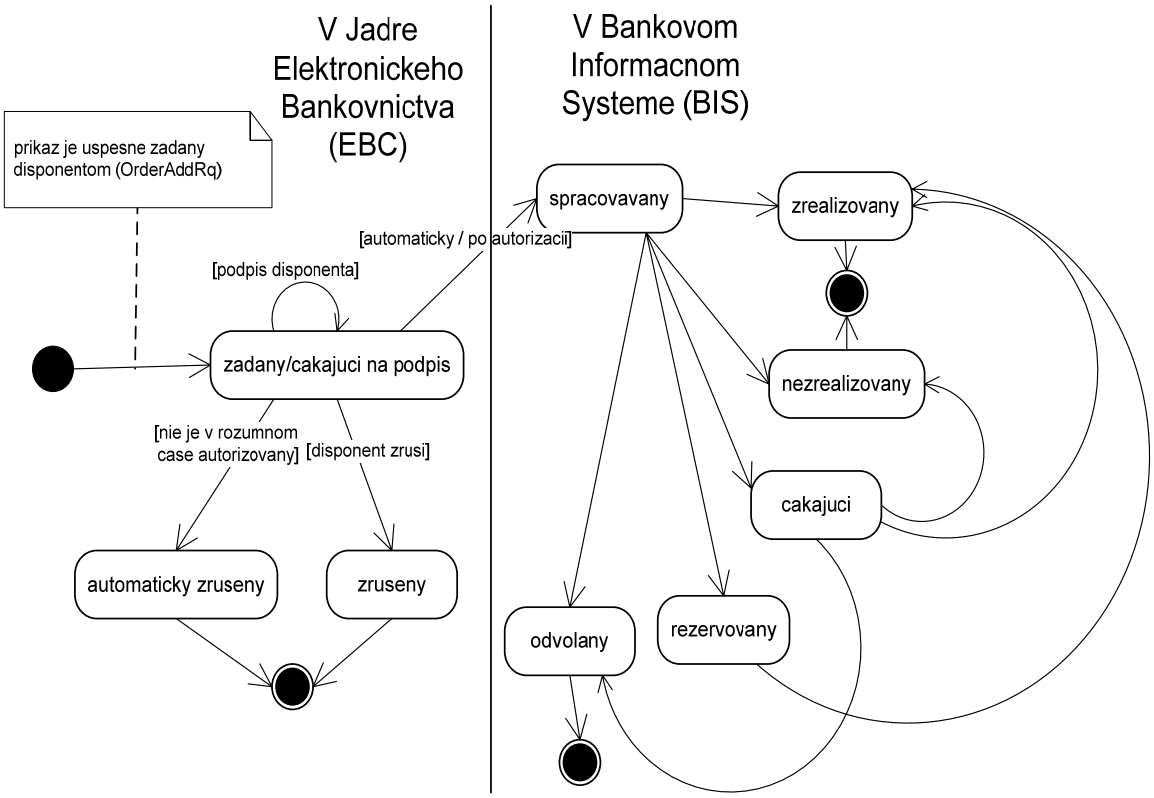


Diagram 27: Stavový diagram životný cyklus príkazu

Príkaz vznikne po úspešnom zadaní disponentom v rámci EBC – vytvorí sa objekt (Order Record) reprezentujúci tento príkaz. V momente po zadaní je v stave **zadaný/čakajúci na podpis**. Môže nastať jedna z nasledujúcich možností.

1) V prípade, že príkaz nie je nutné autorizovať, hneď odchádza na spracovanie do BIS, čomu zodpovedá prechod do stavu *spracovávaný*.

V prípade, že je nutné ho autorizovať, príkaz zotrváva v stave zadaný/čakajúci na podpis až do momentu, keď nastane jedna z nasledujúcich možnosti:

2) Je potrebným spôsobom autorizovaný disponentmi. V tomto prípade odchádza do BIS na spracovanie, t.j. prejde do stavu *spracovávaný*.
3) Ak nie je v „rozumnom čase“[91] autorizovaný, je príkaz systémom zrušený. Príkaz prejde do stavu *automaticky zrušený*. Toto je konečný stav.
4) Disponent požiada o jeho zrušenie. V prípade úspechu prechádza do stavu zrušený. Toto je konečný stav. Je na EBC, či povolí zrušiť príkaz v tomto stave, ak je už niekým (iným) podpísaný, alebo požaduje aby boli všetky podpisy najprv odvolané.

Predpokladajme teraz, že príkaz je v stave **spracovávaný** a odišiel teda do BIS na spracovanie. Môžu nastať nasledovné možnosti:

1) Príkaz je bankou zrealizovaný a prejde do konečného stavu *zrealizovaný*.
2) Príkaz nie je zrealizovaný a prejde do stavu *nezrealizovaný*[92].

[91] Napríklad úhradu je zrejme potrebne autorizovať pred dátumom splatnosti

[92] Toto sa môže stať napríklad v prípade, že disponent zadal príkaz, ktorý bol chybný, ale pri zadávaní sa tento fakt ešte nedal zistiť – napr. pri úhrade to môže byt neexistujúce číslo protiúčtu, alebo nedostatok prostriedkov na účte.

3) Príkaz prejde do stavu *čakajúci*. Toto sa deje napríklad v prípade, že príkaz bol zadaný s budúcim dátumom realizácie (splatnosti) alebo čaká na inú udalosť, ktorá musí nastať predtým, ako sa príkaz môže začať realizovať.
4) Prejde do stavy *rezervovaný*. Toto je špeciálny stav, z ktorého po čase príkaz určíte prejde do stavu *zrealizovaný*. Prechádzajú nim niektoré typy príkazov.
5) Príkaz prejde do stavu *odvolaný*. Toto sa udeje v prípade, že sa ho disponentovi podarí úspešné odvolať.

Stav **čakajúci** a jeho prechody sú podobné stavu *spracovávaný*. V tomto stave príkaz čaká dokým nie je splnená nejaká podmienka alebo nenastane nejaká udalosť, a až potom sa pristúpi k jeho realizácii bankou. Príkaz môže napríklad čakať na (svoj) dátum realizácie (splatnosti).
Čakajúci príkaz môže prejsť do niektorého z nasledujúcich stavov:
1) *zrealizovaný* – v prípade úspešnej realizácie.
2) *nezrealizovaný* – v prípade, že sa ho nepodarilo zrealizovať. Dôvody nezrealizovania sú rovnaké ako pre príkaz v stave spracovávaný
3) *odvolaný* – ak sa ho disponentovi podarí odvolať.

Niektoré typy príkazov sa môžu počas spracovania dostať do stavu **rezervovaný**.
Z hľadiska nášho návrhu je podstatné vedieť iba toľko, že po nejakom čase určite prejde do stavu *zrealizovaný*.

Stavy **automaticky zrušený**, **zrušený**, **odvolaný**, **zrealizovaný** a **nezrealizovaný** sú konečné stavy.

Pozrime sa teraz na životný cyklus príkazu z iného pohľadu, pre náš návrh zaujímavejšieho. Budeme skúmať, aké operácie (pridávanie/odvolávanie podpisov, zrušenie príkazu) môžeme s príkazmi v jednotlivých stavoch vykonávať.

Zrušenie / odvolanie príkazu
Predpokladajme, že disponent sa pokúsi príkaz zrušiť – pošle teda požiadavku na zrušenie daného príkazu. Môže nastať niekoľko situácií v závislosti od toho v akom stave daný príkaz je:
1) Ak je v niektorom z *konečných* stavov alebo v stave *rezervovaný*, táto požiadavka je požadovaná za chybnú a o tomto fakte bude klient informovaný
2) Ak je v stave *zadaný/čakajúci na podpis*, tak v prípade, že nie je nikým podpísaný, je zrušený hneď. Ak niekým podpísaný je, bude je tiež hneď zrušený, alebo je požiadavka považovaná za chybnú – toto závisí od toho, či server povoľuje rušiť príkazy, ktoré sú už niekým podpísané, alebo požaduje, aby pred zrušením mal príkaz nula podpisov.
3) Ak je v stave *spracovávaný* alebo *čakajúci*, príkaz už odišiel na spracovanie do BIS a nie je možné ho zrušiť hneď. V tomto prípade je v EBC vytvorený nový „príkaz na odvolanie príkazu“[93]. Tento je najprv nutné autorizovať za rovnakých podmienok ako pôvodný príkaz. Potom prejde príkaz do BIS (stav spracovávaný) a BIS rozhodne, či bude pôvodný príkaz zrušený. Až do momentu, kým príkaz na zrušenie príkazu prejde do stavu spracovávaný, nie je spracovanie pôvodného príkazu príkazom na odvolanie vôbec ovplyvnené. Kľudne sa môže stať, že pôvodný príkaz sa medzitým zrealizuje. Príkaz na odvolanie sa potom zrejme nemôže úspešne zrealizovať a prejde do stavu nezrealizovaný.

Pridávanie/odvolávanie podpisov
Podpis príkazu môže vykonať disponent práve vtedy, keď je príkaz v stave *zadaný/čakajúci na podpis*. V prípade, že príkaz môže byť podpísaný týmto disponentom

[93] Tento prechádza rovnakým životným cyklom ako každý iný príkaz.

a podpis je správny, je podpis prijatý. V prípade, že tomu tak nie je, alebo je príkaz v inom stave, je požiadavka chybná a klient je o tom informovaný.

Nakoniec si uvedomme, že bez ohľadu na to, aký životný cyklus príkazov si tá-ktorá banka definuje, navrhnutá množina správ a možnosť autorizácie všetkých správ, ktoré majú ľubovoľný „aktívny dopad" na procesy na strane serveru/banky, je postačujúca na podporu zvoleného životného cyklu a daných procesov. Toto by samozrejme neplatilo, keby banke/disponentovi nestačila pre realizáciu ich interakcií tá množina správ (t.j. operacií), ktorú sme navrhli. Tomu sa nedá predísť, ale predpokladáme, že pridaním požadovaných správ/operácii by sa dal návrh ľahko a konzistentne rozšíriť.

4.6.8 Typy príkazov v návrhu

Náš návrh momentálne explicitne podporuje štyri typy príkazov: tuzemský prevodný príkaz, zahraničný prevodný príkaz, hromadný prevodný príkaz a príkaz na zrušenie príkazu. Ďalšie typy príkazov je ľahko možné dodefinovať – stačí vytvoriť príslušný agregát definujúci nový typ príkazu a pripadne agregát, pomocou ktorého ho budeme chcieť zadávať v OrderAddRq. Agregáty pre jednotlivé typy príkazov si tu nebudeme podrobne popisovať – sú značne obsiahle a okrem definovania samotného obsahu nie veľmi zaujímavé. Pokúsime sa ich len stručne popísať.
Pre každý príkaz existuje agregát, pomocou ktorého ho zadávame (v rámci OrderAddRq): DomXferInfo pre tuzemský prevod, ForXferInfo pre zahraničný prevod a MassXferInfo pre hromadný prevod[94].
Po zadaní príkazu sa na strane serveru vytvorí záznam o tomto príkaze. Existuje spoločný agregát (OrderRec), ktorý obsahuje spoločné informácie pre všetky typy príkazov a okrem toho pre každý jednotlivý typ príkazu obsahuje agregát, ktorý daný typ príkazu dodatočne definuje (DomXferRec – tuzemský prevod, ForXferRec – zahraničný prevod, MassXferRec – hromadný prevod, OrderCancelRec – príkaz na zrušenie príkazu)
OrderRec obsahuje informácie spoločné pre všetky agregáty, akými sú napríklad: účet disponenta, ktorému tento príkaz prislúcha; Id príkazu pridelene bankou[95]; stav príkazu, typ príkazu, dátum zadania príkazu atd.

Tuzemský prevodný príkaz[96] – DomXferInfo, DomXferRec

Ide o prevod peňazí medzi účtom disponenta (príkazcu) a účtom príjemcu, pričom účet príjemcu je tiež vedený v slovenskej banke. Úhrady medzi slovenskými bankami vykonáva Národná Banka Slovenska, ktorá (prostredníctvom vyhlášok a opatrení) aj presne určuje, aké náležitosti a parametre musí zadanie úhrady mať a ako sa v konečnom dôsledku peniaze z jedného účtu na druhý prevedú. Takisto sú určené formát čísla účtu pre vnútrozemský platobný styk (predčíslie účtu, číslo účtu, kód banky) a ďalšie atribúty špecifikujúce prevod: konštantný, špecifický, variabilný symbol.
XferInfo teda obsahuje tieto položky a niektoré ďalšie. Detaily nájde čitateľ v prílohách k tejto práci.
K XferRec iba podotknime, že môže obsahovať referenciu (Id príkazu) na hromadný príkaz v prípade, že tento tuzemský prevod bol zadaný ako súčasť hromadného príkazu. O tomto si povieme viac pri popise hromadného príkazu.

Zahraničný prevodný príkaz[97] – ForXferInfo, ForXferRec

Ide o prevod peňazí medzi účtom disponenta (príkazcu) a účtom príjemcu, pričom účet príjemcu je vedený v *zahraničnej* banke. Úhrady sú v tomto prípade vykonávané

---

[94] Príkaz na zrušenie príkazu neuvádzame, lebo ten sa nezadáva explicitne, ale vygeneruje sa automaticky po žiadosti klienta o zrušenie príkazu.
[95] Toto nie je RqUID a ani s ním nijako nesúvisí.
[96] Ekvivalentne názvy: tuzemská úhrada, tuzemská platba
[97] Alebo ekvivalentne: zahraničná platba, zahraničná úhrada.

prostredníctvom systému SWIFT[98], ktorý definuje formát a pravidlá na vykonávanie takýchto úhrad. Položky pre zahraničný prevodný príkaz obsahujú okrem položiek definovaných SWIFTom aj niektoré položky definované Národnou Bankou Slovenska, ktorá pri týchto prevodoch asistuje. Nakoniec ešte môžu/musia byť uvedené položky, ktoré sú špecifické pre slovenskú banka–zákazník komunikáciu. Obsah je presne definovaný v Schéme a zdokumentovaný v príslušnej prílohe tejto práce.

Hromadný prevodný príkaz (MassXferInfo, MassXferRec) a proces jeho spracovania

Hromadný prevodný príkaz je špecifický typ príkazu, ktorý umožňuje zadať niekoľko tuzemských prevodných príkazov naraz a podpísať ich (tiež naraz).
Zadanie je definované v MassXferInfo, ktoré obsahuje niekoľko (N; N>0) DomXferInfo agregátov definujúcich N jednoduchých tuzemských prevodov, pričom účet platcu (disponenta) musí byť pre všetky prevody rovnaký.
Skúsme si podrobnejšie popísať ako je hromadný príkaz zadávaný a spracovávaný v našom proprietárnom návrhu[99].

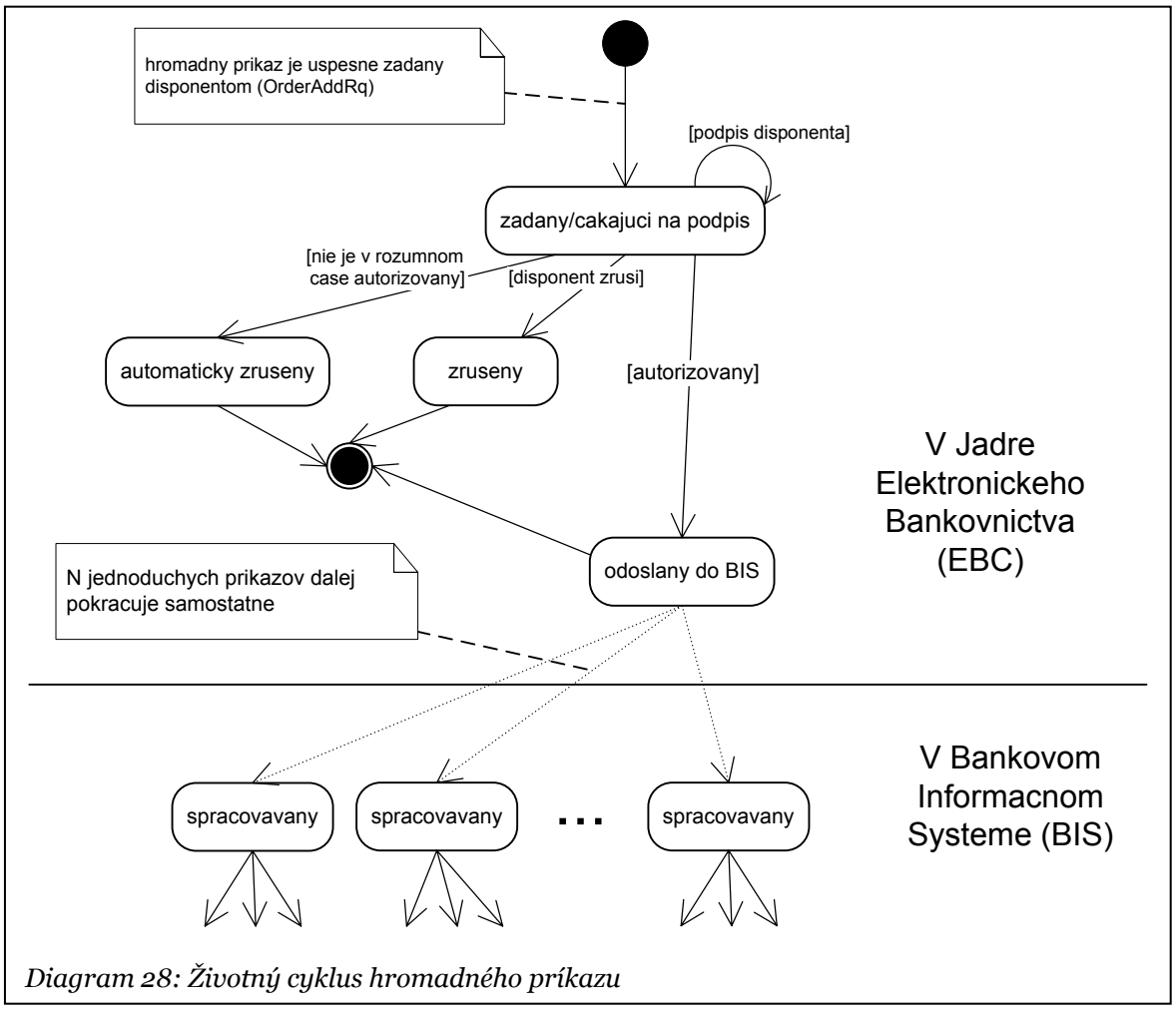


Diagram 28: Životný cyklus hromadného príkazu

Hromadný príkaz (HP) funguje v skratke nasledovne. Je zadaný, následne disponentmi podpísaný (autorizovaný), v momente autorizácie sa „rozpadne“ na N samostatných prevodných príkazov, ktoré sú v BIS spracovávané samostatne. Hlavná výhoda

[98] Society for Worldwide Interbank Financial Telecommunication, http://www.swift.com/
[99] Z toho, čo sa nám podarilo vypozorovať z procesov iných bánk: tento proces sa všade dosť podobá.

hromadného príkazu s N položkami v porovnaní s ekvivalentnými N jednotlivými príkazmi je zjednodušenie podpisovania. Na podpísanie hromadného príkazu nám stačí jeden podpis (bez ohľadu na počet položiek). Na N samostatných príkazov by sme potrebovali N samostatných podpisov. Toto je v praxi výhodné najmä keď podpísanie jedného príkazu je drahé, napr. trvá nezanedbateľný čas, alebo vyžaduje nezanedbateľnú ľudskú pracú. Takto je to napríklad v prípade EOK, ktorý na generovanie jednorazových autentifikačných kódov vyžaduje zadávanie dát človekom (disponentom).
Životný cyklus hromadného príkazu a príslušných N jednotlivých platieb je nasledovný (viď Diagram 29). Hromadný príkaz na strane EBC „funguje“ rovnako ako „normálny“ príkaz – najprv je úspešné zadaný, po čom sa ocitá v stave *zadaný/čakajúci na podpis*. V tomto stave mu môžu disponenti dodatočne pridávať/odoberať podpisy štandardným spôsobom, alebo sa ho pokúsiť zrušiť. Rozdiel nastáva, keď HP dostane dostatok podpisov a je teda autorizovaný. V tomto momente by normálny príkaz odišiel do BIS na spracovanie a zároveň by prešiel do stavu *spracovávaný*. HP ale v tomto momente namiesto toho konci svoj životný cyklus a prejde do konečného stavu *odoslaný do BIS*. V tomto momente sa zároveň „rozpadne“ na N jednotlivých príkazov, ktoré idú na spracovanie do BIS a teda ocitajú sa v stave *spracovávaný*. Tieto ďalej fungujú autonómne, ako keby boli zadané samostatne a môžu byť aj samostatne odvolávané[100]. EBC si ale pamätá, že tieto príkazy boli zadané v rámci HP spolu, aby disponenti mohli sledovať, ako sú jednotlivé príkazy v rámci HP spracovávané. Čo sa dialo s jednotlivými príkazmi v rámci HP do momentu, keď ich HP „porodil“? Už v momente zadania HP vzniknú v EBC okrem HP aj jednotlivé príkazy. Tieto sú ale až do momentu úspešného zániku HP úplné pasívne. Kopírujú stav zastrešujúceho hromadného príkazu a disponenti s nimi nemôžu vykonávať žiadne operácie, jednotlivo ich podpisovať, ani pridávať alebo odoberať z HP. Operácie sa dajú vykonávať iba nad zastrešujúcim HP. Disponent si jednotlivé príkazy môže iba prezerať (v rámci HP) v prehľade príkazov.

Príkaz na odvolanie príkazu – CancelOrderRec

Tento príkaz môže vzniknúť po tom, čo sa disponent pokúsi odvolať nejaký príkaz zadaný na strane serveru, ktorý ale nie je možné odvolať hneď – spravidla je nutné najprv takéto odvolanie podpísať. Vytvorí sa teda príkaz na zrušenie príkazu, ktorý obsahuje iba referenciu na príkaz, ktorý sa má zrušiť. Tento je následné príslušným spôsobom podpísaný a BIS sa potom pokúsi pôvodný príkaz odvolať.

4.6.9 Prehľady príkazov, obratov a účtov

Doteraz sme rozoberali najmä aktívne operácie: zadávanie, rušenie a podpisovanie príkazov. Rovnako dôležité sú ale aj pasívne operácie, ktoré nám umožňujú zisťovať informácie o jednotlivých účtoch, príkazoch a obratoch. Všetky fungujú veľmi podobným spôsobom. V požiadavke klient zadá filtračné kritéria a na ich základe mu server vráti príslušné záznamy – buď záznamy o účtoch disponenta, záznamy o príkazoch alebo záznamy o obratoch vykonaných nad účtom. Keďže vo všetkých prípadoch ide o jednoduchú požiadavku typu Inquiry, pričom filtračných kritérií je veľa, náš popis bude veľmi stručný. Čitateľ môže v prípade záujmu konzultovať príslušné prílohy tejto práce.

Prehľad účtov – SvcAcctInqRq, SvcAcctInqRs & BankAcctRec

Umožňuje disponentovi zistiť zoznam účtov, nad ktorými môže vykonávať operácie a podrobné informácie o jednotlivých účtoch – napr. rôzne zostatky na účtoch apod. Server v tomto prípade vracia disponentovi BankAcctRec agregáty (záznamy o bankovom učte).

[100] Podmienky na odvolanie sú rovnaké ako keby bol zadaný prevod zadaný sám.

Prehľad príkazov – OrderInqRq, OrderInqRs & OrderRec

Umožňuje disponentovi zistiť zoznam príkazov zadaných na strane serveru a kompletné záznamy (OrderRec), ktoré o nich vedie banka. Disponent môže požiadať o zobrazenie príkazov vyhovujúcich nejakým filtračným kritériám. Týmito sú napríklad stav príkazu, typ príkazu, dátum zadania, množstvo prostriedkov apod.

Prehľad obratov – AcctTrnInqRq, AcctTrnInqRs & AcctTrnRec

Umožňuje disponentovi získať informácie o obratoch na niektorom (jeho) účte. Server vráti klientovi zoznam AcctTrnRec agregátov. Opäť môžeme použiť veľké množstvo filtračných kritérií: dátum obratu, suma, typ obratu, či ide o debet alebo kredit apod.

5 Prílohy

K tejto práci sú priložené nasledovné prílohy:

1) `IFX_ws.wsdl` – WSDL dokument pre náš webservis.

2) Kompletná XML schéma definujúca správy, agregáty a elementy pre náš webservis:
   a) `IFX_schema.xsd` obsahuje definíciu základných dátových typov IFX, spoločné elementy a agregáty z IFX a nami definované elementy a agregáty špecifické pre slovenský platobný styk (všetky s prefixom SVK).
   b) `IFXschemaSLSP.xsd` obsahuje definíciu elementov a agregátov pre proprietárne použitie v jednej konkrétnej banke. Tieto sú definované vo vlastnom XML namespace.
   c) `IFX_BankService.xsd` obsahuje definície správ (požiadaviek a odpovedí) pre Bank Service.
   d) `IFX_BaseService.xsd` obsahuje definície správ (požiadaviek a odpovedí) pre Base Service.
   e) `IFXws_Protocol.xsd` obsahuje definície IFXRequest, IFXResponse a IFXHeader a definície servisných správ (Base a Bank Service Request a Response)
   f) `xmldsig-core-schema.xsd` – prikladáme aj definíciu XML Signature, keďže ju používame v našich správach a agregátoch.

3) `message_set.html` – obsahuje podrobnú dokumentáciu všetkých agregátov a správ používaných v našom návrhu.

6 Záver

V tejto práci (a jej prílohách) sme navrhli webservis pre prístup k elektronickému bankovníctvu, jednak všeobecne pre slovenský platobný styk, ale aj konkrétne pre potreby jednej banky. Skúsme teraz zhrnúť naše pozorovania a skúsenosti nadobudnuté počas vytvárania tejto práce a prezentovať nejaké odporučenia pre budúce možné rozšírenia tohto návrhu a pre jeho prípadnú implementáciu.

Štruktúra a obsah správ

Štruktúra vymieňaných správ, ktorú poskytuje IFX, sa ukázala ako postačujúca, dokonca vyhovujúca, pre naše potreby. Po obsahovej stránke sa však množina existujucich IFX správ ukázala ako – pre naše potreby – nedostatočná.
Najprv sme očakávali, že zmeny/rozšírenia, ktoré budeme musieť navrhnúť, budú len kozmetického charakteru. Opak bol však pravdou. Podstatná väčšina individuálnych správ používaných v našom návrhu je nová – reflektujúca pravidlá pre platobný styk a procesy a interakcie v rámci slovenského retailového bankovníctva. „Vnútro“ vymieňaných správ je ale v čo najväčšej možnej miere založené na existujúcich agregátoch a elementoch IFX, často aj na úkor jednoduchosti – toto reflektuje náš pôvodný cieľ o vykonanie len minimálnych zmien.
Pre budúcnosť ale odporúčame zvážiť, či je výhodnejšie snažiť a o čo najväčšiu (rozumnú) kompatibilitu „vnútra“ správ s IFX, alebo využívať iba tie elementy a agregáty, ktoré nám skutočne vyhovujú a naše správy príliš nekomplikujú. Dá sa polemizovať, že štruktúru aj tak budú čítať len programy a teda nie je podstatná. Faktom ale zostáva, že 1. na základe komplikovaného návrhu môže byť menší záujem implementovať a 2. implementácia komplikovaného návrhu môže mat väčšiu náchylnosť na chyby. Toto môže znížiť atraktívnosť produktu implementovaného na základe tohto návrhu a fakt, že „sme konformní a konzistentní“ sa tak stáva irelevantným. Podobne argumenty platia aj z hľadiska akceptácie a použitia tohto návrhu ostatnými bankami.

Úprava IFX pre webservisy

Čo sa týka uprav IFX pre použitie v rámci webservisov, vykonali sme len úpravy, ktoré boli nutne. Pre ďalší vývoj tohto návrhu bude zaujímavé zverejnenie namapovania IFX na webservisy samotnou IFX WS Working group. Tieto zmeny budú pravdepodobne o dosť výraznejšie ako tie naše. Aj kvôli tomuto sme sa snažili o minimálne úpravy – aby sme nezačali zbytočne divergovať.
Takisto komunikačný protokol je v podstate identicky s tým v IFX až na to, že niektoré prvky (napr. asynchrónne odpovede) vynechávame. Procesný model sme sa snažili navrhnúť čo najprirodzenejším zladením ideí webservisov (najmä SOAP procesného modelu) s ideami uvedenými v [IFXBMS] a [IFXHTTP]. Potrebu ďalšieho rozšírenia neočakávame – minimálne do momentu keď vyjde už spomínané IFX->WS namapovanie.

Použiteľnosť pre ostatné banky

Čo sa týka použiteľnosti nášho návrhu pre ostatné banky, podľa našich pozorovaní budú možno potrebné nove agregáty pre rôzne typy bezpečnostných nástrojov. Tieto (nástroje) sú však podobné buď EOK alebo PKI a teda rozšírenie by malo byť priamočiare. Takisto sa domnievame, že náš jednoduchý „model pre zodpovednosť nepopretie“ je dostatočne všeobecný pre použitie ľubovoľného nastroja založeného na autentifikačných kódoch alebo digitálnych podpisoch. Pre to, či v konečnom dôsledku bude náš návrh zaujímavý pre ostatné banky, nerozhodujú len technologické faktory, ale najmä ekonomické[101].

[101] Napr: budú chcieť iné banky použiť návrh, s ktorým prišla iná banka a tak vlastne uznať ich riešenie za „hodné nasledovania”?

Smerom k implementácii

Pred prípadnou implementáciou odporúčame zamyslieť sa nad úpravou obsahu niektorých správ a agregátoch – tak ako sme spomínali vyššie.
Takisto sa treba ešte zamyslieť, ako skonkrétniť a spodrobniť bezpečnostný model – najmä: *čo* bude postačujúce podpisovať a *kedy* bude nutné správy podpisovať. Takisto sa treba zamyslieť, či použiť inline podpisy, globálny podpis, alebo dokonca ich kombináciu.
Popis prvkov komunikačného protokolu a procesu spracovania bude nutné pred alebo počas implementácie ešte spodobniť – niektoré prvky bolo pre nás buď zbytočné prácne rozoberať, alebo ich nebolo možné rozoberať vôbec, keďže závisia od spôsobu zvolenej implementácie.
V prípade snahy o implementáciu na základe open source technologii odporúčame Apache AXIS2, ktorý nedávno vyšiel vo verzii 1.0.

Bibliografia

[AXIS2] – *Apache Axis2*, web services framework, domovská stránka projektu. Viď <http://ws.apache.org/axis2/>.

[CRYPT] – Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone: *Handbook of Applied Cryptography*, CRC Press 1996. Dostupné na internete: <http://www.cacr.math.uwaterloo.ca/hac/>

[EOK] – ActivCard, Inc.: *ActivCard One Reference manual*, nie je verejne prístupné bez zakúpenia produktu. Domovská stránka ActivCard, Inc.: <http://www.activcard.com/>.

[IFX] – *Interactive Financial eXchange Forum, Inc.*, domovská stránka organizácie. Viď <http://www.ifxforum.org/>.

[IFXBMS] – IFX Forum, Inc.: *IFX Business Message Specification, version 1.7*. Dostupné na internete: <http://www.ifxforum.org/standards/standard/>.

[IFXHTTP] – IFX Forum, Inc.: *IFX XML Implementation Specification, Version 1.0.1*. Dostupné na internete: <http://www.ifxforum.org/standards/standard/>.

[IFXWS] – John Hogan, SearchWebServices.com: *Banking industry cashes in on Web services*, 16 Feb 2004. Dostupné na internete: <http://searchwebservices.techtarget.com/qna/0,289202,sid26_gci950653,00.html>.

[IFXWS2] – Colleen Frye, SearchWebServices.com: *IFX Forum expected to unveil Web services work*, 19 Jul 2005. Dostupné na internete: <http://searchwebservices.techtarget.com/originalContent/0,289142,sid26_gci1108743,00.html>.

[OFX] – OFX Consortium: *Open Financial Exchange Specification, version 2.0.2*. Dostupné na internete: <http://www.ofx.net/ofx/de_spec.asp>.

[SLSPDB] – Slovenská Sporiteľňa, a.s.: *Databanking*, produkt a implementačná referencia. Dostupné na internete: <http://www.slsp.sk/index.cfm?module=ActiveWeb&page=WebPage&s=firemne_databanking&newLanguageID=sk> a <http://developer.databanking.sk/>.

[SOAP12pt0] – The World Wide Web Consortium: *SOAP Version 1.2 Part 0: Primer, W3C Recommendation*, 24 Jun 2003. Dostupné na internete: <http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>.

[SOAP12pt1] – The World Wide Web Consortium: *SOAP Version 1.2 Part 1: Messaging Framework, W3C Recommendation*, 24 June 2003. Dostupné na internete: < http://www.w3.org/TR/2003/REC-soap12-part1-20030624/>.

[SOAP12pt2] – The World Wide Web Consortium: *SOAP Version 1.2 Part 2: Adjuncts, W3C Recommendation*, 24 Jun 2003. Dostupné na internete: <http://www.w3.org/TR/2003/REC-soap12-part2-20030624/>.

[RFC4346] – The Internet Society: *The Transport Layer Security (TLS) Protocol, Version 1.1*, 2006. Dostupné na internete: <http://tools.ietf.org/html/4346>.

[XML] – The World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation*, 4 Feb 2004. Dostupné na internete: <http://www.w3.org/TR/2004/REC-xml-20040204 >.

[XMLInfoset] – The World Wide Web Consortium: *XML Information Set (Second Edition), W3C Recommendation*, 4 Feb 2004. Dostupné na internete: <http://www.w3.org/TR/2004/REC-xml-infoset-20040204>.

[XMLSchema0] – The World Wide Web Consortium: *XML Schéma Part 0: Primer Second Edition, W3C Recommendation*, 28 Oct 2004. Dostupné na internete: <http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/>.

[XMLSig] – The World Wide Web Consortium: *XML-Signature Syntax and Processing, W3C Recommendation*, 12 Feb 2002. Dostupné na internete: <http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/>.

[WKPD] – Wikipedia, the free encyclopedia. Viď <http://www.wikipedia.org/>.

[SOSN] – Dennis Sosnoski: *Java Web services, Part 1: The year ahead in Java Web services*, 9 Feb 2006. Dostupné na internete: <http://www-128.ibm.com/developerworks/webservices/library/ws-java1.html>.

[WSDL11] – The World Wide Web Consortium: *Web Services Description Language (WSDL) 1.1, W3C Note*, 15 Mar 2001. Dostupné na internete: <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.

[WSDL2] – The World Wide Web Consortium: *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer, W3C Candidate Recommendation*, 27 Mar 2006. Dostupné na internete: <http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327>.

[WSIBP] – Web Services-Interoperability Organization: *Basic Profile Version 1.1, Final Materiál*, 24 Aug 2004. Dostupné na internete: <http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html>.

[WSIBSP] – Web Services-Interoperability Organization: *Basic Security Profile Version 1.0, Working Group Draft*, 2006-03-29. Dostupné na internete: <http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0-2006-03-29.html>.

[WSSec] – Organization for the Advancement of Structured Information Standards: *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004), OASIS Standard Specification*, 1 Feb 2006. Dostupné na internete: <http://docs.oasis-open.org/wss/v1.1/>.