Edícia výskumných textov
informatiky a informačných technológií

Štúdie vybraných tém softvérového inžinierstva (3)

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie a prezentácie informácie

Kniha vznikla a bola vydaná s finančnou podporou projektu:

Projekt JPD 3 2004/1-022: Podpora vzdelávania mladých vedeckých pracovníkov s cieľom vychovať tvorivých expertov – profesionálov informatikov – pre modernú spoločnosť založenú na vedomostiach

Európsky sociálny fond pomáha rozvíjať zamestnanosť podporovaním zamestnateľnosti, obchodného ducha, rovnakých príležitostí a investovaním do ľudských zdrojov.

Mária Bieliková,
Pavol Návrat
a kol.

Štúdie vybraných tém softvérového inžinierstva 3

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie a prezentácie informácie

STU FIIT
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

Edícia výskumných textov informatiky a informačných technológií

Štúdie vybraných tém softvérového inžinierstva (3)

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie
a prezentácie informácie

Autorský kolektív:

Mária Bieliková
Pavol Návrat
Michal Barla
Peter Bartalos
Marek Ciglan
Jozef Hamar
Martin Kiselkov
Michal Laclavík
Jakub Mažgut
Ján Máté
Ján Suchal
Martin Šeleng
Michal Tvarožek
Peter Vojtek

Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave
Ilkovičova 3
842 16 Bratislava

© Mária Bieliková, Pavol Návrat a kol., 2007

Text Design & Composition: Mária Bieliková
Copy Editor: Anton Andrejko
Cover Designer: Peter Kaminský

Vydala Slovenská technická univerzita v Bratislave
vo Vydavateľstve STU, Bratislava, Vazovova 5.

ISBN 978-80-227-2701-3

PREDHOVOR

Publikácia, ktorú dostávate do rúk, je v relatívne krátkom čase už treťou v poradí v *Edícii výskumných textov informatiky a informačných technológií* na témy z oblasti programových a informačných systémov. Všetky doterajšie *Štúdie vybraných tém softvérového inžinierstva* sa venujú dvom ťažiskovým okruhom. Prvým okruhom sú pokročilé metódy navrhovania programových systémov. Druhým okruhom sú pokročilé metódy získavania, vyhľadávania, reprezentácie a prezentácie informácií. Voľba oboch okruhov tém nebola náhodná. Všetky témy sú aktuálnymi témami súčasného výskumu v oblasti programových a informačných systémov. Ako také sú predmetom záujmu a štúdia výskumných študentov, t.j. najmä študentov doktorandského štúdia. Oni sú nielen prvými čitateľmi *Štúdií*, vybraní doktorandi sú aj autormi jednotlivých častí v každej z publikácií.

Prvé *Štúdie vybraných tém softvérového inžinierstva* sa zaoberali dvomi ťažiskovými témami. Prvou témou bola analýza návrhových vzorov, ktoré predstavujú jednu z kľúčových oblastí vyvíjajúcej sa disciplíny softvérového inžinierstva. Druhá časť obsahovala päť štúdií z vybraných tém programových a informačných systémov, ktoré diskutujú a analyzujú otvorené vedecké problémy v predmetnej oblasti aj v spojitosti so spracovaním informácií na internete.

Druhé *Štúdie vybraných tém softvérového inžinierstva* sa sústredili vo svoje prvej časti na analýzu rôznych aspektov toho, čo sa začalo nazývať webová inteligencia. V rámci druhej časti sme uviedli štyri štúdie, ktoré diskutujú a analyzujú vybrané otvorené vedecké problémy podobne ako v prvom zväzku.

Obdobný postup ako pri doterajších *Štúdiách* sme zvolili aj pri tomto zväzku. Vznikol na základe seminárov študentov doktorandského štúdia študijného programu programové systémy v odbore softvérové inžinierstvo na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave. Semináre podporil projekt Európskych štrukturálnych fondov, ktorého hlavným cieľom je podpora vzdelávania prostredníctvom motivačných nástrojov pre doktorandov a zvyšovaním kvality vzdelávania v treťom stupni vysokoškolského štúdia v oblasti informatiky a informačných technológií.

Informatika a informačné technológie sú kľúčovým prvkom budovania modernej spoločnosti „založenej na vedomostiach", ako je dnes módne vravieť. Mladí talentovaní absolventi druhého stupňa vysokoškolského štúdia v oblasti informatiky alebo príbuzných oblastiach majú v súčasnosti veľké možnosti uplatnenia sa v praxi. Súčasná spoločnosť však potrebuje aj špecializovaných odborníkov a vedeckých pracovníkov s ukončeným tretím stupňom vysokoškolského štúdia v študijných odboroch skupiny informatických vied, informačných a komunikačných technológií tak, aby bolo možné budovať ekonomiku založenú na najnovších vedeckých poznatkoch. V širšom kontexte ide o rozvoj spoločnosti (ak chcete, založenej na vedomostiach), nielen ekonomiky, schopnej vyrovnávať sa so zložitými výzvami, ktoré pred ňou stoja. S tým súvisí

potreba profesionálov v oblasti uchovávania, spracúvania a prezentácie informácií v bohatej palete reprezentácií ako základného prvku informačnej spoločnosti.

S rozvojom informatiky a informačných technológií sa posilňuje potreba odborníkov v špecializovaných oblastiach, schopných samostatne riešiť otvorené problémy, ktoré nemajú doteraz známe riešenia. Práve doktorandi sa na takúto úlohu pripravujú svojím doktorandským štúdiom. Z iného pohľadu ide totiž o výskum, ktorý je podstatnou náplňou ich štúdia. Jedným z prejavov fungujúcej výskumnej činnosti na pracovisku je seminár. Semináre, ktoré sa uskutočňujú na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave v rámci doktorandského štúdia sa zameriavajú na rôzne oblasti programových a informačných systémov. Zatiaľ čo v prvom zväzku *Štúdií* sme podchytili seminár venovaný návrhovým vzorom a v druhom seminár venovaný webovej inteligencii, v tomto zväzku sme spracovali témy seminára, venovaného podstate softvérovej architektúry. Je to nová oblasť, ktorá sa začala veľmi rozvíjať v deväťdesiatych rokoch minulého storočia. Dnes prichádza čas zosumarizovať, preformulovať a nanovo premyslieť výsledky, ktoré sa medzitým v tejto oblasti podarilo dosiahnuť.

Našou ambíciou bolo sprístupniť záujemcom o softvérové inžinierstvo vybrané témy a tým zdieľať výsledky seminárov a tvorivého prístupu študentov k jednotlivým témam v rámci diskusií. Výskumné texty v tejto publikácii sú vhodné aj pre študentov ďalších študijných programov v odboroch ako napr. informatika, aplikovaná informatika, informačné systémy, či umelá inteligencia a to v študijných programoch uskutočňovaných na Slovenskej technickej univerzite v Bratislave a aj na iných univerzitách.

Publikácia pozostáva z dvoch dielov. V prvom (Diel 1: Podstata architektúry softvéru) sa sústreďujeme na opis a analýzu rôznych prístupov k navrhovaniu a tvorbe architektúry softvéru. Druhý (Diel 2: Vybrané témy programových a informačných systémov) obsahuje štyri štúdie, ktoré diskutujú a analyzujú vybrané otvorené vedecké problémy z dynamicky sa rozvíjajúcej oblasti programových systémov so špeciálnym dôrazom na programové informačné systémy aj v spojitosti s Internetom.

Diel 1: Podstata architektúry softvéru

Čo je to podstata architektúry softvéru? Je to dosť ambiciózna otázka, avšak nanajvýš primeraná. Ísť až na podstatu toho, čo je predmetom skúmania, je predsa základom vedeckého bádania. Preto ak sa doktorandi v odbore softvérové inžinierstvo, prípadne v príbuznom odbore, zaoberajú podstatou architektúry softvéru, je takýto seminár na mieste. Pre tento výber témy seminára nám veľmi pomohla vhodná monografia – podobne, ako v predchádzajúcich ročníkoch seminárov. Ian Gorton sa podujal nenapísať len „ešte jednu ďalšiu knižku“ o softvérovej architektúre, ale posunúť poznanie o kúsok dopredu. Jeho monografia *Essential Software Architecture* z roku 2006 (vydavateľstvo Springer) je cenným príspevkom k syntetizujúcemu pohľadu na architektúru softvéru so všetkými výdobytkami, ktoré priniesol výskum, vývoj a prax v období posledných zhruba desiatich rokov. Knižka sa snaží poskytnúť podrobný úvod a systematický prehľad rôznych prístupov k navrhovaniu a k tvorbe architektúry softvéru. Ponúka štúdie súčasného stavu výskumu jednotlivých problémov. Zaoberá sa tiež niektorými aplikačnými aspektami.

Práve toto boli hlavné dôvody, pre ktoré sme sa rozhodli zamerať doktorandský seminár na jar 2007 na architektúru softvéru v takom chápaní, v akom ho prezentuje uvedená knižka. Vybrané kapitoly sa stali základom pre referáty, ktoré boli úvodmi pre

seminárne diskusie. Seminár v rámci doktorandského štúdia viedol Pavol Návrat. Doktorandi, ktorí referáty predniesli, dopracovali ich textovú podobu potom do výsledného tvaru, ktorý máme možnosť čítať v tomto zväzku.

Každá kapitola je tak výsledkom tvorivej činnosti, ku ktorej prispeli viacerí. Samotný text každej z nich písal ten-ktorý doktorand a jeho autorský prínos treba čo najvýraznejšie zdôrazniť a oceniť. Na seminároch prebiehala diskusia, na ktorej sa zúčastňovala celá skupina doktorandov a ktorá v tom-ktorom prípade ovplyvnila definitívne znenie opisu. Napriek tomu považujeme za korektné, aby sme označili ako jediných autorov jednotlivých opisov doktorandov, ktorí im dali písomnú podobu.

Náš výber tém z architektúry softvéru, ktorý sme zaradili do seminára (a teda aj do tejto knižky), možno rozčleniť do troch okruhov (kapitol tejto publikácie): úvod do architektúry softvéru (2 témy), princípy a postupy návrhu architektúry softvéru (4 témy) a prístupy k tvorbe architektúry softvéru (4 témy).

Autori sa podieľali na jednotlivých kapitolách takto:

- *Úvod do architektúry softvéru:*
  - Pochopenie architektúry softvéru: Michal Barla
  - Predstavenie prípadovej štúdie: Jozef Hamar
- *Princípy a postupy návrhu architektúry softvéru:*
  - Atribúty kvality softvéru: Michal Tvarožek
  - Architektúry a technológie pre spojovací softvér: Ján Suchal
  - Proces tvorby architektúry softvéru: Peter Bartalos
  - Dokumentovanie architektúry softvéru: Peter Bartalos
  - Návrh prípadovej štúdie: Michal Tvarožek
- *Prístupy k tvorbe architektúry softvéru:*
  - Pohľad do budúcnosti: Jakub Mažgut
  - Rady softvérových produktov: Jakub Mažgut
  - Aspektovo-orientovaný vývoj softvéru: Peter Vojtek
  - Modelom riadená architektúra: Ján Suchal
  - Architektúry a technológie orientované na služby: Jozef Hamar
  - Web so sémantikou: Michal Barla
  - Softvérové agenty: Peter Vojtek

Naše *Štúdie* sú dielom, ktoré sa líši od Gortonovej monografie. Taký bol náš úmysel, Gortonova monografia bola cennou inšpiráciou a východiskovým zdrojom vedomostí. Sú to dve rozdielne diela. Naše *Štúdie* nemôžu Gortonovu monografiu nahradiť. Práve naopak, všetkým záujemcom ju vrelo odporúčame.

Diel 2: Vybrané témy programových a informačných systémov

Do druhej časti zaraďujeme štyri štúdie, ktoré sa venujú vybraným otvoreným vedeckým problémom, týkajúcim sa programových a informačných systémov. Ide o oblasti, v ktorých prebieha veľmi intenzívny vývoj. Programové systémy sa stávajú systémami, pôsobiacimi v čoraz rôznorodejšom prostredí, vrátane internetu. Stávajú sa súčasťou čoraz komplexnejších systémov – na jednej strane rozsiahlych informačných systémov, na druhej strane systémov, spolu určených technickou platformou, ktorou už dávno nie

je len počítač v klasickom slova zmysle, ale aj najrôznejšie vnorené systémy, (tele)komunikačné systémy a pod.

Informačné systémy sa stávajú univerzálnym modelom spôsobov vyhľadávania, získavania, sprístupňovania, uchovávania, odovzdávania, spoločného používania, prezentovania informácií. I keď sa v zásade dá na ne nazerať odhliadnuc od toho, či sú operácie a procesy podporené počítačom alebo nie, čoraz viac sa zväčšuje praktický význam informačných systémov, ktoré sú realizované pomocou programových systémov (a tie samozrejme pomocou počítačových systémov alebo iných technických systémov, zahŕňajúcich počítače). Je to najmä preto, že softvérovo podporené informačné systémy majú vďaka možnostiam, ktoré poskytuje naprogramovaný počítač, výhody, ktoré sa ručným spracovaním nedajú dosiahnuť. Toto je súčasne aj argumentom pre úzke prepojenie výskumu v oboch oblastiach – ako softvérového inžinierstva, tak aj informačných systémov.

Štúdie sú výsledkom práce doktorandov v rámci ich doktorandského štúdia. Možno nezaškodí pripomenúť, že doktorandské štúdium sa koná pod vedením školiteľa. Na každej štúdii má preto podiel aj príslušný školiteľ. Napriek tomu však považujeme za korektné, aby sme označili ako jediných autorov jednotlivých štúdií doktorandov. Oni im dali písomnú podobu a prípadne aj predložili a úspešne obhájili ako písomnú časť svojej dizertačnej skúšky (J. Máté a M. Kiselkov) alebo ako časť svojej dizertácie (M. Láclavík).

Autori sa podieľali na jednotlivých kapitolách takto:

- Výskum vlastností súborových systémov: Ján Máté
  (školiteľ prof. Jiří Šafařík)
- Využitie znalostí z wikipédie pri hľadaní: Martin Kiselkov
  (školiteľ: prof. Pavol Návrat)
- Vyhľadávanie informácií: Michal Laclavík, Martin Šeleng, Marek Ciglan
  (školiteľ Dr. Ladislav Hluchý)
- FIPA compliant multi-agent systems: Michal Laclavík
  (školiteľ Dr. Ladislav Hluchý)

Poznamenajme, že v tomto zväzku *Štúdií* sa vyskytujú medzi autormi po prvýkrát aj výskumníci z Ústavu informatiky SAV v Bratislave. Je to nielen formálne a vecne odôvodnené (doktorandské štúdium sa tam uskutočňuje v spolupráci s FIIT STU a témy ich štúdií perfektne zapadajú do problematiky odboru), ale aj prirodzené, nakoľko s ústavom máme dlhoročnú intenzívnu spoluprácu.

Dúfame, že táto knižka poslúži záujemcom o poznanie programových a informačných systémov. Umožňuje spoločne využiť výsledky štúdia v tejto oblasti. Tešíme sa na prípadné odozvy alebo pripomienky.

Máj 2007,
Bratislava

Mária Bieliková a Pavol Návrat

OBSAH

DIEL I:
PODSTATA ARCHITEKTÚRY SOFTVÉRU

1 ÚVOD DO ARCHITEKTÚRY SOFTVÉRU

Táto kapitola vo svojej prvej časti predstavuje úvod do problematiky architektúry softvéru. Obsahuje niekoľko definícií pojmu architektúra softvéru, predstavuje architektúru v kontexte štruktúry systému, komunikácie jednotlivých modulov a riešení nefunkcionálnych požiadaviek. V tejto kapitole uvádzame aj požiadavky kladené na softvérového architekta týkajúce sa jeho odborných ale aj povahových vlastností.

V druhej časti kapitoly opisujeme prípadovú štúdiu, v ktorej predstavíme systém ICDE, na ktorom budú v ďalších kapitolách vysvetlené jednotlivé princípy tvorby architektúry softvéru, architektonické riešenia a technológie.

1.1 Pochopenie architektúry softvéru

Architektúra softvéru sa ako jedna z disciplín softvérového inžinierstva dostáva čoraz viac do popredia, na trhu práce je dopyt po špecializovaných profesionáloch – softvérových architektoch. Zároveň je však pojem „architektúra softvéru“ jedným z najviac „prepoužívaných“ a najmenej pochopených pojmov v oblasti vývoja softvéru.

1.1.1 Čo je architektúra softvéru?

V súčasnosti neexistuje jedna, ale hneď niekoľko definícií architektúry softvéru. Každá z nich sa na túto oblasť pozerá z iného uhla, vyzdvihuje iné aspekty. Jednou zo základných definícií architektúry softvéru je definícia organizácie IEEE:

> *„Architektúra je definovaná ako fundamentálna organizácia systému stelesnená v jeho komponentoch, vzťahoch medzi komponentmi navzájom a medzi prostredím. Architektúra v sebe zároveň zahŕňa princípy návrhu a rozvoja systému.“*

Architektúra teda predovšetkým definuje štruktúru systému pomocou komponentov (a ich interakcie) a definuje pravidlá návrhu na úrovni systému. Je dôležité si uvedomiť, že architektúra systému ohraničuje možnosti ďalšieho rozvoja systému.

Iné definície architektúry stavajú na definícii IEEE a zdôrazňujúc ďalšie dôležité body danej problematiky:

- *Architektúra je štruktúra alebo štruktúry systému.* Zdôrazňuje sa teda viacero možných a navzájom sa doplňujúcich pohľadov na systém.

- *Architektúra definuje externe pozorovateľné vlastnosti komponentov.* Zdôrazňuje sa nutnosť určitej úrovne abstrakcie v architektúre softvéru.
- *Architektúra definuje protokoly komunikácie, synchronizácie a prístupu k dátam. Fyzické rozmiestnenie komponentov.* Zdôrazňuje aspekty spolupráce viacerých komponentov.
- *Od architektúry sa odvíja škálovateľnosť a výkonnosť systému*.

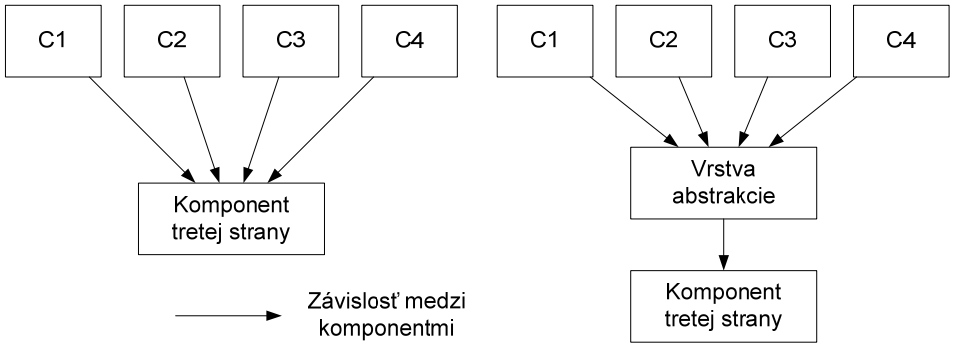
1.1.2 Architektúra ako definícia štruktúry

Jednou z hlavných úloh softvérového architekta je navrhnúť rozumné rozdelenie aplikácie do sady vzájomne prepojených komponentov. Samotný pojem komponent pritom nie je striktne definovaný – je to logicky rozoznateľná časť systému, s dobre definovanou funkcionalitou.

Práve funkcionalita zohráva pri dekompozícii hlavnú úlohu. Softvérový architekt definuje zodpovednosti komponentov – úlohy, ktoré komponent bude vykonávať v rámci aplikácie. Spoluprácou všetkých komponentov architektúry potom dosiahneme požadovanú funkcionalitu. Prístup k dekompozícii systému na komponenty s definovanými zodpovednosťami vychádza z objektovo-orientovaného prístupu k tvorbe softvéru a označuje sa ako zodpovednosťou riadený (*responsibility-driven*).

Funkcionalita však nemôže byť jediné kritérium pri definovaní štruktúry systému. Pri návrhu štruktúry sa usilujeme dosiahnuť čo najmenej závislostí medzi jednotlivými komponentmi a vytvoriť tak voľne zviazanú architektúru. Závislosť medzi komponentmi existuje vtedy, ak si zmena jedného komponentu vynúti zmenu iného komponentu. Dobrý návrh architektúry zmeny izoluje a zabraňuje ich šíreniu cez celú architektúru systému.

Kľúčovým nástrojom pri dosahovaní voľne zviazaných komponentov je abstrakcia. Príkladom je systém, ktorý využíva komponent tretej strany, na ktorého vývoj nemáme vplyv. Obrázok 1-1 vľavo zobrazuje návrh architektúry, kde sú komponenty systému priamo závislé od komponentu tretej strany. Ak sa zmení rozhranie komponentu, je veľmi pravdepodobné, že budeme musieť meniť všetky štyri systémové komponenty. Na obrázku 1-1 vpravo je ten istý problém vyriešený pridaním vrstvy abstrakcie, ktorá efektívne oddeľuje systém od komponentu tretej strany. V prípade zmeny komponentu sa zmeny dotknú iba vrstvy abstrakcie. Abstrakcia sa samozrejme nepoužíva len pri komponentoch tretích strán, ale pri návrhu celej architektúry systému.



Obrázok 1-1. Príklady závislosti komponentov.

1.1.3 Architektúra ako špecifikácia komunikácie

Rozdelenie aplikácie do sady komponentov so sebou prináša problém komunikácie medzi komponentmi (zasielanie dát a kontrolných údajov). Existuje viacero spôsobov ako riešiť komunikáciu od priameho volania metód po synchronizačné mechanizmy a prácu nad spoločnými dátami.

Softvérový architekt sa pri rozhodovaní a výbere riešenia môže riadiť zrealizovanými, otestovanými a funkčnými riešeniami, z ktorých sa generalizáciou stali architektonické vzory. Každý architektonický vzor definuje okrem delenia systému (napr. klient – server) aj zaužívaný a vhodný spôsob komunikácie.

Výhodou použitia vzorov je okrem istoty o vhodnosti použitia riešenia na daný problém aj zvýšenie čitateľnosti návrhu ostatnými členmi tímu. Vzory sú všeobecne známe a pod menom vzoru si člen tímu dokáže predstaviť štruktúru komponentov, komunikáciu a ďalšie abstraktné mechanizmy. Teda, už samotné pomenovanie použitého vzoru dokáže vyjadriť veľa aspektov návrhu architektúry.

1.1.4 Architektúra a nefunkcionálne požiadavky

To, že architektúra nerieši len funkcionálne požiadavky na systém sme už naznačili v predchádzajúcich podkapitolách. Nefunkcionálne požiadavky nie sú zachytené prípadmi použitia a nehovoria o tom *čo* má aplikácia robiť, ale *ako* má poskytovať požadovanú funkcionalitu.

Poznáme tri rozdielne oblasti nefunkcionálnych požiadaviek:

- Technické obmedzenia – obmedzujú návrh systému z hľadiska technológií, ktoré je nutné použiť (napr. platforma .NET pretože máme len .NET vývojárov). Zvyčajne sa o týchto obmedzeniach nedá vyjednávať.
- Biznis obmedzenia – podobne ako technické obmedzenia obmedzujú návrh systému, avšak z iných dôvodov. Tieto obmedzenia vychádzajú z biznis cieľov aplikácie, stratégie spoločnosti, úsporných opatrení (napr. použitie open-source riešení) a pod.
- Atribúty kvality – definujú požiadavky na aplikáciu v oblasti škálovateľnosti, dostupnosti, meniteľnosti, prenosnosti, použiteľnosti, výkonu a pod.

1.1.5 Architektúra ako abstrakcia

Softvérový architekt musí mať schopnosť abstrakcie. Mal by dokázať abstrahovať návrh architektúry na viacerých úrovniach a to až do takej miery, že sa zmestí na jednu stranu, na ktorej sú zobrazené hlavné komponenty systému a ich vzťahy (anglicky sa potom nazýva *marketecture* pre jej použitie v komunikácií s manažmentom).

Abstrakcia je v architektúre použitá vždy, keďže zobrazujeme komponenty ako čierne skrinky, ktoré majú iba externe pozorovateľné vlastnosti. Avšak, to čo je na jednej úrovni reprezentované ako čierna skrinka sa môže v ďalšej úrovni (iný, podrobnejší pohľad na architektúru systému) dekomponovať na sadu subkomponentov. Tento prístup sa nazýva hierarchická dekompozícia.

Hierarchická dekompozícia sa používa len tam, kde je potrebná. Hlavný softvérový architekt zvyčajne navrhne základné komponenty systému, ich zodpovednosti a prepojenia a detailný návrh komponentov ponechá na jednotlivé vývojové tímy.

Niektoré komponenty však môže dekomponovať priamo architekt, ak si to vyžadujú požiadavky (najmä nefunkcionálne) a ovplyvniť tak návrh komponentu.

1.1.6 Pohľady na architektúru

Existuje viacero pohľadov na architektúru softvéru. Základnými sa stali tieto štyri:

- *Logický pohľad* opisuje podstatné elementy architektúry a ich vzťahy, čiže štruktúru aplikácie.
- *Procesný pohľad* sa zameriava na súbežnosť a komunikačné aspekty architektúry.
- *Fyzický pohľad* zobrazuje mapovanie hlavných procesov a komponentov na aplikačný hardvér.
- *Vývojový pohľad* zachytáva internú organizáciu softvérových komponentov do balíčkov, tried a pod. tak, ako sú uložené v prostredí pre správu konfigurácií.

1.1.7 Softvérový architekt

Práca softvérového architekta nespočíva len vo vytvorení architektúry. Podľa SEI[1] musí softvérový architekt ďalej:

- zdokumentovať a komunikovať vytvorenú architektúru;
- uistiť sa, že ju každý používa a to správnym spôsobom;
- uistiť sa, že vytváraný softvér je v súlade s architektúrou;
- uistiť sa, že manažment architektúre rozumie na požadovanej úrovni detailnosti;
- riešiť technické problémy;
- riešiť nezhody a robiť kompromisy;
- uistiť sa, že sa používajú správne modelovacie techniky, aby sa vedelo v dostatočnom predstihu, či sa dosiahnu požiadavky na kvalitu ako je výkonnosť;
- spolupodieľať sa na problémoch ako je výber konkrétnych nástrojov a prostredí;
- uistiť sa, že architektúra je správna z hľadiska nasadenia, údržby a ďalšieho vývoja systému;
- udržiavať morálku vo vývojom tíme;
- chápať a plánovať vývojové smery aplikácie, plánovať nasadenie nových technológií;
- vykonávať manažment rizík súvisiacich s architektúrou.

Uvedený zoznam nie je kompletný a zrejme ani nikdy nemôže byť, keďže každý má svoje vlastné očakávania a definície práce softvérového architekta. Zo zoznamu sa však dajú abstrahovať základné zručnosti softvérového architekta v týchto štyroch oblastiach:

- *Vzťahy* – architekt hrá významnú rolu vo viacerých vzťahoch v rámci spoločnosti ako aj navonok smerom k zákazníkovi. Často sa zúčastňuje stretnutí so zákaz-

[1] Software Engineering Institute, http://www.sei.cmu.edu

níkmi spolu s analytikmi. Tvorí prostredníka medzi viacerými vývojovými tímami, keďže práve architektúra systému predstavuje bod, kde sa spájajú ich záujmy. Je vo vzťahu k manažmentu, ktorému odôvodňuje návrh, rozhodnutia a ceny. Musí takisto spolupracovať s obchodným oddelením, ktorým pomáha v propagovaní systému možným zákazníkom. Softvérový architekt často zjednocuje a vysvetľuje terminológiu rôznych skupín účastníkov vývoja.

- *Softvérové inžinierstvo* – architekt musí byť schopný vytvoriť dobrý návrh, ktorý je v súlade so zásadami softvérového inžinierstva. Návrh musí byť príslušne zdokumentovaný, odôvodnený a jasný ostatným zúčastneným stranám. Architekt si musí byť vedomý možných dopadov jeho rozhodnutí.
- *Znalosť technológií* – architekt musí mať detailné vedomosti o technológiách, ktoré sa vzťahujú na typy aplikácií, ktoré navrhuje. Tieto vedomosti využíva pri rozhodovaní o použití komponentov a technológií tretích strán. Architekt musí sledovať výskum a vývoj nových štandardov, technológií a produktov a chápať ich dopad na svoju doménu.
- *Manažment rizík* – architekt musí byť opatrný. Neustále identifikuje a vyhodnocuje riziká spojené s návrhom a výberom technológií. Riziká dokumentuje a riadi v súčinnosti s manažmentom.

1.1.8 Architektúra a technológie

Návrh architektúry sa nedá efektívne validovať a testovať predtým, ako sa vyvinú aspoň základné časti systému. Prototypovanie tento problém rieši iba čiastočne a nemusí dávať úplnú istotu o vhodnosti návrhu. Z tohto dôvodu sa architekti spoliehajú na odskúšané prístupy k riešeniu skupín problémov – architektonické vzory.

Architektonický vzor však predstavuje abstraktné riešenie, ktoré vznikne generalizovaním niekoľkých konkrétnych prístupov k riešeniu problému. Toto abstraktné riešenie nie je spustiteľné na počítači a je úlohou softvérových inžinierov vytvoriť zo vzoru jeho konkrétnu použiteľnú implementáciu.

Najčastejšie používané vzory sú podporované verejne dostupnými aplikáciami, ktoré poskytujú požadovanú funkcionalitu a sú pritom znovupoužiteľné a aplikačne nezávislé. Vývojový tím teda nemusí nanovo vyvíjať všetky komponenty systému, ale môže niektoré časti pokryť existujúcimi komerčnými alebo voľne dostupnými riešeniami.

Úlohou softvérového architekta je vybrať najvhodnejší produkt na trhu, pričom musí uvažovať rôzne aspekty ako je cena, kvalita a funkcionalita. Takmer žiadna aplikácia neposkytuje presne to, čo si žiada náš návrh. Nákupom aplikácie buď zaplatíme aj za funkcionalitu, ktorú nepožadujeme (ktorú nikdy nepoužijeme aj keď sme si za ňu zaplatili) alebo nám naopak funkcionalita bude chýbať (pričom riešením je doplnenie chýbajúcich častí alebo zmena návrhu).

Softvérový architekt musí zvážiť všetky výhody a nevýhody jednotlivých produktov a vykonať správnu voľbu.

1.1.9 Vyjadrenie architektúry

Použitie architektonických vzorov zvyšuje čitateľnosť návrhu architektúry a zjednodušuje jeho komunikáciu smerom k vývojovým tímom a iným zúčastneným stranám. S čitateľnosťou návrhu úzko súvisí aj spôsob vyjadrenia architektúry softvéru.

Najprirodzenejšie je vyjadriť architektúru obrázkom, resp. diagramom, čo so sebou prináša problém rôznej interpretácie diagramu zúčastnenými osobami. Aby sa takýmto problémom predišlo, používa sa pri vývoji softvéru notácia jazyka UML (*Unified Modeling Language*).

Keďže architektúra softvéru je široký pojem, ktorý v sebe zahŕňa množstvo aspektov aplikácie, používajú sa na jej vyjadrenie rôzne UML diagramy v závislosti od toho, akú črtu systému chceme diagramom objasniť.

Štruktúra systému sa zvyčajne zachytáva diagramom komponentov (*component diagram*), ktorý nám umožňuje modelovať softvérové komponenty a (čo je podstatné) rozhrania týchto komponentov. Práve rozhrania predstavujú miesta, kde sa najčastejšie stretáva práca viacerých vývojových tímov.

Na vyjadrenie fyzického pohľadu na systém, v ktorom identifikujeme systémový hardvér, softvér nainštalovaný na tomto hardvéri a spôsob prepojenia hardvéru, sa používa diagram rozmiestnenia (*deployment diagram*).

1.1.10 Zhrnutie

Pri stavbe domu sa postupuje podľa určitého plánu a v súlade s projektom stavebného architekta. Podobne sa aj pri vývoji komplexnejších softvérových systémov ukázalo ako kľúčové mať k dispozícii architektúru. Tá predstavuje abstrakciu systému, základ o ktorý sa môžu oprieť všetci účastníci vývoja aplikácie.

Architektúra softvéru je jednou z disciplín softvérového inžinierstva, jej návrh predurčuje budúce vlastnosti softvéru, jeho prednosti a obmedzenia. Aj keď táto disciplína nemá svoju jednoznačnú definíciu, je dobre definovaná tým, čo od architektúry očakávame a aké problémy rieši.

Tvorbu architektúry softvéru zatiaľ nedokážeme automatizovať. Aj keď sú k dispozícii rôzne rokmi overené architektonické vzory, tvorba architektúry komplexného systému predstavuje náročný proces, ktorý vyžaduje kreatívny prístup. Softvérový architekt by mal byť vynikajúci softvérový inžinier, ktorý ovláda nielen praktiky tohto odboru, ale má aj potrebné vedomosti z príslušnej domény, prehľad v technológiách a vynikajúce komunikačné a rozhodovacie schopnosti.

1.2 Predstavenie prípadovej štúdie

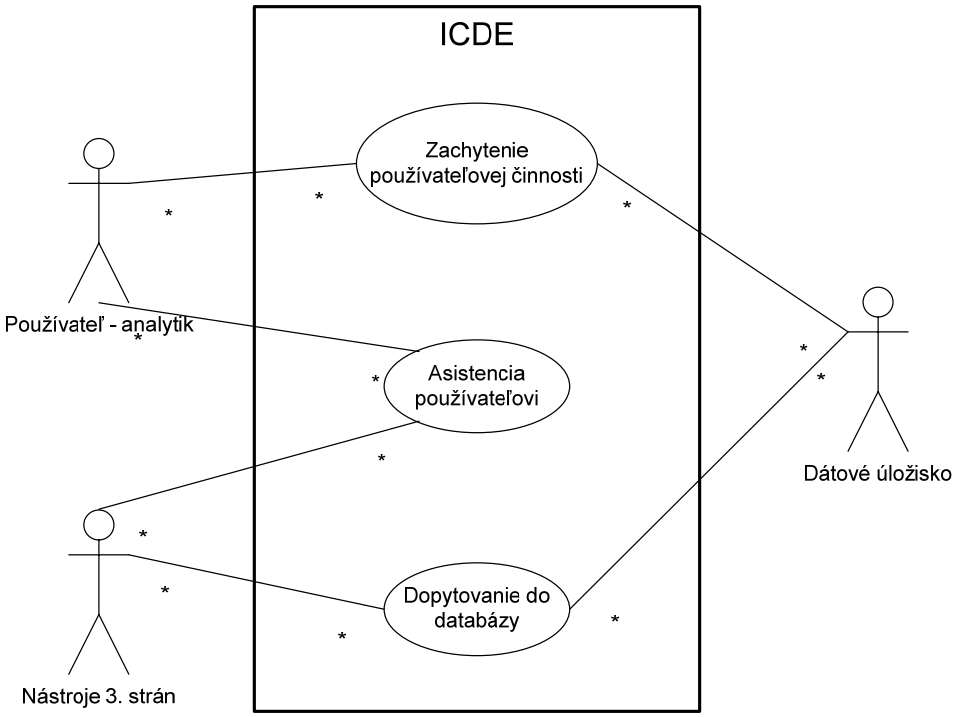
Táto kapitola predstavuje prípadovú štúdiu, ktorá bude použitá v ďalších kapitolách na demonštráciu návrhových postupov v tejto knihe. Prostredie pre zachytávanie a rozširovanie informácií (*Information Capture and Dissemination Environment* – ICDE) je systém pre pomoc pri elektronickom vyhľadávaní informácií v odborných profesiách

1.2.1 Prehľad požiadaviek

ICDE umožňuje zachytiť preddefinované činnosti a akcie používateľa a uložiť ich do databázy. Ak napríklad používateľ použije webový vyhľadávač, do databázy sa uloží dopytovaný reťazec a kópie webových stránok vrátené vyhľadávačom, ktoré si používateľ pozrie.

Tieto dáta môžu byť použité nástrojmi, ktoré sa môžu ďalej pokúšať pomôcť vyhľadať relevantné informácie nielen na základe kľúčových slov, ale aj na základe akcií, ktoré používateľ nad danými informáciami vykonal. Tiež sa môžu rozhodnúť do výstupu pridať informácie z webových stránok, ktoré predpokladajú že používateľ omylom prehliadol.

Obrázok 1-2 prezentuje prípady použitia ICDE. Tri základné prípady použitia sú Zachytenie používateľovej činnosti, Dopytovanie dát z databázy a Interakcia nástrojov tretích strán pre vyhodnocovanie s používateľom.



Obrázok 1-2. Prípady použitia ICDE.

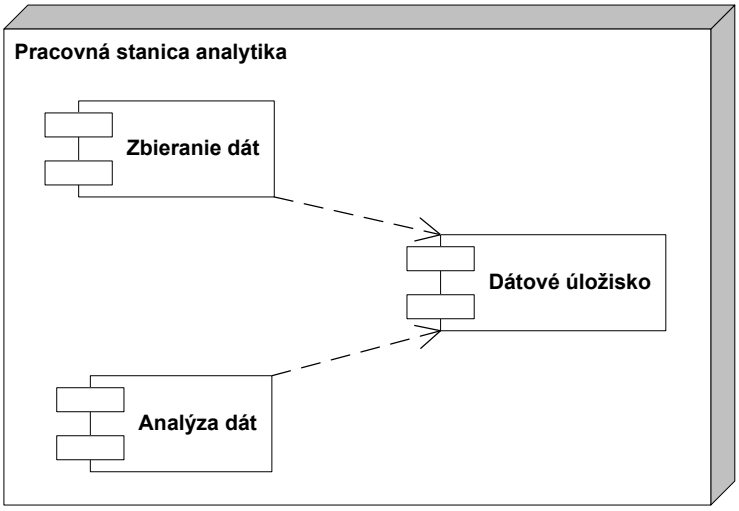
1.2.2 ICDE v1.0

Hlavným cieľom ICDE v1.0 bola implementácia prípadu použitia Zachytenie používateľovej činnosti. Realizovalo sa zbieranie informácií potrebných pre ostatné operácie a tiež bolo navrhnuté a implementované dátové úložisko.

ICDE v1.0 bolo otestované na malom počte používateľov a úspešne demonštrovalo funkcionalitu a koncept zachytávania dát. Testovanie prebehlo na operačnom systéme Microsoft Windows XP, pričom všetky operácie boli vykonávané na strane používateľa a implementované v jazyku Java (obrázok 1-3).

Funkcie komponentov sú:

- Komponent *Zbieranie dát* pozostáva z viacerých procesov, ktoré v pozadí sledujú používateľovu činnosť a zaznamenávajú relevantné informácie. Tieto pozostávajú z dokumentov, ktoré používateľ otvára, modifikuje, resp. vyhľadáva na Internete a z informácií o spúšťaní a zatváraní aplikácií.



Obrázok 1-3. Architektúra ICDE v1.0.

- *Dátové úložisko* predstavuje relačnú databázu, kde sú dáta ukladané v tabuľkách, ktoré zodpovedajú jednotlivým činnostiam. Zaznamenáva sa aj čas, aby bolo možné zrekonštruovať používateľovu činnosť. Veľké objekty, ako napríklad obrázky alebo dokumenty sú uložené v binárnom formáte.
- *Analýza dát* predstavuje grafické používateľské rozhranie slúžiace na jednoduché zobrazenie zachytených dát. Používalo sa na testovanie a základnú prezentáciu uložených dát ďalším vývojárskym tímom, aby nadobudli predstavu o zachytávaných dátach nad, ktorými je možná ďalšia analýza.

1.2.3 Biznis ciele

ICDE v2.0 má vyššie ambície, jeho základné ciele sú:

- presvedčiť ďalšie vývojové tímy aby písali aplikácie pre ICDE,
- predstaviť ICDE zákazníkom a presvedčiť ich o prínose pre ich činnosť.

Na dosiahnutie týchto cieľov bolo treba vyriešiť tieto zásadné problémy:

- jednoduché a spoľahlivé rozhranie pre prístup k dátovému úložisku,
- heterogénna platforma,
- možnosť komunikácie medzi používateľmi ICDE a nástrojmi pre analýzu bežiacimi inde ako na používateľovom počítači,
- možnosť škálovať dátové úložisko až do počtu 150 používateľov,
- lacné nasadenie pre koncového používateľa.

Základný predpoklad pre to, aby sa vytvoril záujem tretích strán o vývoj aplikácií pre ICDE, je existencia silného a jednoduchého rozhrania (API), ktoré bude nezávislé od platformy (Windows, Linux, Unix).

Ďalší dôležitý faktor je počet používateľov. Prieskum ukázal, že potenciálny zákazníci majú 10-150 analytikov – potenciálnych používateľov. Softvér by teda mal umožňovať

prácu takéhoto počtu používateľov, pričom výsledný návrh systému by nemal obsahovať obmedzenia, ktoré môžu v budúcnosti počet používateľov obmedziť. Taktiež by malo byť minimalizované použitie nákladných komerčných riešení a technológií, ktoré by zdvihlo cenu koncového produktu pre zákazníka.

Pre realizáciu ICDE v2.0 bol vyčlenený časový horizont 12 mesiacov. Priebežný výstup bol naplánovaný po 6 mesiacoch pre vývojárov, aby boli schopný vyvíjať nástroje súbežne s vývojom ICDE v2.0. Celkový rozpočet projektu bol obmedzený, čo malo za následok striktné obmedzenie realizácie nových vlastností.

1.2.4 Zhrnutie

ICDE predstavuje zaujímavú prípadovú štúdiu. Existujúcu architektúru treba rozšíriť o nové vlastnosti a možnosti, pričom časové a finančné obmedzenia vytvárajú malý priestor pre zmeny v existujúcej infraštruktúre, resp. zmeny vlastností, ktoré sa budú implementovať.

2 PRINCÍPY A POSTUPY NÁVRHU ARCHITEKTÚRY SOFTVÉRU

Významným aspektom práce softvérového architekta je tvorba architektúry softvéru, ktorá spĺňa stanovené požiadavky a ohraničenia na výslednú funkcionalitu a kvalitu softvéru. Kvalita softvéru opisuje jeho vhodnosť pre daný účel, pričom výsledná kvalita softvéru do značnej miery závisí od návrhu jeho architektúry a jej vlastností, pretože už samotná architektúra definuje rámec, ohraničenia a kompromisy, ktoré musí vytváraný systém spĺňať. Z nehmotnej povahy softvéru a z rôznorodosti spôsobov jeho nasadenia vyplýva nemožnosť priameho merania jeho kvality. Tú možno merať len nepriamo pomocou rôznych atribútov kvality, ktoré opisujú vlastnosti softvéru z viacerých pohľadov ako sú výkonnosť, škálovateľnosť, prenositeľnosť a ďalšie.

Po úvodnom určení požiadaviek a ohraničení pre architektúru, softvérový architekt pristúpi k samotnému návrhu architektúry pre konkrétny systém so špecifickými požiadavkami na kvalitu. Skúsený architekt pri návrhu využíva praxou overené architektonické vzory, ktoré zabezpečia dosiahnutie požadovaných atribútov kvality. Pre tvorbu stredných a väčších softvérových systémov je zvyčajne nutný výber a použitie vhodného integračného spojovacieho softvéru (*middleware*) ako sú napr. aplikačné servery, sprostredkovatelia správ, resp. orchestrácia biznis procesov.

Navrhnutá architektúra softvéru musí na záver prejsť validáciou – overením jej vhodnosti, resp. splnenia všetkých požiadaviek, ktoré architekt vykoná pomocou scenárov použitia systému. Niektoré vlastnosti systému, napr. výkonnosť nie je možné efektívne overiť staticky, ale len na existujúcej aplikácii, prípadne jej prototype.

Neoddeliteľnou súčasťou života softvérového architekta je dokumentácia jeho práce pomocou zaužívaných postupov, medzi ktoré patrí jazyk UML a šablóny pre dokumentáciu jednotlivých fáz tvorby architektúry softvéru od zaznamenania požiadaviek na systém, cez dokumentáciu rozhodnutí a kompromisov vykonaných počas jeho návrhu až po spôsob overenia architektúry pomocou analýzy možných scenárov použitia systému.

V tejto kapitole postupne prevedieme čitateľa prácou softvérového architekta a podrobne opíšeme každú z uvedených fáz tvorby architektúry softvéru.

2.1 Atribúty kvality softvéru

Kvalita vo všeobecnosti predstavuje veľmi dôležitú, aj keď často nesprávne chápanú vlastnosť softvérových produktov. Podľa definície, kvalita je význačná charakteristika nejakej veci. Alternatívne, kvalita môže predstavovať aj všeobecný pohľad na štandard, resp. úroveň nejakej veci. V kontexte tvorby softvéru kvalitu často chápeme ako vhodnosť softvéru pre daný účel.

Celkovo sa kvalita softvéru vyznačuje viacerými špecifikami v porovnaní s „tradičným“ chápaním kvality. Jedným z nich je skutočnosť, že kvalitu softvérového produktu zvyčajne nemožno priamo merať, čo je dané už samotnou jeho nehmotnou povahou. Z tohto dôvodu sa v softvérovom inžinierstve presadil názor, že „kvalitný“ softvérový proces je predpokladom pre tvorbu kvalitného softvérového produktu.

Ďalšou význačnou vlastnosťou softvéru je fakt, že na kvalitu softvéru existuje viacero rozličných pohľadov. Inak sa na kvalitu softvéru pozerajú jeho vývojári, inak testeri a možno úplne inak ju chápe zákazník, resp. koncový používateľ. Kvalitu softvéru možno tiež chápať v kontexte jeho jednotlivých súčastí ako kvalitu dokumentácie, kvalitu zdrojového kódu, kvalitu bežiaceho softvéru počas prevádzky alebo ako celkovú spokojnosť používateľov so softvérom.

Na opis kvality softvéru boli rôznymi normami (napr. ISO 9126) definované atribúty kvality softvéru, ktoré opisujú vlastnosti softvérového produktu z rôznych pohľadov:

- Interné atribúty kvality opisujú pohľad vývojárov, ktorých zaujímajú najmä vlastnosti softvéru spojené s jeho zdrojovým kódom a dokumentáciou.
- Externé atribúty kvality opisujú dynamický pohľad na správanie sa softvéru počas jeho vykonávania.
- Atribúty kvality softvéru počas prevádzky opisujú najmä pohľad zákazníka, resp. pohľad používateľov softvéru na mieru naplnenia ich požiadaviek a celkovú spokojnosť zo softvérom v konkrétnom prostredí.

V súčasnosti existuje pomerne veľké množstvo atribútov kvality softvéru, ktoré možno použiť na opis konkrétnych (nefunkcionálnych) vlastností softvéru. V praxi kvalita softvéru vždy závisí od konkrétneho prostredia, v ktorom je daný softvér nasadený a tiež od požiadaviek jeho používateľov. Dôsledkom je, že kvalitu softvéru nemožno všeobecne určiť na základe jednotlivých jeho atribútov kvality. Napr. ten istý softvér môže mať pre jedného zákazníka primeranú kvalitu avšak pre iného zákazníka môže byť úplne nevyhovujúci.

Zložitosť problematiky kvality a tvorby softvéru si vyžaduje softvérového architekta, ktorý dokáže identifikovať dôležité vlastnosti vytváraného softvérového produktu, nájsť vhodný kompromis medzi požadovanými atribútmi kvality a navrhnúť softvér tak, aby spĺňal množinu konkrétnych požiadaviek na atribúty kvality. Tieto sú najčastejšie súčasťou nefunkcionálnych požiadaviek, ktoré opisujú ako má systém spĺňať jednotlivé funkcionálne požiadavky. Podrobný opis problematiky nefunkcionálnych požiadaviek obsahuje práca (Chung et al., 1999).

Nutným predpokladom na naplnenie požiadaviek na kvalitu je ich jednoznačná špecifikácia, ktorú je možné následne použiť pri návrhu systému. Jednotlivé požiadavky musia hovoriť ako systém má naplniť konkrétne potreby. Požiadavka „systém musí byť škálovateľný“ je príliš nepresná na to, aby bola k niečomu užitočná.

Ako uvidíme neskôr, existuje viacero rôznych pohľadov na škálovateľnosť, ktoré sa vzťahujú na rôzne charakteristiky vytváranej aplikácie. Dôležité je povedať, či má aplikácia zvládnuť zvýšené množstvo údajov, alebo zvýšený počet súčasne pripojených používateľov, alebo či má byť nasaditeľná na 10 tisíc počítačov oproti pôvodným 100 bez výrazne zvýšených nárokov na inštaláciu. Má prípadne vyhovieť všetkým uvedeným požiadavkám?

Špecifikovať, ktoré z uvedených požiadaviek na škálovateľnosť má aplikácia spĺňať je z pohľadu architektúry veľmi dôležité, pretože riešenia pre každú z nich sú odlišné. Je nevyhnutné špecifikovať konkrétne a merateľné požiadavky na atribúty kvality: „Musí byť možné škálovať nasadenie z počiatočných 100 geograficky rozptýlených počítačov na 10 tisíc geografický rozptýlených počítačov bez zvýšených nárokov a nákladov na inštaláciu a konfiguráciu.“

Takto presný a zmysluplný opis požiadaviek softvérového architekta nasmeruje na vytvorenie riešenia s použitím konkrétnych technológií, ktoré umožnia bezproblémovú inštaláciu a nasadenie softvéru.

Dôležité je uvedomiť si, že mnohé požiadavky na kvalitu je len ťažko možné testovať, resp. validovať. V uvedenom príklade je veľmi nepravdepodobné, že by počiatočné testovanie systému zahŕňalo aj inštaláciu na 10 tisíc počítačov.

V takomto prípade architektove skúsenosti a zdravý rozum hovoria, že navrhnuté riešenie bude zjavne použiteľné pre počiatočné nasadenie na 100 počítačov. Následne na základe konkrétnych navrhnutých postupov (napr. stiahnutie cez Internet) môžeme v rozsahu našich najlepších schopností analyzovať, či navrhnuté riešenie bude spĺňať dané požiadavky. Ak neobjavíme žiadne očividné nedostatky alebo problémy, môžeme predpokladať, že riešenie bude škálovateľné. Bude však naozaj škálovateľné na 10 tisíc počítačov? So softvérom existuje len jeden absolútne spoľahlivý spôsob ako si byť stopercentne istý – „všetko sú len reči, pokiaľ kód nebeží“.

Pretože existuje veľa všeobecných atribútov kvality, ktorých opis presahuje rámec tejto publikácie, zameriame sa len na opis vybraných atribútov, ktoré sú najdôležitejšie pre bežné softvérové aplikácie. Súčasne opíšeme aj najčastejšie používané mechanizmy softvérovej architektúry, ktoré slúžia na ich naplnenie.

2.1.1 Výkonnosť

Často uvádzaným atribútom kvality softvéru je výkonnosť (*performance*), aj keď pre mnohé praktické aplikácie výkonnosť nie je v súčasnosti závažným problémom. Výkonnosť je často uvádzaná aj preto, že je ľahko kvantifikovateľná a následne jednoducho overiteľná. V prípadoch, kedy však na výkonnosti aplikácie naozaj záleží, predstavuje výkonnosť jeden z kritických atribútov kvality, na ktorom stojí a padá úspech výsledného softvéru.

Požiadavka na výkonnosť aplikácie opisuje metriku na množstvo práce, ktoré musí aplikácia vykonať za jednotku času, resp. definuje časový limit, do ktorého musí aplikácia vykonať príslušnú úlohu. V praxi len málo aplikácií vyžaduje tvrdé ohraničenia na odozvu v reálnom čase ako je tomu napr. vo vojenských aplikáciách, kde oneskorenie výstupu o pár milisekúnd môže mať mimoriadne nežiaduce dôsledky. Napriek tomu sa aplikácie spracúvajúce stovky a tisíce transakcií za sekundu často vyskytujú v mnohých veľkých organizáciách v oblasti financií, telekomunikácií alebo na úradoch.

Výkonnosť možno merať pomocou nasledovných metrík:

- priepustnosť (*throughput*),
- doba odozvy (*response time*),
- časový limit (*deadline*).

Priepustnosť

Priepustnosť predstavuje mierku množstva práce, ktorú aplikácia vykoná za jednotku času. Typicky sa priepustnosť udáva v transakciách za sekundu (tps), prípadne v správach za sekundu (mps). Napr. internetová banková aplikácia môže spracovať 100 transakcií za sekundu, pričom skladový systém môže spracovať 50 správ za sekundu.

Dôležité je definovať, či sa jedná o priemernú (*average throughput*) priepustnosť napr. za jeden deň alebo o špičkovú (*peak throughput*) priepustnosť. Učebnicovým príkladom je systém pre prijímanie stávok na konské dostihy. Takáto aplikácia väčšinu času nespracúva žiadne požiadavky a následne má nízke a ľahko dosiahnuteľné požiadavky na priemernú priepustnosť. Na druhú stranu, 5 až 10 minút pred každým dostihom systém zaznamená stovky stávok každú sekundu. Ak systém nedokáže spracovať stávky, spoločnosť stratí peniaze a znechutí svojich zákazníkov. Výsledná aplikácia teda musí byť navrhnutá tak, aby spĺňala požiadavky na očakávanú špičkovú priepustnosť a nie na priemernú priepustnosť.

Doba odozvy

Doba odozvy predstavuje mieru oneskorenia, ktoré má aplikácia pri spracovaní transakcií, pričom je najčastejšie spájaná s časom za ktorý aplikácia reaguje na vstupy. Rýchla doba odozvy umožňuje používateľom pracovať so systémom efektívne. Výborný príklad je skladový systém veľkého hypermarketu, kde rýchla odpoveď pokladní na zosnímaný čiarový kód tovaru, obsahujúca jeho cenu, znamená, že zákazníci môžu byť obslúžení rýchlo. Dôsledkom sú spokojní zákazníci aj predajcovia.

Podobne ako v predchádzajúcom prípade je dôležité rozlišovať priemernú dobu odozvy a garantovanú dobu odozvy. Vybrané aplikácie môže požadovať, aby všetky požiadavky boli spracované do daného časového limitu, ktorý zodpovedá garantovanej dobe odozvy. Iné aplikácie zvyčajne udávajú priemernú dobu spracovania požiadavky, čím umožňujú dlhšiu dobu odozvy v čase vysokej záťaže systému. V takomto prípade sa často udáva horná hranica na dobu odozvy, napr. 90% požiadaviek musí byť spracovaných do troch sekúnd, pričom žiadna požiadavka nemôže byť spracúvaná dlhšie ako 10 sekúnd.

Časový limit

Pravdepodobne každý už počul o systéme na predpoveď počasia, ktorý potreboval 36 hodín na prípravu predpovede počasia na nasledovný deň. Aj napriek svojmu neúspechu predstavuje daný systém výborný príklad požiadavky na časový limit spracovania vstupu. V oblasti tvorby softvéru sú časové limity najčastejšie spájané so systémami na dávkové spracovanie vstupu. Napr. systém na spracovanie platieb sociálnych dávok musí skončiť spracovanie a umožniť prevod peňazí načas. V opačnom prípade poberatelia dávok dostanú platby oneskorene, čo spôsobí problémy nielen im.

Všetky uvedené metriky výkonnosti možno jasne špecifikovať a validovať. Je však nevyhnutné vyvarovať sa klasickej chybe, ktorá leží v definícii transakcie, resp.

požiadavky, ktoré boli v predchádzajúcom použité len veľmi voľne. Principiálne treba vhodným spôsobom definovať pracovnú záťaž systému, pretože množstvo práce nutné na spracovanie požiadavky silne závisí od konkrétnej aplikácie, resp. od konkrétneho typu požiadavky v rámci aplikácie. V praxi neexistuje všeobecná miera pracovnej záťaže systému – v jednom systéme sa môžu vyskytovať tak jednoduché ako aj zložité transakcie. Následne je nutné presne definovať aké zloženie transakcií zodpovedá typickej pracovnej záťaži systému (*transaction mix*), pre ktorú uvádzame konkrétne požiadavky na výkonnosť.

Výkonnosť systému ICDE

Z pohľadu systému ICDE (*Information Capture and Dissemination Environment*), ktorý používame ako prípadovú štúdiu, predstavuje výkonnosť veľmi významný atribút kvality, pretože je nevyhnutné zabezpečiť interaktívny spôsob práce používateľov so systémom. Zachytenie a záznam udalostí na strane klienta musia byť vykonané bez vnímateľného oneskorenia, aby nerušili prácu používateľov.

Zachytenie udalostí vyvolaných používateľom a systémom rieši ICDE aplikácia na strane klienta pomocou spätných volaní API vložených do používateľského rozhrania. Je na aplikácii samotnej, aby to robila rýchlo a maximálne efektívne.

Po zachytení udalosti je nutné ju zaznamenať – ICDE klient volá služby servera, ktorý udalosť uloží. Uloženie udalosti by nemalo ovplyvňovať používateľa a preto je udalosť zaznamenaná do vyrovnávacej pamäte a asynchrónne iným vláknom zapisovaná na ICDE server. Toto riešenie oddeľuje zachytenie a záznam udalostí systémom ICDE, kde oneskorený záznam udalosti serverom neovplyvňuje používateľské rozhranie.

Požiadavky na výkonnosť ICDE serveru sú ľahko špecifikovateľné – ICDE server musí poskytovať priemernú dobu odozvy na požiadavky klientov do jednej sekundy.

2.1.2 Škálovateľnosť

Škálovateľnosť (*scalability*) opisuje ako dobre riešenie konkrétneho problému funguje v prípade, že sa veľkosť, resp. zložitosť problému zväčšuje. V kontexte architektúry škálovateľnosť opisuje ako dobre sa daný návrh architektúry vyrovná s narastajúcimi požiadavkami na niektorú časť systému. Konkrétna požiadavka na škálovateľnosť systému vyžaduje definovanie toho, čo bude narastať:

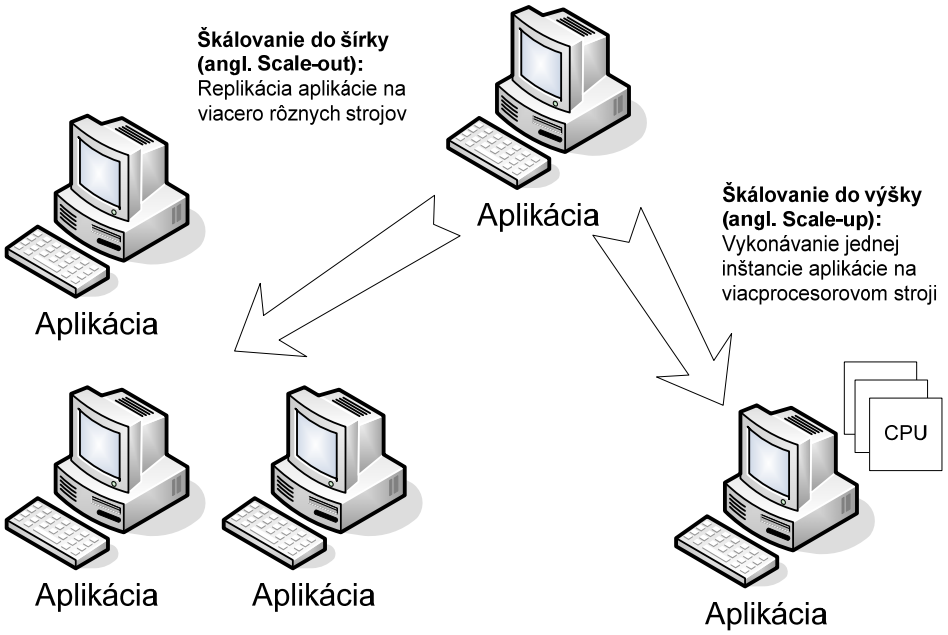
- záťaž požiadavkami (*request load*),
- počet súčasných spojení (*simultaneous connections*),
- veľkosť dát (*data size*),
- rozsah nasadenia (*deployment*).

Záťaž požiadavkami

Definujme transakčný mix pre serverovú aplikáciu na konkrétnej hardvérovej platforme, pomocou ktorého navrhneme architektúru spĺňajúcu požiadavku na priepustnosť 100 tps v špičkovej záťaži pri strednej dobe odozvy do jednej sekundy. Dokáže navrhnutá architektúra zvládnuť desaťnásobné zvýšenie záťaže?

V ideálnom prípade by daný systém bez dodatočného hardvéru mal pri zvýšení záťaže konštantnú priepustnosť 100 tps, pričom doba odozvy by sa lineárne zvýšila na 10 sekúnd. Škálovateľné riešenie v tomto prípade umožní nasadenie ďalších zdrojov

na zvýšenie priepustnosti a zníženie doby odozvy. Jedným zo spôsobov pridania zdrojov je pridať viac procesorov (CPU) a pamäte do stroja na ktorom je aplikácia nasadená – škálovať do výšky. Alternatívnym prístupom je distribuovať aplikáciu na viacero strojov – škálovať do šírky (obrázok 2-1).



Obrázok 2-1. Škálovanie do šírky a škálovanie do výšky.

Škálovanie do výšky funguje najlepšie vtedy, ak aplikácia je viacvláknová alebo v prípade, že viaceré jednovláknové inštancie aplikácie možno vykonávať súčasne na jednom stroji, čo však zvýši nároky na pamäť a dodatočné zdroje, keďže procesy predstavujú „drahý“ spôsob paralelizácie aplikácií.

Škálovanie do šírky je vhodné v prípade, ak správa a rozdelenie záťaže medzi viacero strojov vyžaduje len málo dodatočného úsilia. Cieľom je rovnomerné vyrovnávanie záťaže medzi jednotlivými strojmi (*load balancing*), pretože nevyužité stroje predstavujú zbytočne nakúpený hardvér.

Najdôležitejšou vlastnosťou škálovateľného riešenia je, že škálovateľnosť musí byť dosiahnutá bez zmien v architektúre riešenia, odhliadnuc od nutných zmien v konfigurácii aplikácie. V praxi sa však priepustnosť aplikácií s narastajúcou záťažou znižuje, čo vedie k exponenciálnemu nárastu doby odozvy. Príčinou je najmä súperenie o zdroje (CPU, pamäť) medzi jednotlivými procesmi a vláknami a spotrebovanie dodatočných zdrojov každou ďalšou požiadavkou, čo v konečnom dôsledku vedie k vyčerpaniu všetkých dostupných zdrojov obmedzeniu škálovateľnosti.

Počet súčasných spojení

Konkrétna architektúra môže byť navrhnutá pre 1000 súčasne pripojených používateľov. Keďže každé pripojenie spotrebúva zdroje systému, existuje limit na počet súčasných spojení, ktoré systém dokáže efektívne zvládnuť. V tomto prípade škálovateľný systém ideálne spotrebúva len málo zdrojov na udržanie spojenia.

Odstrašujúci príklad je prípad poskytovateľa pripojenia k Internetu (ISP), ktorý každému používateľovi ponúkal personalizovanú reklamu prostredníctvom samostatného procesu. Tento však mal značnú spotrebu pamäťových a výpočtových zdrojov aj keď používateľ nič nerobil. Testy rýchlo odhalili, že systém bol schopný zvládnuť len cca. 2000 spojení pred vyčerpaním pamäte, čo ISP neumožnilo získanie zamýšľaných 100 tisíc zákazníkov kvôli exponenciálne vysokej cene zdrojov. Aj napriek zúfalým pokusom o zmenu architektúry systému daný ISP skrachoval.

Veľkosť dát

Požiadavky na škálovateľnosť vzhľadom na veľkosť dát opisujú ako sa má aplikácia správať pri spracovaní narastajúceho množstva dát. Aplikácia sprostredkujúca posielanie správ, napr. diskusná miestnosť môže byť navrhnutá len na posielanie správ nejakej priemernej veľkosti. Ako bude architektúra reagovať ak sa výrazne zvýši veľkosť správ? Ako bude reagovať databázová aplikácia, ak sa výrazne zvýši veľkosť dát, resp. počet výsledkov na nejaký dopyt?

Rozsah nasadenia

Ako narastá úsilie nutné na nasadenie, údržbu a modifikáciu aplikácie pre narastajúcu používateľskú základňu? Do potrebného úsilia spadá aj úsilie vynaložené na distribúciu, konfiguráciu a aktualizáciu nových verzií. Ideálne riešenie bude podporovať automatické prostriedky na nasadenie a konfiguráciu aplikácie na distribuovanú množinu pracovných staníc spolu s licenčnými údajmi. Práve týmto spôsobom je v súčasnosti distribuovaný veľký počet aplikácií prostredníctvom Internetu.

Návrh škálovateľných riešení nie je jednoduchý, pretože požiadavky na škálovateľnosť často nie sú zjavné a ani špecifikované v rámci nefunkcionálnych požiadaviek na kvalitu. Schopný softvérový architekt musí zabezpečiť, že sa do návrhu nedostanú inherentne neškálovateľné prístupy, ktoré by v budúcnosti znemožnili škálovanie aplikácie s narastajúcimi požiadavkami. Ďalším problémom je náročnosť overenia požiadaviek na škálovateľnosť, ktoré je nepraktické z pohľadu potrebného času a nákladov. Práve z tohto dôvodu sa architekt musí spoľahnúť na praxou overené postupy a technológie.

Škálovateľnosť systému ICDE

Hlavnou požiadavkou na škálovateľnosť ICDE systému je podpora maximálneho očakávaného nasadenia, ktoré zodpovedá približne 150 používateľom. Keďže architektúra systému zabezpečuje, že každý používateľ má naraz najviac jednu vystávajúcu požiadavku, systém musí mať špičkovú priepustnosť na úrovni 150 súčasných požiadaviek od ICDE klientov.

2.1.3 Modifikovateľnosť

Každý schopný architekt vie, že v praxi bude každú aplikáciu skôr či neskôr nutné zmeniť tak, aby sa prispôsobila novým požiadavkám. Z tohto dôvodu je vhodné pri návrhu architektúry zohľadniť pravdepodobné budúce zmeny aplikácie. Čím flexibilnejší je návrh architektúry na začiatku, tým jednoduchšie a lacnejšie bude zavedenie budúcich zmien.

Modifikovateľnosť (*modifiability*) ako atribút kvality predstavuje mieru náročnosti zavedenia zmien do existujúceho návrhu – udáva ako náročné *môže byť* zapracovanie nových funkcionálnych a nefunkcionálnych požiadaviek. Dôležité je uvedomiť si, že

modifikovateľnosť predstavuje odhad budúceho úsilia a/alebo nákladov nutných na vykonanie zmeny. Skutočné náklady na zmenu vieme určiť len po tom, čo ju vykonáme – až vtedy je možné zistiť ako presný odhad sme vykonali.

Odhad modifikovateľnosti je možný len v kontexte konkrétneho architektonického riešenia, ktoré pozostáva minimálne zo štrukturálneho opisu komponentov, vzťahov medzi nimi a ich interakcie s prostredím. Odhad modifikovateľnosti vyžaduje identifikáciu očakávaných scenárov zmien a ďalšieho vývoja požiadaviek na systém. V niektorých prípadoch zmeny môžu byť vopred známe, či priamo uvedené v projektovom pláne pre ďalšie verzie systému. Vo väčšine prípadov však predpokladané zmeny musí odhadnúť architekt na základe informácií od zúčastnených strán a svojich predchádzajúcich skúseností.

Niektoré možné scenáre zmien sú:

- Podpora prístupu k aplikácii aj cez bezpečnostnú bránu (*firewall*) okrem pôvodného prístupu v rámci vnútornej siete.
- Dodávateľ COTS riešenia (napr. pre rozpoznávanie reči) skrachuje, čo si vyžiada zmenu príslušného komponentu v architektúre.
- Aplikácia musí byť prenesená z platformy Linux na Microsoft Windows.

Dopad zmien v jednotlivých scenároch na architektúru možno vopred odhadnúť (Gorton & Zhu, 2005). Tento odhad je však zriedkakedy jednoduchý, pretože zodpovedajúce riešenie zvyčajne nie je dostupné. Vo väčšine prípadov je jediné čo sa dá spraviť presvedčivá analýza dopadov zmien na jednotlivé komponenty architektúry systému alebo demonštrácia toho, že architektúru nebude nutné meniť.

Na základe odhadu ceny, rozsahu alebo úsilia nutného na vykonanie zmien komponentov, je možné vykonať odhad celkovej ceny vykonania zmeny. Zmeny v jednom komponente a zmeny vo voľne zviazaných komponentoch sú lacnejšie ako zmeny, ktoré vedú naprieč celou architektúrou. Identifikáciou pravdepodobnej zmeny, ktorá je príliš zložitá alebo drahá na realizáciu, môžeme objaviť chybu v návrhu architektúry, čo si môže vyžiadať ďalšiu analýzu a prípadnú zmenu návrhu.

Modifikovateľnosť systému ICDE

Požiadavky na modifikovateľnosť systému ICDE sú ťažko špecifikovateľné. Pravdepodobne bude nutné rozšíriť množinu systémom zaznamenávaných udalostí, čo ovplyvní ICDE klienta, ICDE server aj dátové úložisko.

Ďalšou pravdepodobnou zmenou je pridanie nových typov správ medzi nástrojmi tretích strán, čo bude mať dopad na mechanizmy výmeny správ v ICDE serveri.

2.1.4 Bezpečnosť

Bezpečnosť (*security*) v iformačných technológiách predstavuje komplexnú problematiku, ktorú možno na úrovni architektúry riešiť len na veľmi abstraktnej úrovni – je nutné presne pochopiť požiadavky na zabezpečenie aplikácie a navrhnúť mechanizmy, ktoré ich umožnia realizovať. Najčastejšie požiadavky na bezpečnosť sú:

- Autentikácia – overenie identity používateľov pracujúcich so systémom a aplikácií, ktoré s ním komunikujú.

- Autorizácia – pridelenie prístupových práv autentifikovaným používateľom a aplikáciám.
- Šifrovanie – šifrovanie správ posielaných do/z aplikácie.
- Integrita – zabezpečenie obsahu správ voči zmenám počas prenosu.
- Nepopierateľnosť – zabezpečenie potvrdenia prijatia správ adresátom pre prijímateľa a overenie identity odosielateľa prijímateľom tak, aby žiadny z nich nemohol popierať svoju účasť na výmene správ.

V praxi sa široko používajú viaceré technológie a techniky, ktoré umožňujú zabezpečenie požiadaviek na bezpečnosť. Infraštruktúra verejného kľúča – PKI (*Public Key Infrastructure*) a šifrovanie pomocou SSL (*Secure Sockets Layer*) sa v praxi často používajú na zabezpečenie autentifikácie, šifrovania a nepopierateľnosti komunikácie. Autentifikácia a autorizácia je v Jave podporovaná službou JAAS (*Java Authentication and Authorization Service*), operačné a databázové systémy poskytujú zabezpečenie prístupu prostredníctvom prihlásenie sa pomocou hesla. Návrhári pre .NET s obľubou využívajú bezpečnostné služby operačného systému Windows, Java aplikácie využívajú JAAS, pričom databázové systémy zvyčajne presadzujú vlastný bezpečnostný model. Prehľad jednotlivých techník obsahuje práca (Ramachandran, 2002).

V praxi existuje veľké množstvo dostupných riešení, ktoré možno ľahko použiť v jednej konkrétnej bezpečnostnej doméne. Ak však aplikácia obsahuje viacero komponentov, ktoré chcú riadiť bezpečnosť, je nutné navrhnúť vhodné riešenie, ktoré zvyčajne izoluje riadenie bezpečnosti do jedného komponentu. Tento následne využíva najvhodnejšiu technológiu na splnenie požiadaviek na bezpečnosť.

Bezpečnosť systému ICDE

Hlavnou požiadavkou na bezpečnosť je autentifikácia používateľov a nástrojov tretích strán. Vo verzii 1.0 používatelia používajú na autentifikáciu používateľské meno a heslo overené databázou, ktorá im následne umožní prístup k ich údajom. V ICDE verzie 2.0 bude nutné zahrnúť podporu pre autentifikáciu aplikácií tretích strán. Tieto budú vykonávané buď lokálne alebo na vzdialenom stroji prostredníctvom nezabezpečenej siete, čo si vyžiada šifrovanie prenášaných údajov.

2.1.5 Dostupnosť

Dostupnosť (*availability*) bezprostredne súvisí so spoľahlivosťou aplikácie (*reliability*) a je pomerne ľahko špecifikovateľná a merateľná. Ak aplikácia nie je dostupná v čase, keď je to potrebné, tak pravdepodobne nespĺňa svoje funkcionálne požiadavky. Mnoho aplikácií musí byť dostupných minimálne počas úradných hodín. Väčšina internetových aplikácií vyžaduje stopercentnú dostupnosť, keďže Internet za bežných okolností nepozná úradné hodiny. Dostupnosť možno merať ako podiel času, ktorý je daná aplikácia dostupná k času, ktorý by mala byť dostupná.

Poruchy aplikácie spôsobujú jej nedostupnosť a ovplyvňujú jej spoľahlivosť, ktorú meriame v strednej dobe medzi poruchami. Doba nedostupnosti aplikácie je daná časom, ktorý je potrebný na identifikáciu poruchy a obnovenie systému. Z tohto dôvodu sa aplikácie, ktoré vyžadujú vysokú dostupnosť snažia minimalizovať, resp. eliminovať možné miesta porúch a využívajú mechanizmy na automatickú detekciu porúch a obnovu systému, resp. komponentov, ktoré zlyhali.

Replikácia komponentov je overený spôsob na zvýšenie dostupnosti, pričom v prípade zlyhania komponentu aplikácia môže pokračovať v činnosti pomocou ostatných funkčných komponentov. Porucha komponentu potom zvyčajne znamená zníženie výkonnosti systému, avšak neovplyvňuje celkovú dostupnosť systému.

Úzky súvis s dostupnosťou má tiež obnoviteľnosť (*recoverability*) systému – aplikácia je obnoviteľná, ak je schopná obnoviť požadovanú úroveň výkonnosti a obnoviť údaje v prípade poruchy aplikácie alebo systému. Ukážkovým príkladom obnoviteľnej aplikácie je databázový systém. V prípade poruchy je nedostupný až do jej obnovy, ktorá vyžaduje reštart servera a vyriešenie stavu všetkých transakcií, ktoré prebiehali v momente nastania poruchy. Dôležitú úlohu zohráva (automatický) spôsob detekcie porúch a následný proces obnovy ako aj celkový čas nutný na obnovu systému. Keďže počas obnovy je systém stále nedostupný, je stredná doba obnovy významnou metrikou pre meranie celkovej dostupnosti systému.

Dostupnosť ICDE systému

Aj keď je dostupnosť systému ICDE želateľná, je nevyhnutná len počas úradných hodín organizácie, kde je systém nasadený, čo dáva dostatočný priestor na údržbu a zálohovanie systému. Riešenie by však malo obsahovať mechanizmy na zabezpečenie takmer stopercentnej dostupnosti počas úradných hodín, napr. pomocou replikácie komponentov.

2.1.6 Integrovateľnosť

Jednoduchosť prepojenia aplikácie s inými aplikáciami v rámci širšieho aplikačného kontextu opisuje integrovateľnosť (*integration*). Hodnota aplikácie alebo komponentu je často nepomerne väčšia, pokiaľ jeho funkcionalitu alebo dáta možno použiť aj spôsobom, ktorý jeho tvorca nepredpokladal. Medzi najrozšírenejšie spôsoby integrácie patria dátová integrácia a programové rozhrania API (obrázok 2-2).

Dátová integrácia vyžaduje ukladanie údajov spôsobom, ktorý umožňuje prístup k údajom aj iným aplikáciám. Jednoduchým spôsobom dátovej integrácie je použitie spoločnej databázy, resp. vytvorenie mechanizmov na prácu s dátami v známom formáte, napr. XML alebo CSV, ktorý sú schopné spracovať aj iné aplikácie.

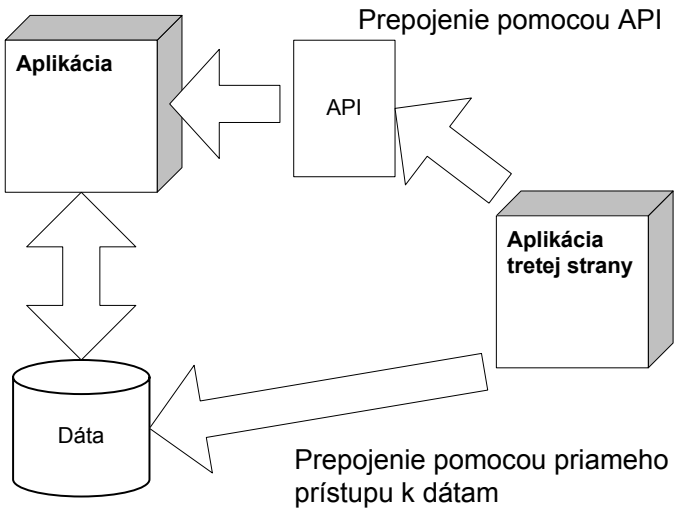
Dátová integrácia neposkytuje prostriedky na kontrolu prístupu iných aplikácií k údajom ani kontrolu ich využitia a zneužitia, pretože táto je mimo dosah pôvodnej aplikácie. Alternatívny prístup predstavuje použitie programového rozhrania API, ktoré skrýva údaje pred ostatnými aplikáciami, ktoré k nim majú prístup len prostredníctvom funkcií daného API. Použitie API umožňuje definovanie a aplikovanie pravidiel prístupu k dátam ako aj zabezpečenie bezpečnosti pri prístupe k dátam.

Výber konkrétneho spôsobu integrácie nebýva jednoznačný – dátová integrácia je jednoduchá a flexibilná, zatiaľ čo integrácia pomocou API poskytuje kontrolované prostredie na prístup k dátam, ale vyžaduje väčšie množstvo úsilia. Použitie API je tiež podstatne robustnejšie z hľadiska integrácie, pretože klienty API sú izolované od zmien v interných dátových štruktúrach.

Integrácia systému ICDE

Požiadavky na integráciu systému ICDE vychádzajú z nutnosti podpory analytických nástrojov tretích strán, ktorá vyžaduje dobre definovaný spôsob prístupu k dátovému úložisku ICDE systému. Keďže nástroje tretích strán budú často pracovať vzdialene je

dátová integrácia nepravdepodobná. Následne bude integrácia pravdepodobne realizovaná prostredníctvom špecializovaného API rozhrania.



Obrázok 2-2. Možnosti integrácie.

2.1.7 Ostatné atribúty kvality

Existuje veľký počet ďalších atribútov kvality, ktoré sú relevantné pre rôzne aplikácie, pričom uvádzame len niektoré:

- Prenosnosť (*portability*) opisuje množstvo úsilia, nutného na prenesenie riešenia na inú hardvérovú/softvérovú platformu ako pre ktorú bolo pôvodne vytvorené. Prenosnosť závisí od softvérových technológií použitých k implementácii riešenia a vlastností platforiem na ktorých má byť riešenie nasadené. Jednoducho prenosné zdrojové kódy izolujú platformovo závislé časti do oddelených komponentov, ktoré možno jednoducho nahradiť bez ovplyvnenia zvyšku systému.
- Testovateľnosť (*testability*) opisuje ako ľahké alebo ťažké je testovať aplikáciu. Počiatočné rozhodnutia pri návrhu môžu mať veľký vplyv na počet potrebných testovacích vzoriek. Základné pravidlo hovorí, že čím je návrh zložitejší, tým náročnejšie bude ho podrobne otestovať. Jednoduchý návrh, resp. návrh využívajúci už otestované komponenty znižuje požiadavky na testovanie.
- Podporovateľnosť (*supportability*) je miera jednoduchosti podpory aplikácie po jej nasadení. Podpora v praxi zahŕňa diagnózu a opravu problémov, ktoré nastanú počas prevádzky systému. Dobre podporovateľné systémy obsahujú explicitné prostriedky na diagnostiku chýb, ako sú záznamy chybových hlásení s opisom porúch a ich príčin. Taktiež majú zvyčajne modulárny návrh, ktorý umožňuje opravu a opätovné nasadenie vybraných komponentov bez zásadného vplyvu na fungovanie aplikácie.

2.1.8 Zhrnutie

Keby život softvérového architekta bol jednoduchý, tak by návrh architektúry vyžadoval len použitie vhodných princípov a mechanizmov, ktoré by zabezpečili

naplnenie požiadaviek na atribúty kvality – vyber atribút a pridaj mechanizmus na jeho splnenie. Praktické riešenia však nie sú priamočiare, pretože atribúty kvality nie sú od seba nezávislé – môžu sa navzájom ovplyvňovať rozličnými skrytými spôsobmi. Návrh, ktorý spĺňa požiadavky na jeden atribút kvality môže mať negatívny dopad na iný atribút kvality. Napr. vysoko zabezpečený systém môže byť veľmi ťažké až nemožné integrovať do otvoreného prostredia. Vysoko dostupná aplikácia môže vymeniť nižšiu výkonnosť za vyššiu dostupnosť.

Pochopenie vplyvov a kompromisov medzi jednotlivými požiadavkami na atribúty kvality a návrh riešenia, ktoré predstavuje zmysluplný kompromis je jednou z najťažších úloh softvérového architekta. Nie je jednoducho možné nájsť riešenie, ktoré v plnej miere spĺňa úplne všetky protichodné požiadavky a je práve úlohou architekta identifikovať problémové miesta a scenáre, definovať priority a dokumentovať jednotlivé rozhodnutia pri návrhu.

Pochopenie a identifikácia požiadaviek na kvalitu predstavuje však len nutný predpoklad na návrh vhodného riešenia. Pozícia architekta je veľmi komplikovaná aj preto, že architekt predstavuje styčný bod viacerých zúčastnených strán a musí nielen podrobne rozumieť atribútom kvality a použitým technológiám ale aj komunikovať s ostatnými účastníkmi. Architekt musí u zákazníka podrobne identifikovať požiadavky na kvalitu, pretože tieto často nie sú dostatočne zachytené v špecifikácii požiadaviek na systém. Súčasne je úlohou architekta diskutovať tolerancie v návrhu, identifikovať podmienky, za ktorých možno znížiť požiadavky na kvalitu a jasne predostrieť kompromisy jednotlivým zúčastneným stranám tak, aby pochopili „do čoho idú“.

2.2 Architektúry a technológie pre spojovací softvér

Napriek tomu, že medzi stavebnou architektúrou a architektúrou softvéru existujú významné rozdiely, je možné aj v tejto nedokonalej analógii nájsť odpoveď na to, akú úlohu zohráva spojovací softvér (*middleware*) v architektúre softvéru.

Keď stavebný architekti navrhujú budovu, vytvárajú nákresy, ktoré v zásade zobrazujú vlastnosti budovy z rôznych uhlov pohľadu. Návrh pritom zohľadňuje požiadavky ako napríklad veľkosť priestorov, funkciu (kancelárske priestory, rodinný dom, nákupné centrum), estetické vlastnosti a rozpočet. Tieto nákresy sú abstraktnou reprezentáciou požadovanej a konkrétnej, aj keď ešte nepostavenej budovy.

Je úplne zrejmé, že samotné architektonické nákresy nestačia na to, aby sa budova začala naozaj stavať. Architektonický návrh je potrebné rozšíriť o detailné nákresy stien, rozvrhnutia poschodí, schodiská, elektrické zariadenia, vodovody, štruktúrovanú kabeláž, atď. Počas navrhovania týchto prvkov sa súčasne vyberajú aj vhodné materiály a komponenty, pomocou ktorých sa konštruujú. Tieto materiály a komponenty sú základnými stavebnými prvkami budov a boli vytvorené tak, aby dokázali spĺňať základnú funkcionalitu v mnohých typoch budov, či už pôjde o kancelárske budovy, železničné stanice alebo len skromné rodinné domy.

Aj keď to možno nie je najlepšou analógiou, spojovací softvér je softvérovej architektúre tým, čím sú kabeláž, rozvody a potrubia v stavebnej architektúre. Dôvodov je hneď niekoľko:

- Spojovací softvér poskytuje overené spôsoby prepojenia rôznych softvérových komponentov v aplikáciách tak, aby výmena dát medzi nimi prebiehala cez

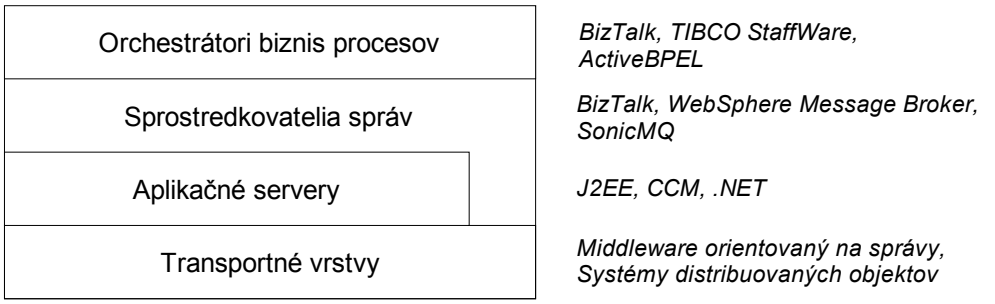
pomerne jednoducho pochopiteľné mechanizmy. Spojovací softvér je teda potrubím pre dáta medzi komponentmi, pričom môže byť využitý v mnohých aplikačných doménach.

- Spojovací softvér je možné využiť na prepojenie viacerých komponentov do vhodných a zaužívaných topológií. Prepojenia môžu byť 1-1, 1-N alebo M-N.
- Z pohľadu používateľa je spojovací softvér úplne skrytý. Používatelia pracujú s aplikáciou a nezaujímajú sa o to ako je zabezpečená vnútorná komunikácia. Pokiaľ všetko funguje tak ako má, spojovací softvér je neviditeľnou infraštruktúrou.

2.2.1 Klasifikácia technológií

Názov spojovací softvér (*middleware*) pochádza z pôvodného využitia ako strednej vrstvy medzi aplikáciami a operačným systémom (preto sa niekedy používa aj pojem *stredná vrstva*). V realite sa však spojovací softvér postupne stal oveľa komplikovanejším ako jednoduchá vrstva oddeľujúca služby operačného systému od aplikácie.

Rôzne aplikačné domény zvyknú za spojovací softvér považovať rôzne technológie. V nasledujúcom texte sa zameriame na typické technológie pre spojovací softvér, ktoré sa využívajú v IT priemysle. Obrázok 2-3 zobrazuje klasifikáciu týchto technológií spolu s názvami najznámejších predstaviteľov týchto technológií.



Obrázok 2-3. Klasifikácia technológií pre spojovací softvér.

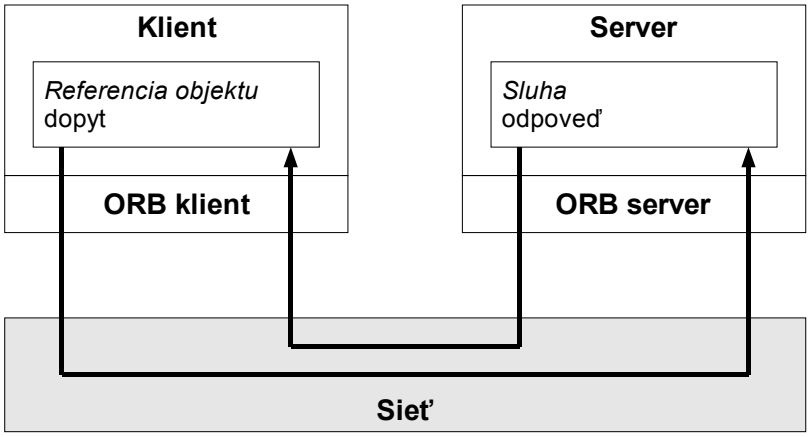
Každá z kategórií bude detailnejšie opísaná v samostatných podkapitolách a preto ich na tomto mieste opisujeme len v krátkosti.

- *Transportné vrstvy* reprezentujú základné komunikačné kanály na zasielanie požiadaviek a prenos údajov medzi softvérovými komponentmi. Tieto komunikačné kanály poskytujú jednoduché mechanizmy, ktoré umožňujú priamočiaru výmenu údajov aj v distribuovaných aplikáciách.
- *Aplikačné servery* sú väčšinou postavené na základných transportných službách, avšak podporujú aj transakcie, bezpečnosť a adresárové služby, za pomoci ktorých je možné vytvárať viacvláknové serverové aplikácie.
- *Sprostredkovatelia správ* využívajú buď základné služby transportných vrstiev alebo aj aplikačných serverov, pričom pridávajú špecializované prostriedky na spracovanie správ. Tieto prostriedky umožňujú rýchle transformácie správ a vysokoúrovňové programovacie techniky na špecifikáciu toho, ako bude zabezpečená výmena, manipulácia ako aj smerovanie správ medzi rôznymi komponentmi.

- *Orchestrátori biznis procesov* (*Business Process Orchestrator*, BPO) obohacujú sprostredkovateľov správ o podporu aplikácií zameraných na pracovné procesy. V takýchto aplikáciách môžu trvať procesy rádovo hodiny až dni. BPO poskytujú nástroje na opis, vykonanie ako aj manažment takýchto procesov.

2.2.2 Distribuované objekty

Technológie distribuovaných objektov sú dlhodobým členom riešení pre spojovací softvér. Asi najznámejším z nich je CORBA (*Common Object Request Broker Architecture*) ako predstaviteľ distribuovaného objektovo orientovaného spojovacieho softvéru, ktorý sa používal na začiatku deväťdesiatych rokov.



Obrázok 2-4. Použitie distribuovaných objektov v CORBA.

Obrázok 2-4 znázorňuje jednoduchý scenár použitia CORBA pre zaslanie požiadavky klientom na server za pomoci sprostredkovateľa požiadaviek pre objekty (*Object Request Broker*, ORB). Sluhovia – objekty, ktoré požiadavky na serveri spracovávajú, musia implementovať rozhranie, ktoré je najskôr zadefinované pomocou opisného jazyka CORBA IDL (*Interface Description Language*). Kostra takto zadefinovaného objektu je následne automaticky vygenerovaná do cieľového jazyka. Táto kostra obsahuje mechanizmy, ktoré zabezpečujú komunikáciu medzi vzdialeným sluhom a referenciou objektu, ktorý ho neskôr zavolá. Programátor samozrejme musí implementovať kód samotného sluhu, ktorý sa vykoná na serveri.

Z pohľadu používateľa sa takýto vzdialený objekt používa úplne rovnako ako objekt lokálny. Spojovací softvér zabezpečuje odchytenie volania metódy, transformáciu do formátu vhodného na prenos, samotný prenos, spracovanie na serveri a spätné zaslanie výsledku klientovi. Všetko sa udeje bez toho, aby klientska strana vedela, kde a ako sa samotná požiadavka vykoná. Toto je však len veľmi zjednodušený pohľad na problematiku technológie distribuovaných objektov. Z pohľadu softvérového architekta je dôležité si uvedomiť, aké sú základné vlastnosti a dôsledky využitia takéhoto riešenia v reálnych aplikáciách.

- Požiadavky na sluhov sú volania na vzdialené objekty a môžu byť pomalé kvôli rýchlostiam transformácií a prenosu po sieti. Je preto vhodné takéto volania čo najviac minimalizovať.

- Ako v každej distribuovanej aplikácii, servery môžu byť dočasne či dokonca permanentne nedostupné kvôli hardvérovým alebo softvérovým problémom. Aplikácie musia nutne počítať s takýmito výpadkami a zvoliť vhodnú stratégiu zotavenia sa z týchto situácií.
- Ak sluha na serveri udržiava údaje o stave klienta je možné, že ak sluha, server či sieť zlyhá sa tento stav stratí. Je preto nutné navrhnúť mechanizmy, ktoré zabezpečia obnovu stratených stavov.

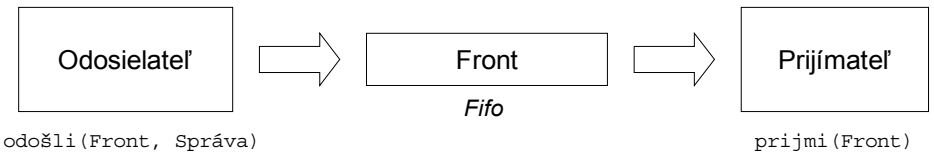
2.2.3 Spojovací softvér orientovaný na správy

Spojovací softvér orientovaný na správy (*Message-Oriented Middleware*, MOM) je kľúčovou technológiou pri stavbe veľkých systémov. Je spojovacím článkom inak nezávislých a samostatných aplikácií, ktorý ich zoskupuje do jedného integrovaného celku. Tieto aplikácie môžu byť navyše vytvorené rôznymi technológiami a nasadené na rôznych platformách.

Základy MOM

Základom MOM je pridanie jednej vrstvy medzi odosielateľa a prijímateľa vo forme frontu správ. MOM technológie sa tak stávajú voľne zviazanými a asynchrónnymi. Synchrónne technológie síce majú veľa silných stránok, no môžu byť viac náchylné na chybovosť pretože vyžadujú, aby všetky komponenty a sieťové spojenia súčasne pracovali správne.

Keďže odosielateľ a prijímateľ správ sú rozdelený frontom správ, odosielateľ môže zaslať správu prijímateľovi pričom je zabezpečené, že táto správa bude v konečnom dôsledku doručená, napriek tomu, že sieťové spojenie či samotný prijímateľ sú dočasne nedostupní. Odosielatelia teda jednoducho zasielajú správy na MOM server a pokračujú vo svojej práci, pričom ani nevedia, ktorý proces či aplikácia požiadavku spracuje. Obrázok 2-5 znázorňuje základný mechanizmus MOM.



Obrázok 2-5. Základ MOM.

MOM server zvyčajne plní niekoľko základných funkcií. Musí zabezpečiť príjem správy od odosielateľa a keďže takýchto správ môže byť niekoľko za sebou, je nutné zabezpečiť ich skladovanie pre neskoršie zaslanie prijímateľovi. Taktiež musí zabezpečiť zaslanie notifikácie odosielateľovi o tom, že správu sa podarilo doručiť.

Takýto asynchrónny a rozdelený prenos správ možno veľmi dobre využiť na niektoré typické problémy spojené s návrhom aplikácií.

- Ak odosielateľ nepotrebuje žiadnu odpoveď na zaslanú správu, tak môže jednoducho pokračovať vo svojej práci ďalej. Takáto forma zasielania správ sa nazýva „pošli a zabudni“ (s*end-and-forget*).

- Ak odosielateľ nepotrebuje okamžitú odpoveď na zaslanú správu. Prijímateľ správy môže bežne potrebovať na jej spracovanie aj niekoľko minút a odosielateľ správy môže zatiaľ namiesto zbytočného čakania pracovať ďalej.
- Prijímateľ alebo spojenie medzi odosielateľom a prijímateľom nemusí pracovať kontinuálne. Odosielateľ sa spolieha na MOM server, ktorý zabezpečuje doručenie správy pri najbližšom spojení s prijímateľom.

Úrovne zabezpečenia doručenia správ

V rôznych aplikáciách je nutné zabezpečiť rôzne kvalitatívne úrovne zabezpečenia doručenia správ. V bankovom sektore môže strata transakcie viesť k veľkej finančnej strate, no v iných aplikáciách môžu byť takéto straty správ tolerované napríklad kvôli získanému vyššiemu výkonu. Typicky MOM servery delíme podľa úrovne zabezpečenia doručenia správ na tri hlavné skupiny:

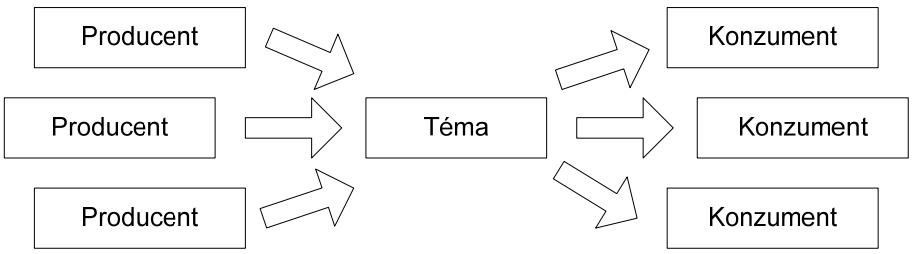
- *Najväčšia snaha* (*best effort*) je úroveň najnižšou úrovňou zabezpečenia. Nedoručené správy sa ukladajú iba do hlavnej pamäte, čo znamená, že ak server zlyhá, tieto správy sa stratia. Výpadky siete alebo nedostupní prijímatelia správ môžu taktiež spôsobiť, že správy sa odstránia z čakacieho frontu kvôli prekročenému časovému limitu.
- *Perzistentná úroveň* zabezpečuje odoslanie správ aj napriek systémovým a sieťovým zlyhaniam. Nedoručené správy sú ukladané na pevné disky, z ktorých sa vymazávajú až po úspešnom doručení.
- *Transakčná úroveň* je nadstavbou perzistentnej úrovne zabezpečenia doručenia správ a umožňuje zasielať skupiny správ, ktoré sa buď doručia ako celok alebo sa nedoručia vôbec. Zabezpečenie takejto nedeliteľnosti doručenia správ je potrebné pri zložitejších systémoch.

Producent-konzument

MOM je zaužívaným a dobre overeným spôsobom na stavbu voľne zviazaných veľkých systémov. Aj MOM má však svoje limitácie a jednou z hlavných je to, že odosielateľ môže naraz zaslať správu iba jednému prijímateľovi.

Obrázok 2-6 znázorňuje zasielanie správ schémou producent-konzument (*publish-subscribe*), ktorá je rozšírením MOM mechanizmov o podporu komunikácie v topológiách 1-N, M-N a N-1.

Producenty zasielajú správy na témy, ktoré sú ekvivalentom frontov v MOM serveroch, pričom konzumenty správy z týchto tém odberajú. Server zabezpečujúci takúto distribúciu správ potom zasiela správy v témach každému konzumentovi danej témy.



Obrázok 2-6. Zasielanie správ schémou producent-konzument.

Z hľadiska zviazanosti má takáto forma rozosielania správ niekoľko výhodných vlastností. Producenty ako aj konzumenty nevedia, kto správy na témy publikuje, ako aj to, kto tieto témy odberá. Každá téma tak môže mať hneď niekoľko producentov, pričom producenty sa môžu pridávať a odoberať dynamicky. Taktiež konzumenty tém sa môžu meniť úplne dynamicky.

MOM a softvérová architektúra

Z hľadiska softvérového architekta je využitie MOM technológií vo veľkých aplikáciách pomerne bežné. Existuje aj množstvo pragmatických dôvodov prečo môžu byť takéto technológie vhodnejšie ako klasické synchrónne technológie:

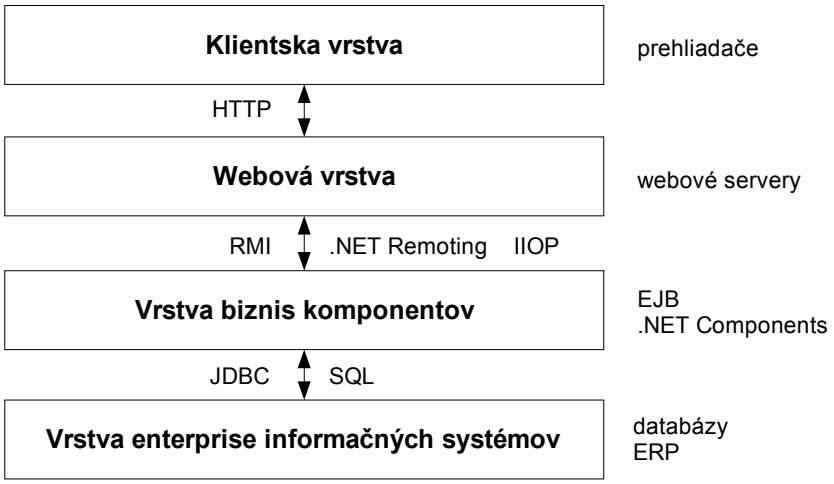
- Zavedenie MOM technológií do existujúcich projektov je pomerne jednoduché, lacné a bezpečné. Aplikácie nie je potrebné príliš modifikovať, aby dokázali komunikovať formou MOM servera.
- Je dostupné široké spektrum hotových MOM technológií, čo umožňuje lepší výber najvhodnejšieho variantu a taktiež zjednodušuje integráciu s už existujúcimi systémami.
- Organizácie, ktoré zakúpili a získali skúsenosti s MOM technológiou niekedy nemusia potrebovať dodatočnú funkcionalitu aplikačných serverov.

2.2.4 Aplikačné servery

Existuje mnoho rôznych definícií aplikačných serverov, no skoro všetky sa v zásadných prvkoch zhodujú. Aplikačný server teda môžeme vo všeobecnosti považovať za komponentovú serverovú technológiu, ktorá je medzivrstvou vo viacvrstvovej architektúre a jej úlohou je zabezpečenie distribuovanej komunikácie, bezpečnosti, transakčného manažmentu a perzistencie.

Aplikačné servery sa používajú najmä na stavbu internetových aplikácii. Obrázok 2-7 znázorňuje blokovú schému viacvrstvovej architektúry využívanej v množstve webových stránok.

- *Klientskou vrstvou* vo webových aplikáciách je typicky webový prehliadač, ktorý zasiela HTTP požiadavky a sťahuje HTML stránky z webového servera. Táto vrstva však nie je priamou súčasťou aplikačného servera.
- *Webová vrstva* zabezpečuje spracovanie klientskych požiadaviek. Webový server po príchode požiadavky zavolá serverový komponent (JSP alebo ASP), ktorý spracuje parametre požiadavky a pomocou biznis logiky požiadavku vykoná. Komponent následne upraví výstup do HTML formy a ten zašle naspäť používateľovi.
- *Vrstva biznis komponentov* je jadrom celej aplikácie. Biznis komponenty môžu byť realizované napríklad pomocou *Enterprise Java Beans* (EJB) v J2EE, ako .NET komponenty či CORBA objekty. Spúšťanie týchto biznis komponentov väčšinou zastrešuje kontajner, ktorý zabezpečuje transakčný manažment, správu životných cyklov komponentov, správu stavov, bezpečnosť, viacvláknovosť a zdieľanie zdrojov. Programátor tak nemusí prepletať biznis logiku s kódom, ktorý už zabezpečuje samotný kontajner komponentov.
- *Vrstva podnikových informačných systémov* bežne pozostáva z jednej alebo viacerých databáz, na ktoré sa dopytujú biznis komponenty.



Obrázok 2-7. Viacvrstvová architektúra pre webové aplikácie.

Jadrom aplikačného servera je práve kontajner biznis komponentov a služby, ktoré ponúka pre implementáciu biznis logiky. Keďže jednotlivé aplikačné servery sa líšia v detailoch, v ďalšom texte sa zameriavame len na EJB model J2EE ako reprezentanta technológie aplikačných serverov.

Enterprise Java Beans

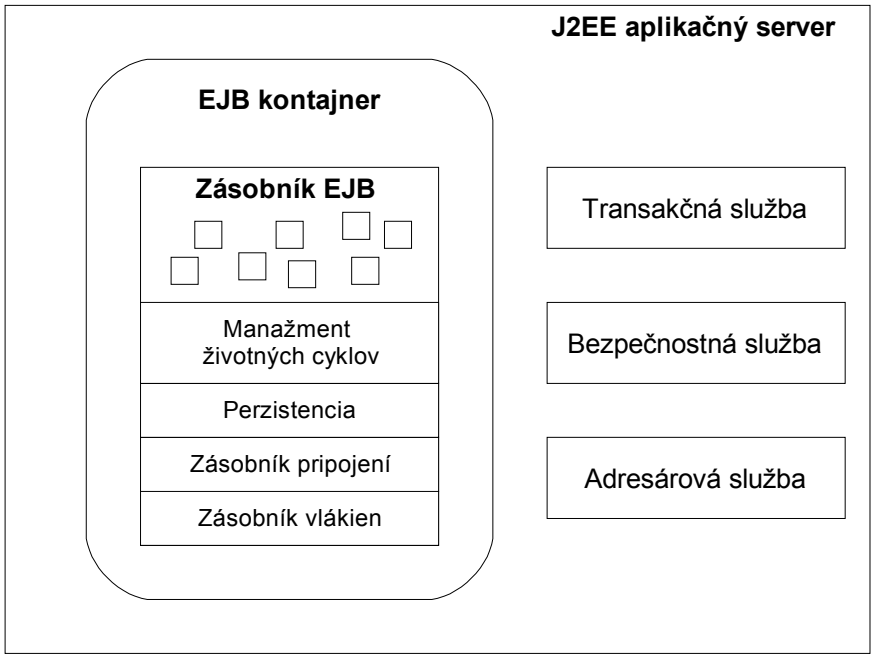
Architektúra EJB definuje štandardný model pre stavbu serverových Java aplikácií. Aplikačný server, ktorý je kompatibilný s J2EE poskytuje EJB kontajner na spravovanie vykonávania aplikačných komponentov. Z praktického hľadiska je kontajner operačným systémom pre EJB komponenty bežiacim na virtuálnom stroji. Obrázok 2-8 zobrazuje vzťah medzi aplikačným serverom, kontajnerom a službami, ktoré poskytuje.

Komponentový model EJB

Komponentový model EJB určuje základnú architektúru EJB komponentov, ich štruktúru a spôsoby, ktorými komunikujú so samotným kontajnerom ako aj ostatnými komponentmi.

Špecifikácia EJB verzie 1.1 definuje dva hlavné typy EJB komponentov – *session beans* a *entity beans*. *Session beans* bežne slúžia na vykonávanie biznis logiky a poskytovanie služieb klientom cez verejné rozhrania. *Entity beans* sa používajú na reprezentáciu dátových objektov, ktoré sú uložené v databázach. E*ntity beans* sú zvyčajne využívane v biznis logike obsiahnutej v *session beans*, ktoré ich sprístupňujú klientskej vrstve.

Session beans môžu byť navyše bezstavové (*stateless*) alebo so stavom (*statefull*). Ako je zo samotného názvu zrejmé bezstavové komponenty nemôžu obsahovať žiadne dáta spojené so stavom klienta, ktorý komponent využíva. Môže sa teda stať, že kontajner ponúkne v dvoch volaniach tomu istému klientovi dve rôzne inštancie komponentov. Naopak pri komponentoch so stavom je garantované, že klient komunikuje vždy s tou istou inštanciou komponentu a je preto možné do tohto komponentu odkladať údaje o klientovi.



Obrázok 2-8. J2EE aplikačný server, EJB kontajner a pomocné služby.

EJB kontajner je zodpovedný za správu životných cyklov takýchto komponentov. Po určitej dobe nečinnosti môže kontajner dokonca zapísať stav komponentu na disk a potom ho pri ďalšom volaní automaticky načítať. Toto je známe ako pasivácia a aktivácia komponentu so stavom. Kontajner môže byť nakonfigurovaný aj tak, že komponenty, ktoré neboli určitú dobu používané sa zničia – uvoľnia miesto v pamäti.

Entity beans môžu byť z hľadiska perzistencie taktiež dvoch typov – spravované kontajnerom (*container managed persistence*, CMP) a spravované komponentom (*bean managed persistence*, BMP). Perzistencia komponentov spravovaná kontajnerom znamená, že kontajner je zodpovedný za načítanie, ukladanie zmien údajov komponentu do databázy a to vo vhodnom okamihu, ako je napríklad začiatok či koniec transakcie. V prípade perzistencie spravovanej komponentom musí sám programátor zabezpečiť ukladanie údajov komponentu do databázy, čo je väčšinou možné pomocou manuálne vyprodukovaných JDBC volaní a SQL dopytov do databázy. Takéto ručné vytváranie je na jednej strane komplikáciou pre programátora komponentu, no na druhej strane umožňuje optimalizácie, ktoré na vyššej úrovni nemusia byť vôbec možné.

Zodpovednosti EBJ kontajneru

Keďže EJB kontajner je pomerne zložitý softvér, je vhodné zdôrazniť akú úlohu zohráva kontajner v EJB aplikáciách. Vo všeobecnosti poskytuje kontajner tieto služby EJB komponentom:

- Zabezpečuje správu životných cyklov a zdieľanie komponentov, ich vytváranie, aktiváciu, pasiváciu a deštrukciu.
- Zachytáva volania na vzdialených objektoch a zabezpečuje transakčný manažment a bezpečnosť.

- Zabezpečuje perzistenciu zvolených atribútov komponentov spravovaných kontajnerom.
- Odbremeňuje programátora od problematiky viacvláknových a vysoko výkonnostných systémov tým, že optimalizácie a alokácie vlákien vykonáva transparentne.
- Optimalizuje prístup k *entity beans* pomocou vyrovnávacích pamätí.
- Zabezpečuje a spravuje zásobník zdieľaných spojení s databázami z dôvodu ich efektívnejšieho využitia.

Aplikačný server z pohľadu softvérového architekta

Aplikačné servery sú komplexná technológie a je preto zrejmé, že softvérový architekt musí veľmi dobre poznať konkrétne požiadavky na systém, aby dokázal zvoliť vhodný typ použitého aplikačného servera. Technológie aplikačných serverov však umožňujú vyššiu voľnosť pri využívaní rôznych architektonických vzorov. Komplexnosť komunikácie s komponentmi v aplikačnom serveri zasa môže byť vzhľadom na výkonnosť značne limitujúcim faktorom použitia takéhoto riešenia.

2.2.5 Sprostredkovatelia správ

Bežné systémy zasielania správ postačujú v mnohých aplikáciách. V skutočnosti však nastáva problém pri integrácii viacerých systémov, ktoré nemajú jednoznačne určený spoločný výmenný formát údajov. Tento problém sa snaží adresovať sprostredkovateľ správ ako medzivrstva v sústave viacerých systémov. Sprostredkovateľ správ transformuje vstupný formát správy z komponentu do viacerých výstupných formátov požadovaných inými systémami (Obrázok 2-9).

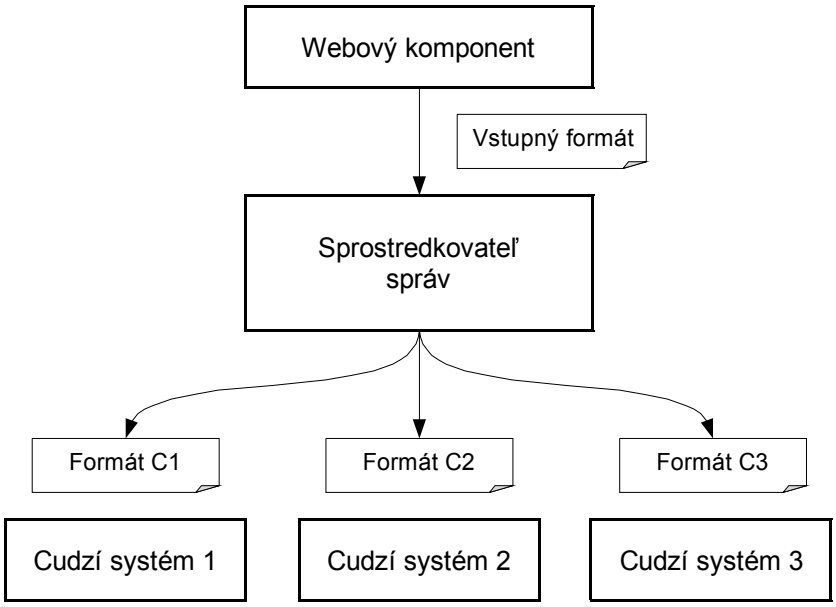
Napriek tomu, že idea sprostredkovateľa správ je pomerne jednoduchá, je potrebné si uvedomiť, že riešenia sprostredkovateľov správ sa môžu veľmi rýchlo stať úzkym hrdlom celého systému. Riešenia takýchto transformátorov správ musia byť teda veľmi dobre škálovateľné.

Definícia transformácií medzi formátmi je miestom, kde technológie sprostredkovateľov správ naozaj vynikajú. Existujú veľmi intuitívne grafické rozhrania na špecifikáciu transformácii medzi formátmi, pričom vhodnou kombináciou elementárnych a používateľom nastaviteľných prvkov je možné opísať aj komplexné transformačné mechanizmy.

Z hľadiska softvérové architekta treba dobre zvážiť nevýhody takéhoto pomerne drahého a väčšinou proprietárneho riešenia voči výhodám, ktoré ponúka.

2.2.6 Orchestrátori biznis procesov

Biznis procesy v moderných aplikáciách sú čoraz komplikovanejšie, no už aj bežná objednávka tovaru je procesom, ktorý môže trvať rádovo aj niekoľko týždňov. Takéto dlhé transakcie prinášajú nové problémy, ktoré je potrebné zvládnuť. Je nutné udržiavať aktuálny stav pretrvávajúcich transakcii, čo vzhľadom na ich dĺžku môže znamenať pomerne veľké množstvo uchovávaných údajov.



Obrázok 2-9. Schéma funkcionality sprostredkovateľa správ.

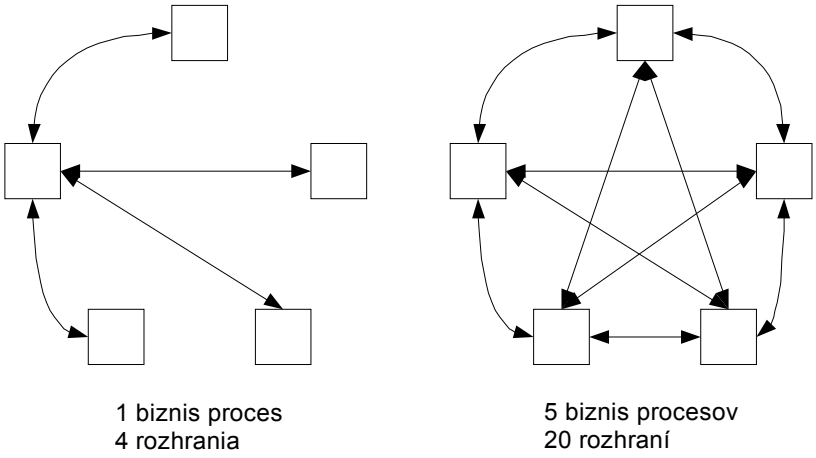
Použitie štandardných ACID transakcií, ktoré zmrazia všetky zdroje, ktorých sa transakcia dotýka je skoro nemožné. Veľmi rýchlo by totiž nastal stav, kde by sa väčšina zdrojov úplne zablokovala čakaním na ukončenie iných transakcií, čím by sa aplikácia stala nepoužiteľnou. Riešením tohto problému je definovanie kompenzačnej funkcie ku každej operácii. Ide o inverznú funkciu, avšak nie vždy ide o triviálny problém. Napríklad zaslaný email o potvrdení objednávky nie je možné zrušiť – je nutné zákazníka vhodným spôsobom upozorniť, že jeho objednávka síce bola odoslaná, ale nastali komplikácie, napr. havária kuriérskej služby, ktoré tento stav zmenili.

Vo všeobecnosti je takéto dlhé transakcie veľmi zložité implementovať, a preto vznikajú nástroje, ktoré sa tento proces snažia zjednodušiť a urýchliť. Orchestrátory biznis procesov sú väčšinou realizované nad sprostredkovateľmi správ a umožňujú manažment stavov biznis procesov a vizuálne modelovanie biznis procesov. Podporujú taktiež prepájanie rôznych systémov pomocou webových protokolov, súborových systémov, frontov, atď.

2.2.7 Problémy integračných architektúr

Veľkým problémom v softvérovej architektúre je integrácia veľkých aplikácii v heterogénnych prostrediach. Asi najdôležitejším problémom je zohľadnenie modifikovateľnosti vo veľkých architektúrach.

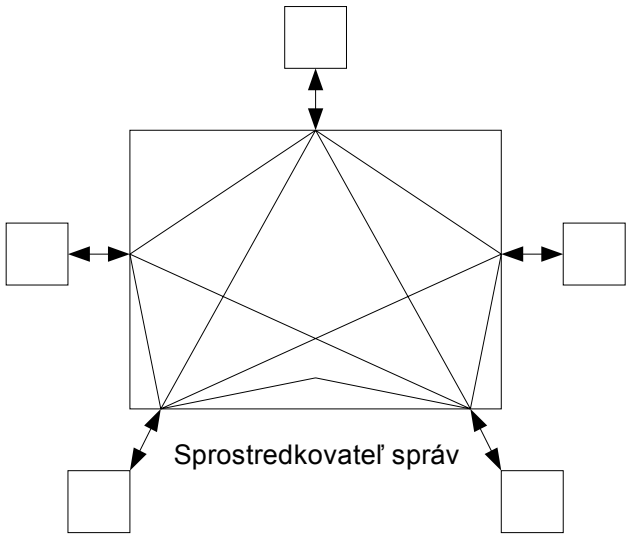
Obrázok 2-10 zobrazuje biznis proces, ktorý na svoju funkciu vyžaduje štyri rozhrania s inými systémami (vľavo). Problém nastáva, ak niektorý zo systémov zmení formát komunikácie s okolitými systémami. Pri zmene výmenného formátu je totiž nutné zmeniť rozhrania a teda transformácie tohto formátu. Bežne je však používa oveľa viac biznis procesov (vpravo). Pri zmene výstupného formátu je nutné zmeniť oveľa viac transformácií a takáto zmena značne komplikuje ďalší vývoj systému. Takúto architektúru nazývame „špagetová architektúra“.



Obrázok 2-10. Problém integrácie aplikácií.

Riešenie, ktoré sa pokúša tento problém riešiť je zavedenie sprostredkovateľa správ medzi rozhrania biznis procesov (Obrázok 2-11). Problémom zostáva skutočnosť, že „špagetová architektúra“ sa len presunula do sprostredkovateľa správ, čím sa celkový problém nevyriešil len presunul na jedno miesto. Sprostredkovateľ správ môže neskôr stať úzkym hrdlom celého systému.

Jediným možným riešením tohto problému je určenie všeobecného výmenného formátu nazývaného *Enterprise Data Model*, ktorý bude použitý v celom systéme. Zavedenie všeobecného formátu v praxi spravidla sprevádzajú vážne prekážky. Všeobecný formát musí zachytávať všetky aspekty údajov, ktoré sa budú prenášať a preto je jeho návrh a proces schvaľovania zdĺhavý, čím brzdí vývoj systému. Vo väčšine prípadov sa systémy integrujú s inými už existujúcimi systémami a dohoda o všeobecnom výmennom formát môže byť prakticky nemožná.



Obrázok 2-11. Integračná architektúra so sprostredkovateľom správ.

2.2.8 Zhrnutie

Technológie pre spojovací softvér vo všeobecnosti slúžia na urýchlenie a zjednodušenie návrhu a implementácie aplikácií. Tieto technológie vznikli z dvoch hlavných dôvodov:

- Umožňujú jednoduchší vývoj komplexných a distribuovaných aplikácií.
- Podporou vzorov a overených praktík v nástrojoch pomáhajú ich zavádzaniu a rozšíreniu do praxe.

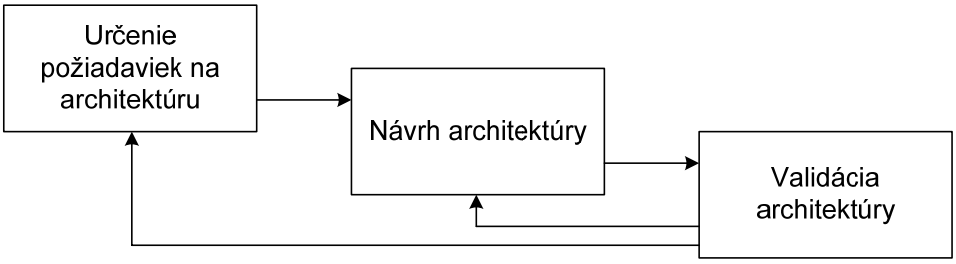
Softvéroví architekti majú neľahkú úlohu vo výbere a kombinácii rôznych architektúr a technológii pre spojovací softvér tak, aby vhodne splnili požiadavky aplikácií a zohľadňovali pri tom softvérové, hardvérové, finančné či ľudské ohraničenia.

2.3 Proces tvorby architektúry softvéru

Úloha softvérového architekta pri tvorbe softvérových systémov zahŕňa množstvo činností, ktoré nesúvisia len so samotným návrhom:

- *Spolupráca s tímom zabezpečujúcim zber požiadaviek na systém:* Tím zabezpečujúci zber požiadaviek sa zameriava na definovanie funkcionálnych požiadaviek. Architekt pri zbere požiadaviek musí pochopiť celkové požiadavky kladené na vyvíjaný systém, aby pochopil relevantné atribúty kvality, ktoré musí systém spĺňať.
- *Spolupráca s rôznymi používateľmi aplikácie:* Architekt hrá dôležitú úlohu pri pochopení požiadaviek používateľov a ich zapracovaní do výsledného návrhu.
- *Vedenie tímu technického návrhu:* Definovanie architektúry aplikácie je návrhová činnosť, pri ktorej architekt vedie tím systémových a technických návrhárov.
- *Spolupráca s manažmentom projektu:* Architekt úzko spolupracuje s manažmentom projektu a pomáha pri plánovaní, odhadovaní a prideľovaní úloh.

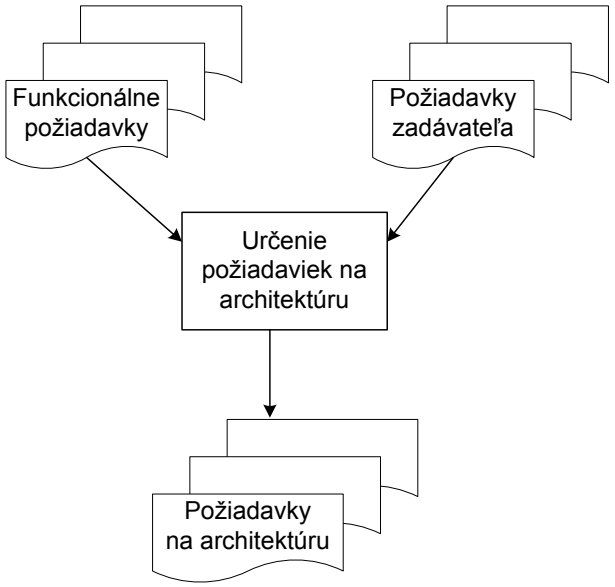
Pri návrhu architektúry sa používa overený inžiniersky proces tvorby architektúry (Obrázok 2-12). Pred samotným návrhom je nevyhnutné podrobne preskúmať všetky požiadavky a nároky, ktoré sú kladené na výslednú aplikáciu spolu s požiadavkami na nefunkcionálne aspekty aplikácie. Keďže samotný proces tvorby architektúry je zo svojej podstaty iteratívny – po návrhu architektúry môže validácia identifikovať potrebu modifikácií alebo dodefinovania požiadaviek, je nutné sa k fázam určenia požiadaviek a návrhu vracať až kým nie sú primerane uspokojené všetky požiadavky.



Obrázok 2-12. Trojkrokový iteratívny proces tvorby architektúry.

2.3.1 Určenie požiadaviek na architektúru

Obrázok 2-13 znázorňuje vstupy a výstupy procesu určenia požiadaviek na architektúru. Vstupmi sú dokumenty opisujúce funkcionálne požiadavky a požiadavky zadávateľa odzrkadľujúce nároky budúcich používateľov systému, výstupom sú požiadavky kladené na architektúru výsledného softvérového systému.



Obrázok 2-13. Vstupy a výstupy pre určenie požiadaviek na architektúru.

Typickým príkladom požiadavky na architektúru je:

> *„Komunikácia medzi komponentmi musí garantovať bezstratové zasielanie správ.“*

Príklad ohraničujúcej požiadavky je nasledovný:

> *„Systém musí na spracovanie požiadaviek používať webový server založený na IIS a ASP.“*

Nefunkcionálne požiadavky a ohraničenia obmedzujú architekta pri výbere riešení a tvorbe architektúry. V tabuľke 2-1 uvádzame príklady niektorých požiadaviek na architektúru systému, typické príklady obmedzení kladených na architektúru znázorňuje tabuľka 2-2.

Tabuľka 2-1. Príklady požiadaviek na architektúru.

Atribút kvality	**Požiadavka**
Výkonnosť	Aplikácia musí mať dobu odozvy do 0,25 sekúnd pre 90 % požiadaviek.
Bezpečnosť	Komunikácia musí byť autentifikovaná a šifrovaná použitím certifikátov.

Manažment zdrojov	Serverová časť aplikácie musí byť prevádzkovateľná na kancelárskom počítači s 512MB pamäťou.
Použiteľnosť	Používateľská časť aplikácie musí fungovať vo webových prehliadačoch.
Dostupnosť	Systém musí fungovať 24x7x365 s celkovou 99 % dostupnosťou.
Spoľahlivosť	Strata správ je neprijateľná a notifikácia musí prebehnúť za menej ako 30 sekúnd.
Škálovateľnosť	V špičke musí byť aplikácia schopná zvládnuť 500 súčasne prihlásených používateľov.
Modifikovateľnosť	Architektúra musí podporovať migráciu zo súčasného jazyka štvrtej generácie na .NET technológiu.

Tabuľka 2-2. Príklady obmedzení na architektúru.

Obmedzenie	**Požiadavka na architektúru**
Obchod	Technológia musí byť zrealizovaná ako zásuvný modul (*plug-in*) pre MS BizTalk, keďže ju chceme predávať Microsoftu.
Vývoj	Systém musí byť napísaný v programovacom jazyku Java, aby sa dali využiť existujúci pracovníci.
Plánovanie	Prvá verzia produktu musí byť hotová do šiestich mesiacov.
Obchod	Chceme úzko spolupracovať a byť financovaní spoločnosťou *MegaHugeTech Corp*, takže treba použiť ich technológie.

Definovanie priorít požiadaviek na architektúru

Keďže požiadavky na architektúru sú spravidla protichodné, uspokojenie niektorých z nich je nevyhnutné zatiaľ čo iné majú skôr charakter odporúčania. Z tohto dôvodu rozdeľujeme požiadavky podľa ich priority do troch úrovní:

- *Vysoká priorita* – aplikácia musí bezpodmienečne splniť túto požiadavku. Tieto požiadavky odzrkadľujú celkové nároky na aplikáciu.
- *Stredná priorita* – aplikácia musí splniť túto požiadavku v niektorej fáze vývoja, avšak nie nevyhnutne v prvej verzii.
- *Nízka priorita* – požiadavky, ktoré majú charakter prianí. Uspokojenie týchto požiadaviek je žiaduce, avšak nie nevyhnutné.

Často po identifikovaní požiadaviek zistíme, že niektoré z nich sa navzájom úplne alebo čiastočne vylučujú. Problém je zložitejší v prípade, ak majú tieto požiadavky pridelenú rovnakú prioritu. Príklady požiadaviek, ktoré sa vylučujú sú napríklad:

- Znovupoužiteľnosť komponentov verzus rýchly vývoj aplikácie. Dôvodom rozporu je fakt, že vývoj generických znovupoužiteľných modulov trvá omnoho dlhšie ako riešenia pre konkrétny, špecifickejší problém.
- Výdavky na použitie *COTS* produktov verzus znížené vývojové náklady. Použitie *COTS* produktov znamená tvorbu menšieho množstva kódu, ale s vyššími nákladmi.

Neľahké riešenie konfliktných požiadaviek je úlohou softvérového architekta, ktorý ich musí analyzovať a diskutovať o nich so zadávateľom a budúcimi používateľmi. Výsledkom jeho snaženia je zvyčajne kompromis primerane uspokojujúci obe požiadavky, avšak architekt sa nakoniec môže rozhodnúť aj pre riešenie, v ktorom nebude vôbec uspokojená jedna z požiadaviek.

2.3.2 Návrh architektúry

Napriek tomu, že všetky úlohy softvérového architekta sú veľmi dôležité, to na čom naozaj záleží je kvalita výsledného návrhu systému. Samotný návrh je najnamáhavejšou časťou práce architekta. Dobrý softvérový architekt potrebuje mnohoročné skúsenosti v oblasti softvérového inžinierstva a návrhu architektúr softvéru vo viacerých projektoch.

Na obrázku 2-14 sú znázornené vstupy a výstupy procesu návrhu architektúry. Požiadavky na architektúru predstavujú vstupy a výstup tvoria rôzne pohľady na architektúru v podobe viacerých modelov a samotná dokumentácia architektúry.

Prvým krokom procesu je výber celkovej stratégie, na ktorej bude systém postavený. Tu je vhodné využiť známe a overené architektonické vzory. Druhým krokom je špecifikovanie samotných komponentov, z ktorých sa vybuduje architektúra podľa zvolených vzorov a pridelenie ich zodpovedností.

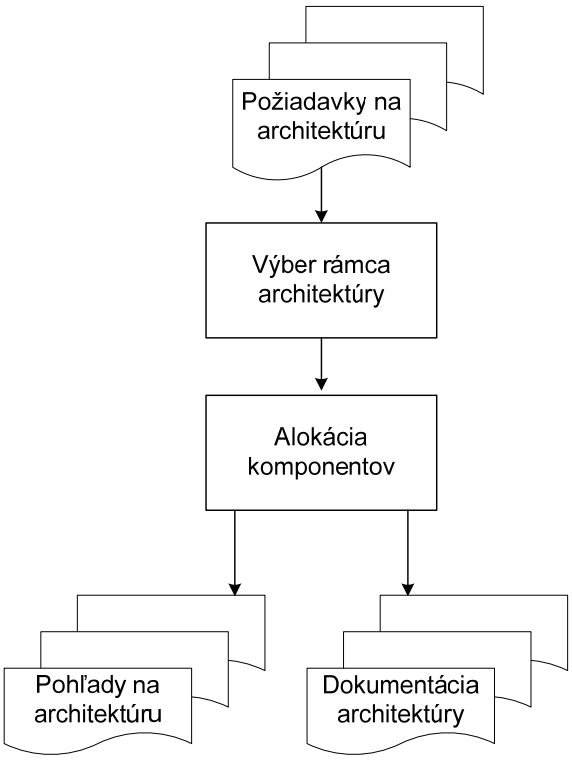
Výber rámca architektúry

Väčšinu vyvíjaných aplikácií je možné postaviť na známych, overených architektúrach. Dôvodom prečo je vhodné použiť známe, overené riešenie je jednoducho to, že fungujú. Použitie takéhoto riešenia minimalizuje nebezpečenstvo, že aplikácia nebude pracovať správne z dôvodu zle zvolenej architektúry.

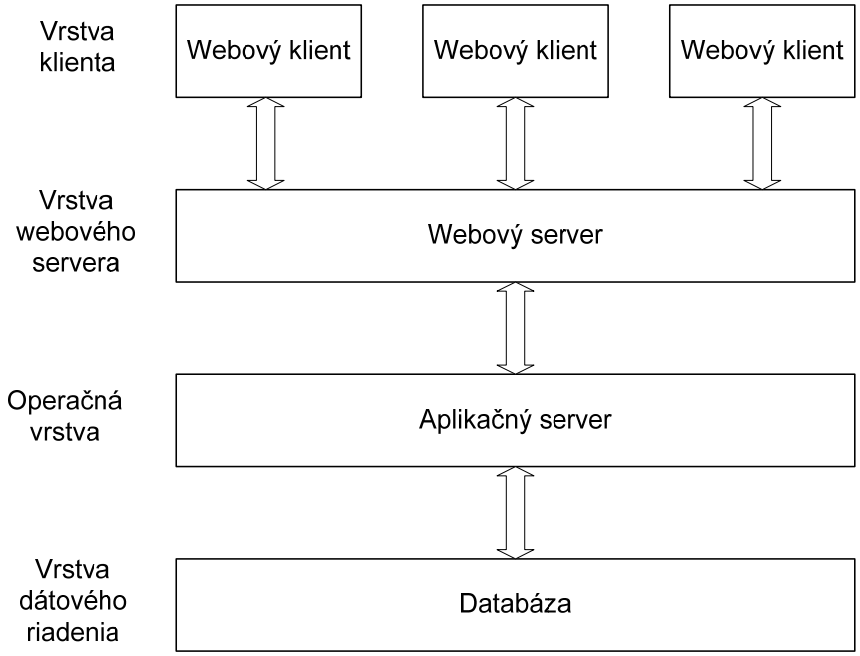
Prvý krok návrhu zahŕňa voľbu rámca architektúry, ktorý môže uspokojiť požiadavky. Pre mnoho aplikácií môže vyhovovať známa vrstvová architektúra. Pre niektoré komplexné systémy môže byť potrebné zapojiť jeden alebo viacero iných známych vzorov architektúr. Neexistuje univerzálny recept na návrh architektúry systému. Vždy je nevyhnutné brať do úvahy špecifické požiadavky každej aplikácie a k nim prispôsobiť samotný návrh.

N-vrstvová klient-server architektúra

Na obrázku 2-15 je znázornená štruktúra známej *N-vrstvovej klient-server architektúry*.



Obrázok 2-14. Vstupy o výstupy návrhu architektúry.



Obrázok 2-15. N-vrstvová klient-server architektúra.

Základné vlastnosti tohto vzoru sú:

- *Oddelenie logicky súvisiacich častí:* Prezentačná, operačná a dátová časť systému sú od seba oddelené do vrstiev.
- *Synchrónna komunikácia:* Komunikácia medzi jednotlivými vrstvami je synchrónna v tvare požiadavka-odpoveď. Jednotlivé požiadavky sú propagované cez viaceré vrstvy, ktoré majú definované rozhrania, ktorými poskytujú svoju funkcionalitu. Vybavenie požiadavky zväčša vyžaduje vytvorenie sub-požiadavky zaslanej ďalšej vrstve v smere postupu (zhora nadol alebo zdola nahor). Odpovede postupujú v opačnom smere ako požiadavky. Príkladom je zaslanie požiadavky webovým serverom aplikačnému serveru, ktorý na jej spracovanie vytvorí sub-požiadavku a zašle ju vrstve dátového riadenia. Odpoveď na sub-požiadavku potom umožní spracovanie pôvodnej požiadavky webového servera.
- *Flexibilné nasadenie:* Neexistujú obmedzenia na nasadenie vrstvovej architektúry. Jednotlivé vrstvy môžu byť umiestnené na rovnakom alebo na rôznom fyzickom zariadení. Vo webových aplikáciách je klientska vrstva zväčša realizovaná webovým prehliadačom komunikujúcim s ďalšími vrstvami aplikácie cez Internet.

Tabuľka 2-3 znázorňuje rôzne atribúty kvality v kontexte N-vrstvovej klient-server architektúry. Presné vlastnosti tejto architektúry vzhľadom na jednotlivé atribúty kvality závisia od konkrétne zvolenej technológie a implementácie. Medzi základné technológie v tejto oblasti patria .NET a J2EE. Pri voľbe technológie musia byť známe špecifiká každej alternatívy. V neskorších fázach vývoja aplikácie je zistenie, že zvolená technológia nepodporuje určitú funkcionalitu veľmi nepríjemné a môže spôsobiť značné komplikácie , ktorých riešenie môže byť veľmi nákladné.

Architektonický vzor N-vrstvová klient-server architektúra sa bežne používa a má veľkú podporu zo strany existujúcich aplikačných serverov.

Tabuľka 2-3. Atribúty kvality pre N-vrstvovú klient server architektúru.

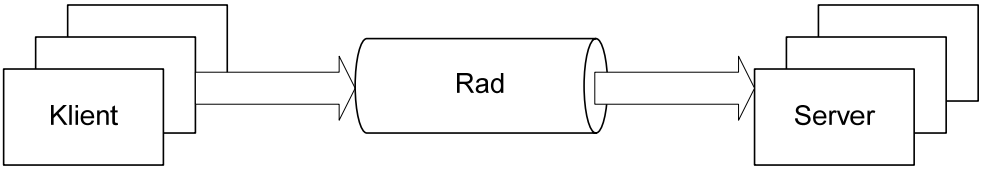
Atribút kvality	**Aspekty a možnosti realizácie**
Dostupnosť	Servery na každej vrstve môžu byť replikované tak, aby v prípade poruchy jedného boli ostatné schopné pracovať. Počas poruchy môže aplikácia poskytovať nižšiu kvalitu služby.
Spracovanie porúch	Ak klient komunikuje so serverom, ktorý má poruchu, mnoho webových a aplikačných serverov realizuje transparentné presmerovanie požiadavky klienta na záložný server bez vedomia klienta.
Modifikovateľnosť	Oddelenie jednotlivých súčastí systému do vrstiev zvyšuje modifikovateľnosť, keďže prezentačná, operačná a vrstva dátového riadenia sú zapuzdrené do modulov.
Výkonnosť	Vrstvová architektúra preukázala vysokú výkonnosť. Kľúčovými sú množstvo paralelných spracovaní podporovaných servermi, rýchlosť komunikácie medzi vrstvami a množstvo prenášaných dát. Tak ako v každom distribuovanom systéme, aj v tomto prípade je vhodné minimalizovať volania medzi vrstvami pri spracovaní požiadavky.

Škálovateľnosť	Servery na jednotlivých vrstvách môžu byť replikované a inštancie bežiace na vrstvách môžu byť znásobené a vykonávať sa na rovnakom alebo inom serveri. Týmto spôsobom je aplikácia dobre škálovateľná. V praxi je často úzkym hrdlom systému vrstva dátového riadenia.

Architektúra zasielanie správ

Na obrázku 2-16 je znázornená štruktúra architektonického vzoru *Zasielanie správ*. Základná charakteristika tohto vzoru je:

- *Asynchrónna komunikácia:* Klienty zasielajú požiadavku do radu, v ktorom sú uchovávané do doby kým ich aplikácia neodoberie k spracovaniu. Klienty nečakajú na odpoveď, ale pokračujú v ďalšej činnosti.
- *Konfigurovateľná kvalita služby:* Rad, v ktorom sú uchovávané správy môže byť nakonfigurovaný pre rýchle, ale menej spoľahlivé doručovanie alebo pre pomalšie, ale o to spoľahlivejšie doručovanie. Operácie v rade môžu byť koordinované databázovými transakciami.
- *Slabá previazanosť:* Neexistuje priame prepojenie medzi klientmi a servermi. Pre klienta nie je podstatné, ktorý server spracuje jeho správu rovnako ako pre server je nepodstatné, ktorý klient správu do radu vložil.



Obrázok 2-16. Architektúra zasielanie správ.

Tabuľka 2-4 opisuje jednotlivé atribúty kvality pre vzor zasielanie správ. Konkrétne vlastnosti vzoru silne závisia od použitej technológie a implementácie.

Tabuľka 2-4. Atribúty kvality architektúry zasielanie správ.

Atribút kvality	**Aspekty a možnosti realizácie**
Dostupnosť	Fyzické rady s rovnakým logickým menom môžu byť replikované na rôznych inštanciách serverov. V prípade poruchy jedného môže klient zasielať správy replike radu.
Spracovanie porúch	V prípade poruchy v komunikácii s radom klient môže nájsť vhodnú repliku radu a zasielať mu správy.
Modifikovateľnosť	Zasielanie správ je zo svojej podstaty slabo previazaná architektúra zlepšujúca modifikovateľnosť, keďže klienty a servery nie sú priamo previazané cez rozhranie. Zmeny v implementácii servera sú potrebné v prípade zmeny formátu správy zasielanej klientom. Samoopisnosť správ môže prispieť k zvýšeniu nezávislosti v prípade zmien.

Výkonnosť	Zaraďovanie správ do radov umožňuje doručiť tisícky správ za sekundu. V prípade menej spoľahlivého doručovania môže byť zasielanie rýchlejšie v závislosti od kvality použitej technológie na zasielanie správ.
Škálovateľnosť	Rady môžu byť usporiadané na komunikačných koncoch alebo replikované na klastroch serverov na zasielanie správ usporiadaných do jedného alebo viacerých zariadení. Toto robí architektúru zasielania správ vysoko škálovateľnou.

Zasielanie správ je vhodná voľba v prípadoch kedy klient nepotrebuje okamžitú odpoveď po zaslaní správy. Príkladom je zaslanie emailu, ktorý sa uloží na mailovom serveri pre spracovanie. Server po čase správu odoberie z radu a spracuje ju. V tejto situácii klient nepotrebuje vedieť, kedy server jeho správu spracuje.

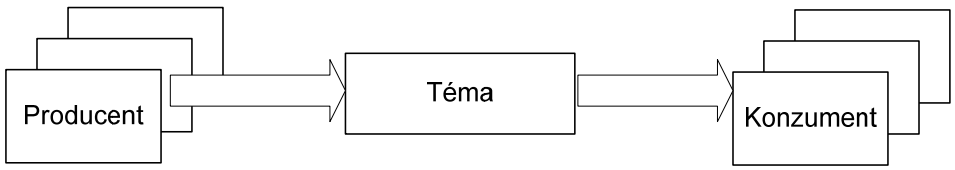
Aplikácie, ktoré rozdeľujú spracovanie požiadavky do niekoľkých diskrétnych krokov prepojených radmi sú priamym rozšírením vzoru *Zasielanie správ*. Toto rozšírenie je totožné so vzorom *Dátovody a filtre*.

Zasielanie správ je riešenie, ktoré je odolné voči situáciám, kedy je komunikácia prostredníctvom siete dostupná len v obmedzenom čase, napríklad z dôvodu poruchy. V takom prípade sú správy uchované v rade pokiaľ ich server neodoberie.

Zasielanie správ je vo svojej podstate asynchrónny systém. Tok dát pritom postupuje jedným smerom. Modifikáciou tohto vzoru, v ktorej sa použijú rady, ktoré sú v páre a realizujú jednak prijatie požiadaviek, ale aj zaslanie odpovede je možné zabezpečiť synchrónnu komunikáciu. V takomto prípade dáta prúdia oboma smermi.

Architektúra producent-konzument

Základná štruktúra vzoru *Producent-konzument* je znázornená na obrázku 2-17.



Obrázok 2-17. Architektúra Producent-konzument.

Jeho kľúčovými vlastnosťami sú:

- *Zasielanie správ formou mnohé-s-mnohými:* Správy publikované do témy sú zasielané všetkým konzumentom, ktorí sú na danú tému registrovaný. Pritom, do rovnakej témy môžu publikovať viaceré producenty.
- *Konfigurovateľná kvalita služby:* Zasielanie správ môže byť spoľahlivé alebo nespoľahlivé. Navyše mechanizmus komunikácie môže byť realizovaný formou *point-to-point* alebo *broadcast/multicast*.
- *Slabá previazanosť:* Rovnako ako v prípade vzoru *Zasielanie správ*, aj v tomto prípade neexistuje úzke previazanie medzi producentmi a konzumentmi. Produ-

cent nevie kto prijme jeho správu a rovnako konzument nevie kto danú správu zverejnil.

Architektúry založené na vzore producent-konzument sú veľmi flexibilné a vhodné pre aplikácie vyžadujúce asynchrónnu komunikáciu *jeden-s-mnohými*, *mnohé-s-jedným* alebo *mnohé-s-mnohými*. Rovnako ako v prípade zasielania správ aj tu je možné vytvoriť páry tém zabezpečujúce prijatie požiadavky aj odpovede na ňu. Vlastnosti vzoru producent-konzument z pohľadu atribútov kvality opisuje tabuľka 2-5.

Tabuľka 2-5. Atribúty kvality architektúry Producent-konzument.

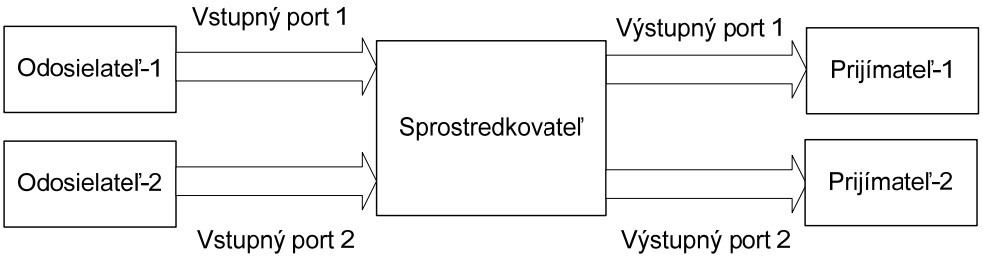
Atribút kvality	**Aspekty a možnosti realizácie**
Dostupnosť	Témy s rovnakým logickým menom môžu byť replikované na rôznych inštanciách servera organizovaných do klastrov. V prípade poruchy, producent zasiela správy do replík radov.
Spracovanie porúch	Ak producent komunikuje s témou, umiestnenou na serveri s poruchou, môže byť presmerovaný na jeho repliku.
Modifikovateľnosť	Architektúra producent-konzument je z podstaty slabo previazaná, čo prispieva k modifikovateľnosti. Nový producent alebo konzument môže byť pridaný do systému bez zmeny architektúry alebo konfigurácie. V prípade zmien formátu zverejňovanej správy môže byť potrebná zmena implementácie konzumenta.
Výkonnosť	Systém založený na architektúre producent-konzument je schopný doručiť tisícky správ za sekundu. Pri menej spoľahlivom zasielaní správ môže rýchlosť ďalej narásť. V prípade ak je podporované zasielanie správ formou *multicast/broadcast*, ich doručenie môže byť realizované v jednom čase.
Škálovateľnosť	Témy môžu byť replikované v klastroch serverov organizovaných do jedného alebo viacerých zariadení. Klastre serverov môžu byť škálované tak, aby podporovali veľmi vysokú priepustnosť správ. Riešenia realizované formou *multicast/broadcast* sú lepšie škálovateľné ako riešenia *point-to-point*.

Architektúra sprostredkovateľ

Hlavné komponenty vzoru *Sprostredkovateľ* sú znázornené na obrázku 2-18. Jeho kľúčovými vlastnosťami sú:

- *Hub-and-spoke architektúra:* Sprostredkovateľ predstavuje *hub*, na ktorý sa pripájajú odosielatelia a prijímatelia. Spojenia na sprostredkovateľa sa realizujú cez porty asociované s určitým formátom správy.

- *Realizácia smerovania správ* – Sprostredkovateľ zapuzdruje logiku smerovania správ prijatých na vstupnom porte na výstupný port. Cesta doručovania môže byť definovaná pevne alebo môže závisieť od správy prijatej na vstupnom porte.
- *Realizácia transformácie správ* – Sprostredkovateľ transformuje správu prijatú na vstupnom porte do tvaru požadovaného na výstupnom porte.



Obrázok 2-18. Architektúra sprostredkovateľ.

Tabuľka 2-6 obsahuje opis atribútov kvality vzoru sprostredkovateľ.

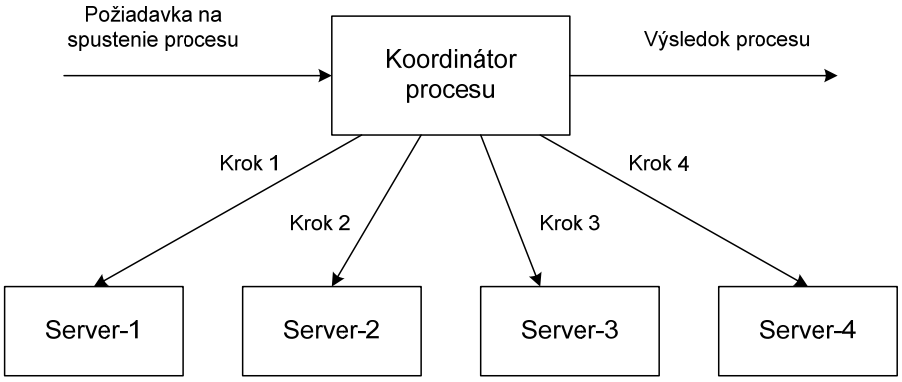
Tabuľka 2-6. Atribúty kvality architektúry sprostredkovateľ.

Atribút kvality	**Aspekty a možnosti realizácie**
Dostupnosť	Na vybudovanie spoľahlivo dostupnej architektúry musia byť sprostredkovatelia replikovaní. Toto je typicky zabezpečené použitím podobného mechanizmu ako pri architektúre zasielanie správ alebo producent-konzument s využitím klastrov.
Spracovanie porúch	Sprostredkovatelia majú typované vstupné porty. Správy, ktoré prijímajú sú validované a tie, ktoré majú zlý formát sú zahodené. Replikovaním sprostredkovateľov je možné zabezpečiť presmerovanie požiadavky na záložnú repliku v prípade poruchy.
Modifikovateľnosť	Sprostredkovatelia oddeľujú transformáciu a smerovanie správ od odosielateľa a prijímateľa. Toto rozširuje možnosti modifikovateľnosti, keďže zmeny v transformáciách a logike smerovania môžu byť realizované bez zmien v odosielateľoch a prijímateľoch.
Výkonnosť	Sprostredkovateľ sa stáva úzkym hrdlom systému ak musí poskytovať spracovanie veľkého množstva správ a zložitú transformáciu. Priepustnosť je v takom prípade zväčša menšia ako v prípade jednoduchého zasielania správ so spoľahlivým doručením.
Škálovateľnosť	Organizovanie sprostredkovateľov do klastrov umožňuje konštruovať dobre škálovateľné systémy.

Vzor sprostredkovateľ je vhodný pre aplikácie, ktoré vyžadujú spracovanie správ počas ich doručovania. Sprostredkovateľ umožňuje odosielateľovi a prijímateľovi zachovať si vlastný formát správ a preberá zodpovednosť za transformáciu medzi požadovanými tvarmi. Zároveň sa sústredením transformácie do sprostredkovateľa správ zvyšuje jej pochopiteľnosť a modifikovateľnosť.

Architektúra koordinátor procesu

Základné prvky vzoru *Koordinátor procesu* sú znázornené na obrázku 2-19.



Obrázok 2-19. Architektúra koordinátor procesu.

Jeho hlavné vlastnosti sú:

- *Zapuzdrenie procesu:* Koordinátor procesu zapuzdruje kroky realizujúce proces, ktorý môže byť ľubovoľne zložitý. Keďže koordinátor procesu je jediným miestom definície procesu, stáva sa tento ľahko pochopiteľným a modifikovateľným. Proces sa realizuje po prijatí požiadavky postupným volaním serverov podľa definície, po skončení posledného kroku koordinátor vráti výsledok procesu.
- *Voľné previazanie:* Jednotlivé servery nevedia, aká je ich rola v celkovom procese a kde v procese sú požadované ich služby. Vedie len čo môžu poskytnúť a koordinátor procesu ich volá podľa potreby procesu.
- *Flexibilná komunikácia* – Komunikácia medzi koordinátorom a jednotlivými servermi môže byť synchrónna aj asynchrónna. Pri synchrónnej komunikácii koordinátor procesu čaká na odpoveď servera. Pri asynchrónnej komunikácii koordinátor procesu nečaká na odpoveď a pokračuje ďalším krokom procesu.

Tabuľka 2-7 opisuje atribúty kvality vzoru koordinátor procesu.

Tabuľka 2-7. Atribúty kvality architektúry koordinátor procesu.

Atribút kvality	**Aspekty a možnosti realizácie**
Dostupnosť	Koordinátor je potenciálnym miestom vzniku poruchy. Z tohto dôvodu je potrebné ho replikovať k vytvoreniu vysoko dostupného riešenia.

Spracovanie porúch	Spracovanie poruchových stavov je komplexný problém, keďže porucha sa môže vyskytnúť v ktoromkoľvek stave procesu. Porucha v neskorších krokoch procesu môže spôsobiť potrebu realizovať kroky na odstránenie výsledkov predchádzajúcich krokov. Spracovanie porúch vyžaduje dômyselný návrh spôsobu zachovania konzistentnosti dát na serveroch.
Modifikovateľnosť	Modifikovateľnosť procesu je podporovaná keďže definícia procesu je zapuzdrená do koordinátora procesu. Implementácia serverov môže byť zmenená pokiaľ nedôjde k zmene definícií poskytovaných služieb.
Výkonnosť	K dosiahnutiu vysokej výkonnosti je potrebné aby koordinátor bol schopný spracovať veľké množstvo požiadaviek súčasne a riadiť stav každej z nich počas realizácie procesu. Tá je obmedzovaná najpomalším krokom – najpomalším serverom.
Škálovateľnosť	Koordinátori môžu byť replikovaní k dosiahnutiu dobrej škálovateľnosti aplikácie.

Koordinátor procesu je vhodný, ak je potrebné realizovať zložité procesy, ktoré vyžadujú služby distribuované na viacerých serveroch. Sústredením definície procesu na jedno miesto sa umožní ľahké riadenie, zmena a monitorovanie procesu.

Integračné architektúry sprostredkovateľ správ a orchestrátor biznis procesov sú obe založené na vzore koordinátor procesu, pričom sprostredkovateľ správ je určený pre požiadavky s krátkou životnosťou a orchestrátor biznis procesov pre procesy, ktoré môžu trvať od niekoľkých minút po niekoľko týždňov.

Alokácia komponentov

Po výbere rámca založeného na architektonických vzoroch treba definovať hlavné komponenty architektúry:

- identifikovať hlavné komponenty a spôsob akým sú zasadené do rámca,
- identifikovať rozhrania a služby, ktoré budú jednotlivé komponenty ponúkať,
- identifikovať zodpovednosti komponentov,
- identifikovať závislosti medzi komponentmi,
- identifikovať tie časti architektúry, ktoré môžu byť distribuované na rôznych serveroch po sieti.

Komponenty, z ktorých je postavená architektúra sú hlavnými abstrakciami v aplikácii. Nie je prekvapením, že existuje podobnosť medzi návrhom komponentov architektúry a objektovo-orientovaným návrhom. Diagramy tried a balíkov sa často používajú na opis komponentov architektúry a vzťahov medzi nimi.

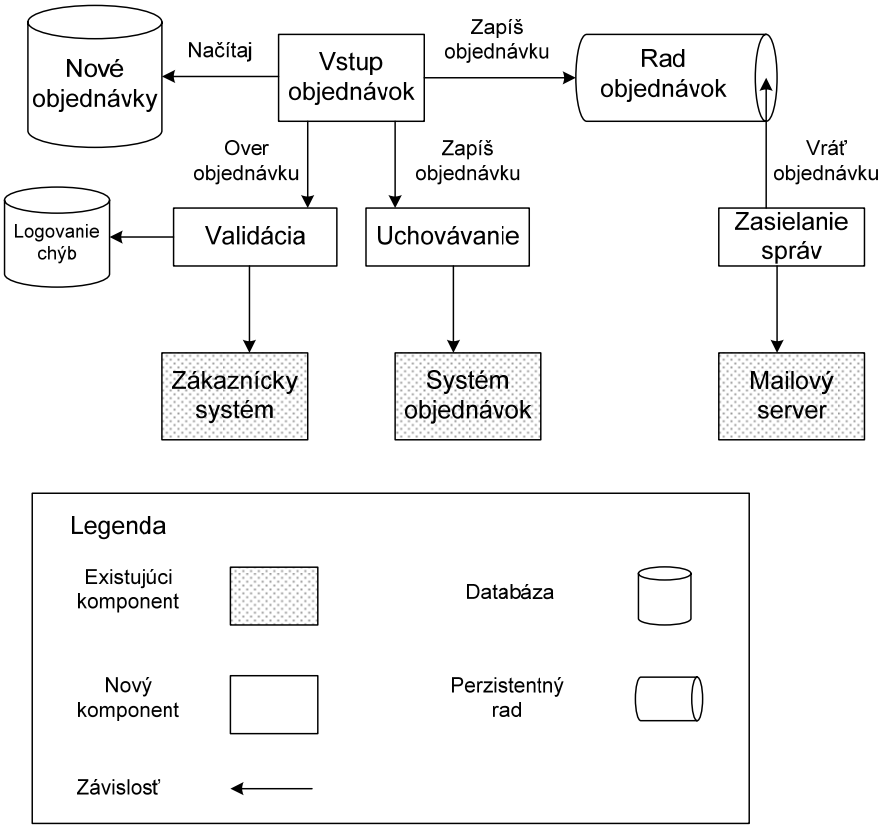
Pri návrhu komponentov je vhodné:

- Minimalizovať závislosti medzi komponentmi. Vždy sa treba snažiť vytvárať komponenty medzi ktorými je slabá previazanosť, aby zmena jedného komponentu nevyžadovala zmenu celého radu ďalších komponentov v ostatných

častiach architektúry. Dôležité je uvedomiť si, že po každej zmene komponentu je potrebné nanovo spustiť proces testovania systému.

- Je vhodné navrhovať komponenty, ktoré sú vysoko súdržné. Súdržnosť hovorí nakoľko spolu jednotlivé časti komponentu vnútorne súvisia. Vysoko súdržné komponenty zapuzdrujú dobre definovanú funkcionalitu. Ich výhodou je, že sústreďujú potenciálne zmeny do jedného miesta, čím minimalizujú náklady na udržiavanie a potrebu testovania.
- Štruktúru komponentov je potrebné hierarchicky dekomponovať. Jednotlivé komponenty pozostávajú zo subkomponentov. Vonkajší komponent pritom poskytuje dostupné rozhrania subkomponentov.
- Je potrebné minimalizovať volania medzi komponentmi najmä v prípade distribuovaných komponentov a snažiť sa sústreďovať postupnosť volaní medzi komponentmi do jedného volania.

Na obrázku 2-20 je znázornený príklad pohľadu na štruktúru aplikácie na spracovanie objednávok. Systém prijíma objednávky a ukladá ich do databázy. Každá objednávka musí byť validovaná voči existujúcemu zákazníckemu systému na overenie zákazníckych informácií a platnosti platby. Po úspešnej validácii je objednávka uložená do databázy (rad objednávok) a zákazníkovi je zaslaný email informujúci o začatí spracovania jeho objednávky.



Obrázok 2-20. Príklad architektúry – systém na spracovanie objednávok.

Základným rámcom aplikácie je zasielanie správ. Pri riešení boli využité štyri nové komponenty:

- *Vstup objednávok:* Má zodpovednosť za prístup do databázy nových objednávok, zapuzdruje logiku spracovania a zapisuje do radu objednávok.
- *Validácia:* Zabezpečuje validáciu v spolupráci s existujúcim zákazníckym systémom. V prípade chybnej objednávky zaznamenáva informácie pomocou logovania chýb.
- *Uchovávanie:* Spolupracuje so systémom objednávok a ukladá dáta objednávok.
- *Zasielanie správ:* Vyberá správy z radu, vytvára email a zasiela ho zákazníkovi prostredníctvom mailového servera. Zapuzdruje znalosti o formáte emailu a prístupu k mailovému serveru.

Každý komponent má jasné závislosti a malú množinu zodpovedností, vytvárajúc tak slabo previazané, vysoko súdržné riešenie.

2.3.3 Validácia

Úlohou fázy validácie je potvrdiť, že navrhnuté riešenie je schopné uspokojiť požadované ciele. Dôkladné preskúmanie a overenie návrhu je časovo a finančne veľmi náročné. Navyše v praxi máme často na validáciu riešenia len obmedzené zdroje.

Pri overovaní návrhu architektúry je potrebné si uvedomiť, že návrh je len návrh. Nie je možné ho spustiť a testovať ako uspokojuje definované požiadavky. Často obsahuje nové komponenty, ktoré ešte len treba vytvoriť alebo „*black-box*“ komponenty. Tieto súčasti musia byť integrované k vytvoreniu výsledného systému.

Existujú dva hlavné prístupy k validácii, ktorej cieľom je odhalenie chýb v návrhu ešte pred implementáciou:

- manuálne testovanie architektúry použitím scenárov,
- prototypovanie kritických častí systému.

Použitie scenárov

Použitie scenárov je technika vyvinutá na SEI (*Software Engineering Institute*). Jednotlivé scenáre súvisia s rôznymi aspektmi navrhovaného systému, napr. rôznymi atribútmi kvality.

Metóda SEI ATAM určená na vyhodnotenie architektúry systému podrobne opisuje scenáre a spôsob ich vytvárania. Pri tejto metóde je potrebné definovať podnety, ktoré majú vplyv na architektúru. Scenár potom opisuje ako má systém reagovať na tieto podnety. Ak systém reaguje správne na daný podnet, príslušný scenár sa považuje za uspokojený. V opačnom prípade je pravdepodobné, že navrhnutá architektúra nepokrýva určité požiadavky.

Jednotlivé scenáre môžu byť naviazané na rôzne atribúty kvality. Možné príklady sú znázornené v tabuľke 2-8. Napríklad scenár súvisiaci s dostupnosťou systému hovorí, že správy sa môžu stratiť len v prípade, že sever zlyhal pred doručením správy a obnovením spojenia. Implikáciou je, že správy nie sú perzistentne uchovávané na disk, zrejme z dôvodu výkonnosti. Strata správ môže byť tolerovaná v niektorých kontextoch. Ak nie je, scenár poukazuje na problém, ktorý môže vyžadovať zavedenie perzistentného uchovávania správ k zamedzeniu straty správ.

Tabuľka 2-8. Príklady scenárov.

Atribút kvality	**Podnet**	**Reakcia**
Dostupnosť	Zlyhalo sieťové spojenie ku konzumentovi.	Správy sú uchované na MOM serveri pokiaľ nie je obnovené spojenie. Správy sa stratia iba v prípade ak server zlyhá pred obnovením spojenia.
Modifikovateľnosť	Musia byť vytvorené nové komponenty na analýzu dát v aplikácii.	Aplikácia musí byť prestavaná s novou knižnicou a na každom počítači musia byť aktualizované konfiguračné súbory k tomu aby boli viditeľné nové komponenty v GUI toolboxe.
Bezpečnosť	Nie je zaznamenaná aktivita používateľa po dobu 10 minút.	Aplikácia vyhodnotí stav ako potenciálne nebezpečný. Používateľ sa musí znovu prihlásiť ak chce pokračovať v práci.
Modifikovateľnosť	Dodávateľ transformačného stroja skrachuje.	Musí byť zakúpený nový transformačný stroj. Abstraktná vrstva služby obaľujúcej transformačný stroj musí byť reimplementovaná tak, aby podporovala nové riešenie. Klientske komponenty nie sú ovplyvnené.
Škálovateľnosť	Počet súčasne pracujúcich používateľov sa zdvojnásobuje za trojtýždňové obdobie.	Aplikačný server je rozšírený do dvoch strojov v klastri, aby bolo možné zvládnuť zaťaženie.

Prototypovanie

Scenáre sú veľmi užitočnou technikou na validáciu navrhovanej architektúry. Niektoré scenáre však nie je možné ľahko vyhodnotiť. Príkladom je scenár pre výkonnosť systému. Pre objednávkový systém je takýto:

> *V piatok poobede musia byť všetky prijaté objednávky spracované pred uzávierkou tak, aby boli doručené do pondelka. Päť minút pred uzávierkou príde 5000 objednávok.*

Odpoveď na otázku, či je systém schopný spracovať zvýšený počet objednávok je veľmi náročná keďže niektoré komponenty, ktoré majú zabezpečovať spracovanie ešte nie sú vytvorené.

Jedinou možnosťou ako zodpovedať túto otázku s dostatočnou dôveryhodnosťou je vytvorenie prototypu – minimalistickej, zjednodušenej verzie celej alebo len určitej časti aplikácie, určenej na overenie niektorých kritických aspektov vyvíjaného systému.

Cieľom je:

- Ukázať, či navrhnutá architektúra je vhodná na vytvorenie aplikácie, ktorá uspokojuje definované požiadavky (*proof of concept*).
- Zistiť, či sa zvolená technológia správa podľa očakávaní (*proof of technology*).

V oboch prípadoch prototypovanie pomáha pri overovaní tých aspektov, ktoré je veľmi náročné alebo nemožné vyhodnotiť iným spôsobom.

Pred vytváraním prototypu je potrebné definovať tie časti aplikácie, ktoré budú overované prototypom. V ideálnom prípade sa prototyp správa (v niektorých aspektoch) identicky ako výsledný systém. Aby bolo možné výsledky prototypovania brať do úvahy je potrebné vytvoriť prostredie, ktoré sa čo najviac podobá reálnemu.

Ak architekt vie, že mailový systém je schopný zvládnuť 5000 objednávok (rovnaký subsystém je nasadený aj v inej aplikácii) nie je potrebné zahrnúť túto časť do prototypu. Vlastnosti niektorých častí alebo prepojení komponentov architektovi nemusia byť známe. Do prototypu môžu byť zahrnuté rôzne generátory (simulátory), ktoré vytvárajú požiadavky, ktoré má systém spracovať.

Po vytvorení prototypu a jeho otestovaní je možné zodpovedať rôzne otázky súvisiace s tými aspektmi systému, ktoré iným spôsobom nie je možné zodpovedne overiť. Takými sú napríklad výkonnosť, škálovateľnosť či náročnosť integrácie zvolených komponentov.

Navzdory veľkej užitočnosti prototypu musí byť jeho vytváranie realizované opatrne, musia byť ustrážené náklady na jeho vytvorenie. V ideálnom prípade trvá prototypovanie jeden alebo dva dni, najviac jeden až dva týždne. Je potrebné si uvedomiť, že značná časť prototypu bude po testovaní „zahodená“. Nie je teda vhodné stráviť prototypovaním veľa času a investovať doň množstvo úsilia. Každá investícia musí byť dostatočne odôvodnená.

2.3.4 Zhrnutie

Návrh architektúry systému je kreatívna činnosť opísaná jednoduchým procesom. Tento zahŕňa definovanie požiadaviek, využitie existujúcich vzorov pri návrhu architektúry a jeho validáciu. Trojkrokový proces opísaný v tejto kapitole je zo svojej podstaty iteratívny. Prvotný návrh je validovaný voči požiadavkám a scenárom, výsledok validácie môže viesť k úpravám návrhu. Celý proces iteruje až kým nie sú primerane uspokojené všetky požiadavky a je možné vytvoriť výsledný systém.

Opísaný proces je dobre škálovateľný. Je možné ho prispôsobiť na použitie v malých aj veľkých projektoch, pričom je potrebné si zvoliť vhodnú cestu súvisiacu s množstvom dokumentácie a úrovňou formálnosti.

2.4 Dokumentovanie architektúry softvéru

Dokumentovanie architektúry softvérového systému patrí k činnostiam, ktorým sa ľudia často neradi venujú. To má za následok nedokonalé, neaktuálne, prípadne aj úplne chýbajúce dokumentácie. Existujú však aj projekty v ktorých sa vytvárajú obrovské množstvá dokumentov. Tento druhý extrém je tiež nevhodný, pretože sa v takýchto dokumentáciách problematicky vyhľadávajú informácie.

Skúsenosti poukazujú na to, že vytvorenie užitočnej dokumentácie nie je ľahká úloha. Medzi dôvody, kvôli ktorým je dobré vytvárať dokumentáciu patria:

- Aj iní môžu pochopiť a vyhodnotiť návrh. Medzi tých, ktorým môže byť dokumentácia najviac užitočná patria hlavne ďalší členovia vývojárskeho tímu.
- My sami vieme pochopiť návrh keď sa k nemu vraciame po dlhšom čase. Tiež počas vytvárania dokumentácie si lepšie uvedomíme rôzne súvislosti týkajúce sa architektúry.
- Vieme vykonávať analýzy návrhu. Môžeme napríklad generovať štandardné metriky týkajúce sa architektúry ako previazanosť alebo súdržnosť.

Dokumentovanie architektúry softvérového systému je problematické z nasledujúcich dôvodov:

- Neexistuje univerzálne akceptovaný štandard dokumentovania.
- Architektúra systému je väčšinou zložitá a jej dostatočné dokumentovanie je časovo náročná, netriviálna úloha.
- Existuje viacero pohľadov na architektúru. Dokumentovanie všetkých potenciálne potrebných pohľadov je časovo náročné a drahé.
- Architektúra systému sa vyvíja. Udržovanie aktuálnosti dokumentácie je náročné hlavne pri potrebe dodržania termínov a plánu v projekte.

2.4.1 Čo dokumentovať

Najväčší vplyv na odpoveď na otázku čo dokumentovať má zložitosť architektúry navrhovaného systému. Dvojvrstvová klient-server architektúra s komplexnou aplikačnou logikou môže byť vcelku jednoduchá a na jej dokumentovanie môže postačiť diagram znázorňujúci základné komponenty a ich prepojenie spolu s databázovou schémou. Tvorba takejto dokumentácie je rýchla a jednoduchá práca.

Pri tvorbe dokumentácie je potrebné si uvedomiť k čomu bude slúžiť. Väčšinou hrá dokumentácia veľmi dôležitú úlohu v komunikácii účastníkov vývoja softvéru medzi ktorých patria samotní architekti, návrhári, vývojári, testéri, manažéri projektu, zákazníci a partnerské organizácie. V malých tímoch je často najvhodnejšia osobná komunikácia. V takomto prípade môže byť dokumentácia minimalistická. Vo väčších tímoch, ktorých členovia ani nemusia byť sústredení na jednom mieste je dokumentovanie nevyhnutné na zachytenie nasledovných elementov:

- rozhrania komponentov,
- ohraničenia subsystémov,
- testovacie scenáre,
- rozhodnutia o zakúpení komponentov od tretích strán,
- externé služby ponúkané aplikáciou.

Rozhodnutie čo dokumentovať je náročné. Vždy si treba premyslieť k čomu dokumentácia bude slúžiť a aké sú prípustné náklady na jej tvorbu, čo následne definuje aj aspekty systému, ktoré budeme dokumentovať a na akej úrovni podrobnosti.

2.4.2 UML 2.0

Pri dokumentovaní je dôležité vopred stanoviť spôsob akým sa bude dokumentácia vytvárať a ktoré nástroje pritom budú použité. Architektúru systému je možné zachytiť aj jednoduchým diagramom s využitím notácie s „krabičkami“ a šípkami a legendou opisujúcou význam jednotlivých elementov diagramu.

Samozrejme existuje množstvo spôsobov a pohľadov, ktorými sa dá opísať architektúra. Dominantné postavenie medzi notáciami a jazykmi použiteľnými na zachytenie architektúry systému má jazyk UML, ktorý je podporovaný mnohými komerčnými aj voľne dostupnými nástrojmi.

UML 2.0 pridáva nové vlastnosti a vnáša do jazyka formalizmus pri zachytávaní rôznych aspektov systému. Dôležité je, že zavedená notácia je známa všetkým návrhárom, čo eliminuje viacznačnosti v dokumentácii. Použitie jazyka UML prispieva aj k podpore modelom riadeného vývoja (*model driven development*), keďže z modelov je možné generovať rôzne časti aplikácie.

Pomocou UML je možné zachytiť tak štruktúru ako aj správanie sa systému. Medzi štrukturálne diagramy jazyka UML patria:

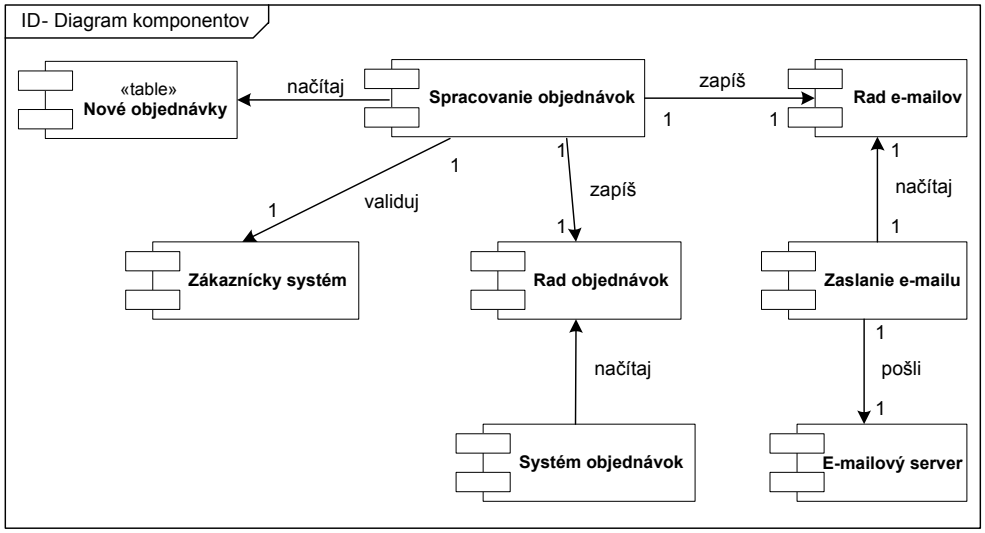
- *Diagram tried* – zobrazuje triedy v systéme a vzťahy medzi nimi.
- *Diagram komponentov* – znázorňuje vzťahy medzi komponentmi s definovanými rozhraniami. Komponenty väčšinou zahŕňajú viacej tried.
- *Diagram balíkov* – rozdeľuje model do skupín elementov a opisuje vzťahy medzi nimi na vysokej úrovni abstrakcie.
- *Diagram rozmiestnenia* – ukazuje ako sú jednotlivé komponenty a ďalšie prvky systému (napr. procesy) fyzicky rozmiestnené na hardvéri.
- *Objektový diagram* – zobrazuje ako medzi sebou súvisia objekty a ako sa používajú počas vykonávania aplikácie.
- *Zložený diagram štruktúry* – stvárňuje vnútornú štruktúru tried alebo komponentov v podobe objektov, z ktorých sa skladajú, a vzťahov medzi nimi.

Medzi diagramy opisujúce správanie sa systému patria:

- *Diagram činností* – definuje programovú logiku a biznis procesy.
- *Diagram komunikácie* – opisuje sekvencie volaní medzi objektmi počas vykonávania aplikácie.
- *Sekvenčný diagram* – zobrazuje sekvenciu výmen správ medzi objektmi.
- *Diagram stavového automatu* – zachytáva vnútorné stavy objektu a udalosti s podmienkami, ktoré majú za následok zmenu tohto stavu.
- *Diagram prehľadu interakcie* – podobný diagramu činností. Je určený na opis toku riadenia pri rôznych scenároch.
- *Diagram časovania* – kombinuje sekvenčný a stavový diagram na opis stavov objektov v čase a správ, ktoré zmenu spôsobujú.
- *Diagram prípadov použitia* – Zachytáva interakciu medzi systémom a prostredím, do ktorého je zasadený spolu s používateľmi a externými systémami.

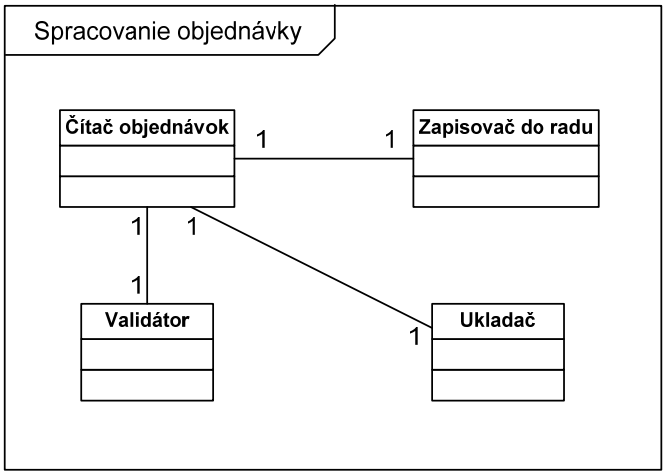
2.4.3 Pohľady na architektúru

V predchádzajúcej kapitole bol spomenutý systém na spracovanie objednávok. Obrázok 2-20 znázorňuje architektúru systému s použitím notácie krabičiek a šípok. Diagram zachytávajúci ten istý pohľad v notácii UML je zobrazený na obrázku 2-21. Na základe vyhodnotenia z predchádzajúcej kapitoly bol kvôli komunikácii do architektúry pridaný rad medzi komponent *Spracovanie objednávok* a *Systém objednávok*.



Obrázok 2-21. Diagram komponentov objednávkového systému v notácii UML.

Iba dva komponenty v architektúre vyžadujú úplnú novú implementáciu. Jedným z nich je komponent *Spracovanie objednávok*. Vnútorná štruktúra tohto komponentu je zobrazená na obrázku 2-22 diagramom tried. Diagram nezobrazuje všetky potrebné triedy, ale len tie najdôležitejšie, čím zostáva stále prehľadný.

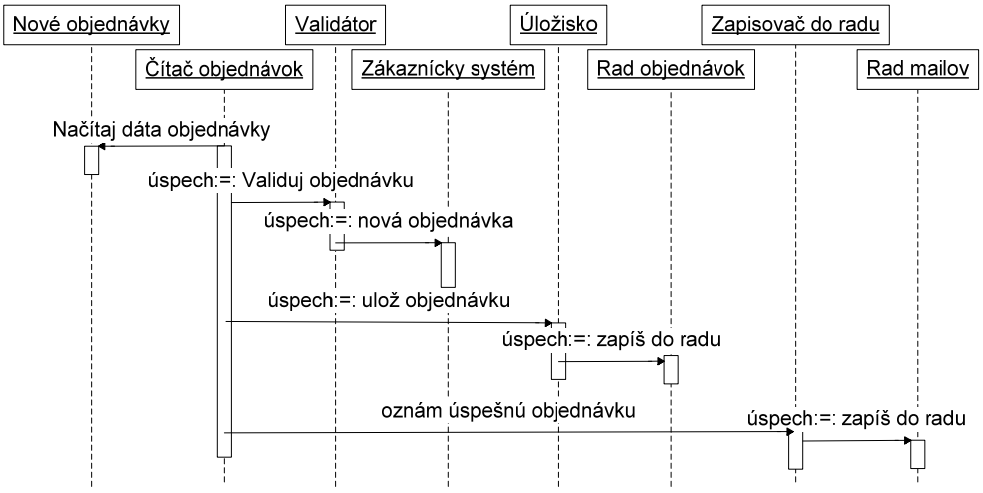


Obrázok 2-22. Triedy realizujúce komponent Spracovanie objednávok.

Na tejto úrovni opisu sme schopní vytvoriť sekvenčný diagram na opis interakcií medzi jednotlivými prvkami architektúry. Tento diagram je v štandardnej notácii UML znázornený na obrázku 2-23. Rovnako ako diagram tried z predchádzajúceho obrázku ani diagram sekvencií nezachytáva všetky detaily a všetky prípady, ktoré môžu nastať počas vykonávania aplikácie. Napríklad nerieši situáciu kedy objednávka neprešla validáciou. Dôvodom vynechania niektorých detailov je znovu snaha o čo najjednoduchší model, ktorý opisuje len podstatné časti systému. Pri vytváraní modelov si vždy musíme uvedomiť, či treba danú časť systému alebo situáciu, ktorá môže počas vykonávania aplikácie nastať, modelovať v danom diagrame. Niektoré skutočnosti postačuje spomenúť v sprievodnom texte k diagramu.

Sekvenčný diagram je jednou z najlepších techník na opis správania sa systému v notácii UML. Pomocou množstva modelovacích prvkov umožňuje opísať zložité spracovanie, ktoré aplikácia realizuje – opakovanie krokov v cykloch, výber ďalšieho kroku z viacerých možností a iné. Prehnané používanie týchto prvkov v snahe zachytiť celé správanie systému však môže mať za následok vytvorenie neprehľadného modelu.

Rozmiestnenie jednotlivých komponentov architektúry do prostredia, kde budú vykonávať svoju činnosť je možné zachytiť diagramom rozmiestnenia. Príklad takého diagramu je znázornený na obrázku 2-24. Diagram znázorňuje alokáciu komponentov – inštancií komponentov na serveroch a vzťahy medzi nimi. Je užitočné popísať vzťahy medzi komponentmi protokolom využívaným na komunikáciu. Napríklad komponent spracovanie objednávok komunikuje pri prístupe do tabuľky nových objednávok v databáze protokolom JDBC.

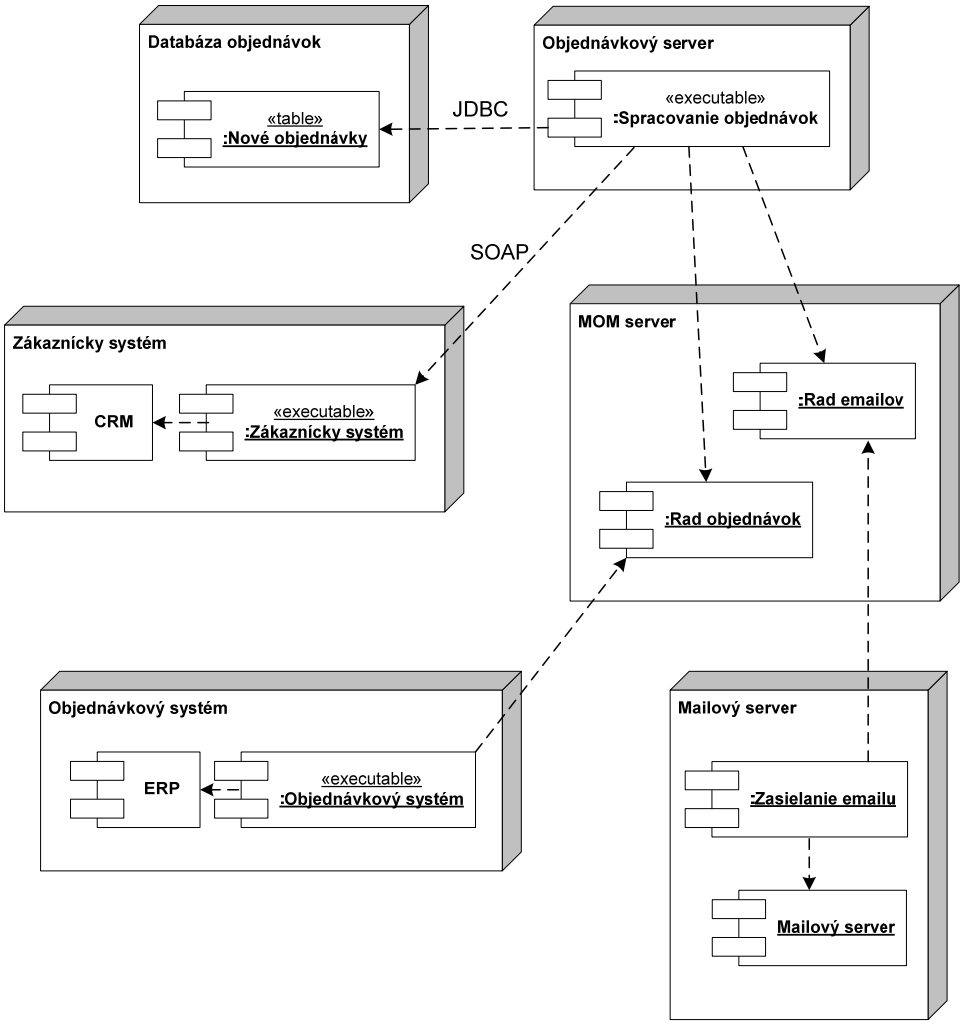


Obrázok 2-23. Sekvenčný diagram pre objednávkový systém.

2.4.4 Diagramy komponentov

Diagramy komponentov sú veľmi užitočné na zachytenie štruktúry architektúry aplikácie, lebo softvérovému architektovi umožňujú zobraziť komponenty, z ktorých sa aplikácia skladá a vzťahy medzi nimi. Jazyk UML 2.0 má dobre prepracovanú notáciu pre zobrazovanie rozhraní komponentov – množiny funkcií, ktoré komponent realizuje

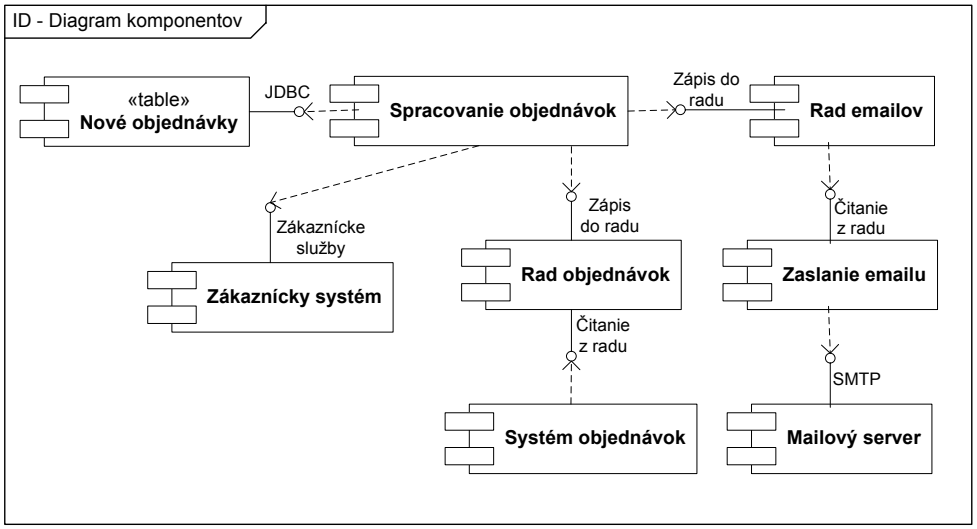
a ktoré poskytuje iným častiam systému a okolitému svetu. Diagram znázorňujúci rozhrania komponentov objednávkového systému je zobrazený na obrázku 2-25.



Obrázok 2-24. Diagram rozmiestnenia pre objednávkový systém.

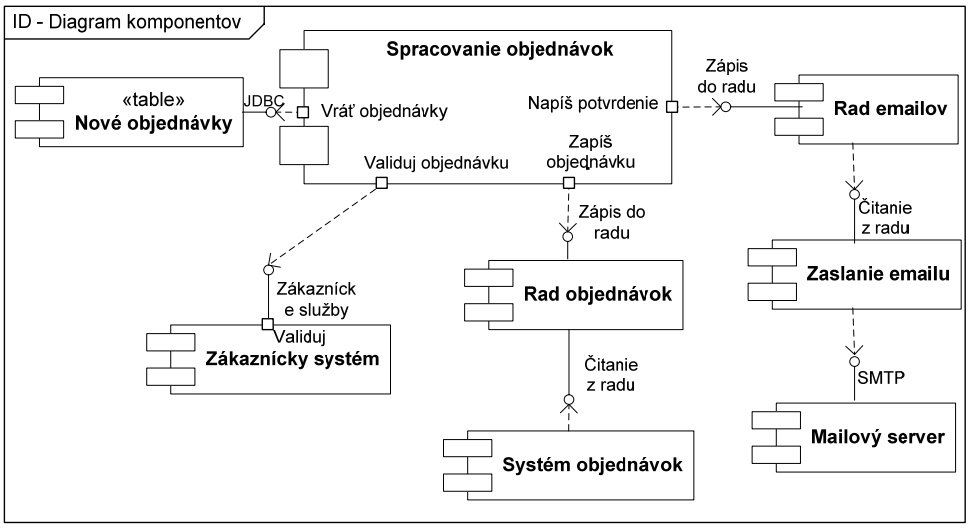
Definovanie rozhraní komponentov je dôležité aj preto, že jednotlivé komponenty sú zväčša vyvíjané rôznymi vývojármi. Jednotlivé komponenty navzájom závisia a ich vývojári potrebujú vedieť ako môžu pristupovať k službám poskytovaným inými komponentmi. Napr. vývojár, ktorý vytvára komponent *Spracovanie objednávok* potrebuje využívať služby zákazníckeho systému. Nepotrebuje však vedieť ako je poskytovanie služieb vo vnútri komponentu *Zákaznícky systém* realizované. Stačí mu vedieť cez aké rozhrania môže k týmto službám pristúpiť a využívať ich. Jednotlivé časti systému sú prepojené cez rozhrania komponentov, čím vytvárajú funkčný celok.

UML 2.0 poskytuje možnosť ďalej špecifikovať rozhrania a vyjadriť spôsob akým sú podporované daným komponentom. To sa realizuje stotožnením rozhrania s portami. Port definuje miesto interakcie medzi komponentom a vonkajším svetom. V notácii UML 2.0 je znázornený malým štvorčekom na hranici komponentu.

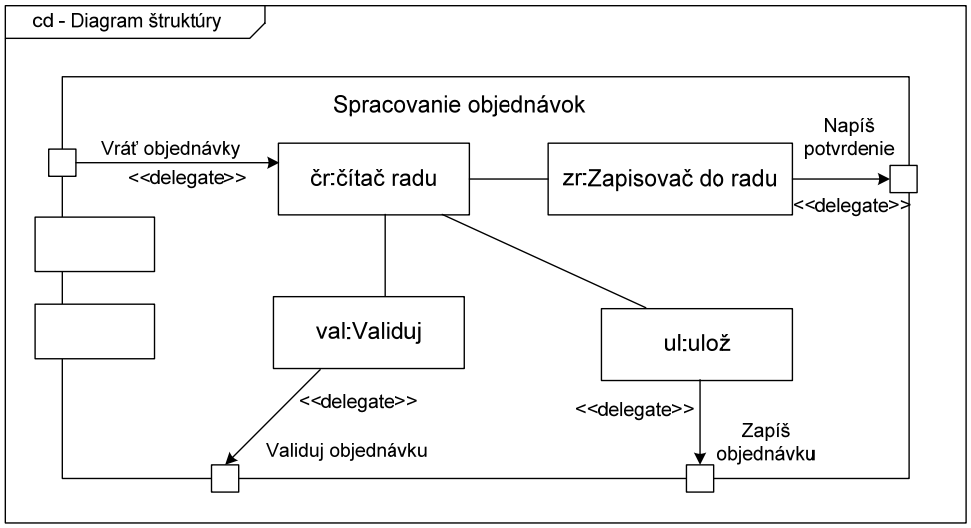


Obrázok 2-25. Reprezentácia rozhraní v objednávkovom systéme.

Obrázok 2-26 znázorňuje diagram komponentov objednávkového systému zobrazujúci aj porty. Ak je niektorý z komponentov zložitý a je potrebné detailnejšie opísať jeho vnútornú štruktúru, môžeme vytvoriť podrobnejší diagram komponentov opisujúci vnútro daného komponentu, čím hierarchicky dekomponujeme systém. Alternatívnym diagramom, ktorý môžeme v takomto prípade použiť, je diagram štruktúry zobrazujúci objekty obsiahnuté v implementácii komponentu, spôsob ich prepojenia a tiež ich napojenie na porty (Obrázok 2-27).



Obrázok 2-26. Použitie portov v diagrame komponentov objednávkového systému.



Obrázok 2-27. Vnútorný návrh komponentu Spracovanie požiadaviek.

2.4.5 Šablóna dokumentácie architektúry

Je dobré ak existuje šablóna dokumentácie na opis požadovaných aspektov systému. Použitie šablóny šetrí čas v počiatočných fázach projektu tým, že poskytuje štruktúru dokumentu, ktorú treba naplniť. Šablóna je tiež užitočná pre nových pracovníkov, ktorí z nej vidia čo je požadované pri vytváraní dokumentácie. Príklad šablóny dokumentácie:

Názov projektu: XXX

1. Kontext projektu
2. Požiadavky na architektúru
   - 2.1. Prehľad kľúčových cieľov
   - 2.2. Prípady použitia architektúry
   - 2.3. Požiadavky zúčastnených strán
   - 2.4. Ohraničenia
   - 2.5. Nefunkcionálne požiadavky
   - 2.6. Riziká
3. Riešenie
   - 3.1. Potenciálne použiteľné architektonické vzory
   - 3.2. Prehľad architektúry
   - 3.3. Štrukturálne pohľady na systém
   - 3.4. Procesné pohľady na systém
   - 3.5. Implementačné aspekty
4. Analýza architektúry
   - 4.1. Scenáre
   - 4.2. Riziká

2.4.6 Zhrnutie

Tvorba dokumentácie je pri tvorbe architektúry softvéru nevyhnutná. Dôležité však je investovať do tejto činnosti primerané časové a finančné zdroje. Je potrebné vytvárať len užitočnú dokumentáciu a vyhýbať sa tvorbe príliš povrchných, prípadne príliš podrobných dokumentov.

Problém nedostatočnej dokumentácie sa často vyskytuje pri voľne dostupných (*open-source*) produktoch. Množstvo dobrých rámcov pre tvorbu aplikácií nie je používaných práve preto, že nie sú dostatočne zdokumentované a je veľmi náročné zistiť ako ich použiť, pričom s primeranou dokumentáciou by tieto rámce mohli byť veľmi užitočné.

Pri dokumentovaní je vhodné používať známu notáciu s dostatočnou vyjadrovacou silou. Medzi v súčasnosti najpoužívanejšie notácie patrí modelovací jazyk UML, ktorým je softvérový architekt schopný opísať tak štruktúru ako aj správanie sa systému. UML poskytuje množstvo pohľadov, ktorými je možné zachytiť rôzne aspekty systému. Jeho výhodou je podpora v mnohých nástrojoch, pomocou ktorých môžeme efektívne vytvárať rôzne diagramy alebo pomocou modelom riadeného vývoja softvéru priamo generovať časti zdrojového kódu z vytvorených modelov.

2.5 Návrh prípadovej štúdie

V tejto kapitole opíšeme návrh prípadovej štúdie pre systém ICDE. Pre lepšie pochopenie samotnej prípadovej štúdie najprv opíšeme technické pozadie systému ICDE. Následne opíšeme samotný návrh systému na základe štruktúry opísanej v predošlej kapitole, okrem opisu kontextu projektu, ktorý bol už vysvetlený v predchádzajúcich kapitolách.

2.5.1 Technické aspekty systému ICDE

Všeobecný opis požiadaviek na systém ICDE a opis cieľov pre tvorbu nasledujúcej verzie systému v predchádzajúcich kapitolách, predstavuje iba začiatočný bod pre návrh samotného riešenia, pred ktorým treba opísať viaceré technické aspekty systému.

Zdieľanie zdrojov viacerými používateľmi

Systém ICDE je možné nasadiť oddelene pre každého používateľa, ktorý bude mať vlastnú serverovú časť s databázou. Alternatívne môžu viacerí používatelia zdieľať jednu inštanciu serverovej časti systému s databázou.

Zdieľanie zdrojov viacerými používateľmi prináša spravidla viaceré výhody v podobe nižších celkových nákladov a jednoduchšej správy systému, najmä kvôli možnosti lepšieho využitia zdrojov a nižšej zložitosti systému. V prípade systému ICDE má zdieľanie zdrojov tieto potenciálne prínosy:

- Zníženie nákladov na databázové licencie; stačí jedna databáza pre celý systém namiesto jednej databázy pre každého používateľa.
- Zníženie požiadaviek na výkonnosť pracovných staníc používateľov – bude sa na nich vykonávať len ICDE klient a nie celá databáza. Zníženie požiadaviek následne zníži náklady potrebné na nasadenie systému.
- Zníženie nákladov na podporu, keďže je potrebné spravovať a sledovať len jeden zdieľaný ICDE server.

Zdieľanie ICDE databázy stále umožňuje použitie tak dvojvrstvovej aj trojvrstvovej klient-server architektúry systému. Dvojvrstvová architektúra by pravdepodobne zabezpečila lepšiu výkonnosť pre malé nasadenia, pričom by súčasne bola jednoduchšia a lacnejšia (nevyžaduje strednú vrstvu). Trojvrstvová architektúra by bola škálovateľnejšia pre väčšie nasadenia (100-150 používateľov), pretože by podporovala znovupoužitie pripojení k databáze (*connection pooling*) a tiež pridanie ďalších zdrojov do strednej vrstvy.

Veľkosť dát

Systém ICDE zachytáva a zaznamenáva veľké množstvo rôznych údajov o aktivite používateľov. Udalosti ako spustenie a ukončenie aplikácií, písanie na klávesnici, pohyb myšou a prístup k Internetu sú zaznamenávané a ukladané do databázy. Samotná databáza je pravidelne vyprázdňovaná (napr. raz za deň/týždeň) za účelom archivácie dát a udržiavania primeranej veľkosti databázy. Aj napriek tomu obsah databázy rastie veľmi rýchlo – veľkosť tabuliek môže v krátkom čase dosiahnuť milióny riadkov.

Samotná databáza nemá problém pracovať s uvedeným množstvom dát, avšak toto predstavuje zaujímavý aspekt návrhu programového rozhrania API systému ICDE. V dvojvrstvovej architektúre ICDE v1.0 nástroje na analýzu dát môže vykonať naivné dopyty do databázy (SELECT * FROM velmiVelkaTabulka), ktoré môžu vrátiť veľmi veľké dátové sady. Tieto dopyty sú pomalé a môžu vyvolať poruchu analytického nástroja tretej strany, ak je výsledok príliš veľký.

V prípade ICDE v1.0, kde každý používateľ má vlastnú databázu je výsledok síce nepríjemný, ale plne v duchu „dostal si, čo si chcel“. Používateľ uškodí len sám sebe a po niekoľkých poruchách sa pravdepodobne poučí. V prípade zdieľania systému viacerými používateľmi, napr. za účelom zníženia nákladov, správanie sa jedného klienta ovplyvní aj ostatných, pričom treba uvažovať:

- výkonnosť databázy,
- spotrebu zdrojov v strednej vrstve pre trojvrstvovú architektúru.

Spotreba pamäte v strednej vrstve predstavuje veľmi významný aspekt návrhu, pretože klienti (používatelia aj nástroje tretích strán) môžu požadovať vykonanie dopytov s veľkým počtom výsledkov. Aj v prípade, že server v strednej vrstve bude mať veľa pamäte, môže viacero zložitých dopytov spôsobiť vyčerpanie zdrojov, zásadné zníženie výkonnosti a zlyhanie dopytov v dôsledku nedostatku pamäte a veľkej doby odozvy. V extrémnych prípadoch môže dôjsť až k poruche celého systému.

Takýto stav je nevyhovujúci, pretože umožňuje nástrojom tretích strán využívajúcich ICDE API spôsobiť nepredvídateľné poruchy systému. Tieto budú vznikať v náhodných stavoch v závislosti od aktuálnej záťaže systému a veľkosti výsledkov. Jediné API volanie by mohlo spôsobiť pád celého servera a znemožniť prácu všetkých používateľov systému.

Notifikácia

Použitie notifikácia v systéme ICDE je vhodné v dvoch prípadoch:

- Keď nástroj tretej strany chce byť informovaný o tom, že používateľ vykonal nejakú akciu, napr. otvoril v prehliadači novú internetovú stránku.

- Keď nástroje tretích strán budú medzi sebou chcieť zdieľať zmysluplné dáta o svojej činnosti, ktoré sú uložené v databáze ICDE. V tomto prípade budú potrebovať mechanizmy pre notifikáciu iných nástrojov o zmenách údajov.

Oba prípady vyžadujú, aby notifikácia bola zaslaná čo najskôr, ideálne v momente nastania danej udalosti. Pri použití dvojvrstvovej architektúry nie je použitie notifikácie prirodzené ani jednoduché. Databázy umožňujú použitie spúšťačov (*trigger*) – postupností operácií, ktoré sa vykonajú, ak dôjde k zmene niektorej tabuľky v databáze (UPDATE, INSERT, DELETE). Databázové spúšťače spravidla využívajú špecifické vlastnosti konkrétnych databázových systémov, čo obmedzuje prenosnosť.

V prípade systému ICDE je kľúčová flexibilnosť výsledného riešenia, pretože nie je možné vopred poznať všetky typy udalostí a dát, ktoré budú nástroje tretích strán používať (aj preto, že ešte neexistujú). Z tohto dôvodu je nutné použiť mechanizmy, ktoré umožnia vývojárom pridávať a zverejňovať nové typy udalostí podľa potreby počas prevádzky systému, bez potreby zmien v samotnom systéme ICDE.

Dátová abstrakcia

Prechod systému ICDE z verzie v1.0 na v2.0 spôsobil zásadné zmeny v databázovej schéme, ktoré boli nutné na zahrnutie nových typov položiek a optimalizáciu internej organizácie databázy kvôli zvýšeniu výkonnosti. API rozhranie ICDE systému skrýva internú organizáciu údajov v systéme pred programátormi nástrojov tretích strán a zabezpečuje tak funkčnosť nástrojov aj v prípade zmeny štruktúry dát. Ak by rozhranie neabstrahovalo od internej štruktúry dát, jednotlivé nástroje by prestali fungovať pri každej (netriviálnej) zmene databázovej štruktúry systému ICDE.

Platformy a distribuovanosť

Systém ICDE musí podporovať tvorbu a prevádzku nástrojov tretích strán aj na iných platformách ako je Windows, napr. na platforme Linux. Niektoré nástroje budú spúšťané lokálne v prostredí Windows, iné bude nutné spúšťať vzdialene, pričom komunikácia bude prebiehať pomocou bežne dostupných prostriedkov – elektronickej pošty, resp. okamžitých správ (*instant messaging*). Dôležité je, aby platforma ICDE podporovala jednoduché využitie relevantných spôsobov komunikácie.

Programové rozhranie API

Rozhranie API umožňuje programový prístup k dátam uloženým v databáze systému ICDE. Táto obsahuje podrobné informácie o rozličných typoch udalostí spolu s časovými pečiatkami ich výskytu. API musí poskytovať aj množinu rozhraní, ktoré umožňujú efektívne dopytovanie nad uloženými dátami. Napr. analytický nástroj môže chcieť vedieť, aké aplikácie používateľ spustil od posledného prihlásenia sa. Ak povedzme otvoril internetový prehliadač, nástroj môže pristúpiť k údajom o prehliadaných internetových stránkach vrátane ich obsahu.

API rozhranie musí tiež umožniť aplikáciám uloženie vlastných údajov do databázy, napr. za účelom zdieľania s inými nástrojmi alebo s používateľom. V takom prípade budú dáta uložené v oddelenej databázovej tabuľke a registrované konzumenty budú notifikované o zmenách v nich.

Predpoklady pre úspešné použitie API rozhrania vývojármi nástrojov tretích strán sú:

- Jednoduchá naučiteľnosť a flexibilita API rozhrania pri tvorbe dopytov.

- Jednoduchá laditeľnosť API rozhrania.
- Transparentnosť vzhľadom na umiestnenie – nástroje tretích strán by nemali závisieť od konkrétnej distribuovanej konfigurácie systému spoliehajúcej sa na konkrétne umiestnenie vybraných komponentov.
- Odolnosť voči zmenám platformy – aplikácie by mali bez zmien fungovať aj pri zmene API rozhrania alebo pri zmene databázy.

Zhrnutie technických aspektov systému ICDE

Možno povedať, že všetky uvedené aspekty a požiadavky na systém ICDE spolu tvoria pomerne komplexnú množinu požiadaviek a pohľadov na vytváraný systém. Požiadavka na notifikáciu o udalostiach nás silne smeruje k flexibilnej architektúre založenej na vzore producent-konzument (*publish-subscribe*). Potreba podpory viacerých platforiem a transparentnosť distribuovanej konfigurácie smeruje k použitiu Javy a komunikácie pomocou protokolov RMI a JMS. Veľkosť dát a abstrakcia prístupu k nim si pravdepodobne vyžiada trojvrstvovú architektúru so strednou vrstvou, ktorá bude prekladať API volania na SQL dopyty a súčasne riadiť bezpečný prístup k (veľkým) výsledkom dopytov.

2.5.2 Požiadavky na architektúru systému ICDE

Kľúčovým cieľom ICDE v2.0 je vytvorenie infraštruktúry poskytujúcej programové rozhranie pre prístup nástrojov tretích strán k dátam uloženým v ICDE databáze. Táto musí spĺňať tieto hlavné požiadavky:

- Flexibilitu výberu platformy a možností nasadenia a konfigurácie nástrojov tretích strán.
- Musí poskytovať rámec, ktorý umožní dynamické pridávanie aplikácií tretích strán do systému, s možnosťou okamžitého získania informácií o akciách používateľov. Rámec súčasne musí umožňovať zdieľanie informácií s inými nástrojmi v prostredí a tvorbu výstupu pre analytikov.
- Poskytovať jednoduchý prístup na čítanie a zápis dát z/do ICDE databázy.

Sekundárnym cieľom je upraviť architektúru ICDE systému tak, aby bola škálovateľná na nasadenia v rozsahu 100-150 používateľov s nízkymi nákladmi na nasadenie dodatočných pracovných staníc.

Prípady použitia architektúry

Po diskusii s niekoľkými potenciálnymi tvorcami nástrojov tretích strán boli identifi—kované dva základné prípady použitia API rozhrania systému ICDE:

- *Prístup k dátam v databáze ICDE:* Väčšina dopytov nástrojov tretích strán sa zameriava na aktivity jedného konkrétneho používateľa systému. Na začiatku aplikácia zistí projekt, na ktorom práve pracuje. Následne postupne načítava detailné údaje o jeho činnosti v časovej následnosti ako nastali a vyhľadáva špecifické položky (napr. názvy okien alebo stlačené klávesy). Zistené údaje použije na inicializáciu iného nástroja alebo na tvorbu výstupu pre používateľa.
- *Uloženie dát do databázy ICDE:* Nástroje tretích strán ukladajú do databázy ICDE údaje o svojej aktivite, ktoré zdieľajú s inými nástrojmi. Potrebný je tiež notifikačný mechanizmus, pomocou ktorého nástroje oznamujú existenciu

nových údajov. Každý nástroj môže vytvoriť veľké množstvo najrôznejších dát, ktoré je nutné opísať metadátami, aby ich mohli použiť iné nástroje.

Požiadavky zúčastnených strán

Požiadavky vývojárov analytických nástrojov tretích strán:

- *Jednoduchosť prístupu k dátam:* Schéma relačnej databázy obsahuje približne 50 tabuliek s komplexnými vzťahmi, pričom sa nevyhne budúcim zmenám v dôsledku zmeny funkcionálnych požiadaviek. Systém ICDE musí zakryť zložitosť databázy pred nástrojmi tretích strán a zabezpečiť, že zmeny v databáze nespôsobia nefunkčnosť existujúcich nástrojov.
- *Podpora rôznych platforiem:* Systém ICDE v1.0 bol silne závislý od platformy Windows. ICDE v2.0 musí zabezpečiť prístup k dátam aj pre nástroje na iných platformách spolu s možnosťou ich začlenenia do spoločného prostredia.
- *Okamžitá notifikácia o udalostiach:* Keďže nástroje tretích strán chcú poskytovať rýchlu odozvu používateľom, musí ICDE systém poskytovať mechanizmus pre okamžitú notifikáciu nástrojov o aktivitách používateľov v systéme.

Požiadavky API programátorov, podľa ktorých API musí:

1. Byť jednoducho a intuitívne naučiteľné.
2. Zabezpečiť dobrú čitateľnosť a modifikovateľnosť kódu, ktorý ho používa.
3. Poskytovať konzistentný a jasný spôsob implementácie štandardných prípadov použitia, ktoré načítavajú údaje z ICDE databázy.
4. Umožniť zápis vlastných typov dát a metadát do ICDE databázy za účelom komunikácie nástrojov prostredníctvom systému ICDE.
5. Poskytovať prostriedky na prehliadanie údajov v ICDE databáze nezvyčajnými a nepredpokladanými spôsobmi tak, aby nebránilo „kreatívnym“ spôsobom prístupu k dátam.
6. Mať dobrú výkonnosť a vracať výsledky v krátkom čase (ideálne do 1-5 sekúnd) na typickom hardvérovom vybavení, čo umožní krátku dobu odozvy aplikácie.
7. Byť flexibilné vzhľadom na možnosti nasadenia a distribúcie komponentov pre malé aj veľké nasadenia.
8. Byť prístupné pomocou Java API.

Požiadavky vývojového tímu ICDE, podľa ktorých architektúra musí:

1. Plne abstrahovať od databázovej schémy a implementácie servera, pričom bude izolovať nástroje tretích strán od zmien v nich.
2. Podporovať jednoduchú modifikovateľnosť servera s minimálnym dopadom na existujúce aplikácie využívajúce ICDE API.
3. Podporovať súbežný prístup viacerých vláken a aplikácií vykonávaných v rôznych procesoch, resp. na rôznych strojoch.
4. Byť dobre dokumentovateľná a jednoznačne pochopiteľná pre API vývojárov.
5. Byť dobre škálovateľná vzhľadom na výkonnosť bez potreby zmien v implementácii API, pomocou škálovania do šírky alebo do výšky.

6. Minimalizovať alebo úplne vylúčiť schopnosť nástrojov tretích strán spôsobiť poruchu servera. API musí zabezpečiť ošetrenie zlých parametrov a predísť nadmernej spotrebe systémových zdrojov.
7. Byť primerane testovateľná tak, že testovací tím dokáže zostaviť testovaciu sadu s vysokým pokrytím testovacích prípadov pre automatizované testovanie API.

Ohraničenia a nefunkcionálne požiadavky

Ohraničenia na architektúru vyžadujú použitie databázovej schémy pre ICDE v2.0 a nasadenie prostredia ICDE v2.0 na platforme Windows.

Hlavné nefunkcionálne požiadavky sú:

- *Výkonnosť:* ICDE v2.0 by malo mať dobu odozvy do 5 sekúnd pri vykonaní API dopytov, ktoré vrátia do 1000 položiek na „typickom" hardvéri.
- *Spoľahlivosť:* ICDE v2.0 by malo byť odolné voči poruchám spôsobeným nástrojmi tretích strán, napr. pomocou volaní so zlými parametrami alebo v dôsledku vyčerpania zdrojov. Zvýšenie spoľahlivosti zníži náklady na podporu aplikácie. V prípade nutnosti kompromisu medzi výkonnosťou a spoľahlivosťou by mala byť uprednostnená spoľahlivosť systému.
- *Jednoduchosť:* Keďže konkrétne požiadavky na API sú veľmi nepresné, mal by byť preferovaný jednoduchý návrh založený na rozšíriteľnej architektúre. Jednoduchý návrh je ľahšie realizovateľný, spoľahlivejší a modifikovateľnejší. Požiadavka na jednoduchosť návrhu súčasne zabezpečí, že (zbytočná) flexibilnosť a optimalizácia na prácu s veľkým množstvom údajov sa nedostanú do návrhu pokiaľ si ich nevynútia konkrétne prípady použitia.

Riziká

Najväčšie riziko spojené s návrhom systému ICDE predstavuje nesprávne odhadnutie požiadaviek nástrojov tretích strán, pretože len minimum potenciálnych vývojárov týchto nástrojov ich dokáže (v súčasnosti) špecifikovať.

Vhodným opatrením na zníženie tohto rizika je použitie jednoduchého a ľahko rozšíriteľného počiatočného programového rozhrania API, ktoré bude rozšírené po identifikovaní dodatočných požiadaviek vyplývajúcich z nových prípadov použitia.

2.5.3 Architektúra systému ICDE

Návrh architektúry systému ICDE vychádza z troch architektonických vzorov:

- Trojvrstvová klient-server architektúra – nástroje tretích strán predstavujú klientov, API rozhranie tvorí strednú vrstvu, ktorá zabezpečuje perzistenciu pomocou dátového úložiska ICDE v2.0.
- Producent-konzument (*publish-subscribe*) – stredná vrstva podporuje prihlásenie sa konzumentov na odber udalostí a zverejňovanie udalostí od producentov.
- Vrstvy – klient a stredná vrstva z trojvrstvovej klient-server architektúry sú interne členené na viaceré vrstvy.

Obrázok 2-28 znázorňuje prehľad výslednej architektúry systému ICDE v2.0. Klienty systému ICDE využívajú komponent ICDE API Klient na volanie služieb ICDE API, ktorý prekladá volania na JDBC volania do dátového úložiska. Služby ICDE API sú

spolu s notifikačnou službou JMS (*Java Messaging Service*) a službami pre zber dát vykonávané na J2EE aplikačnom serveri.

Závislosť pôvodných služieb pre zber dát z ICDE v1.0 na dátovom úložisku bola odstránená pomocou ich refaktorizácie, pričom všetky operácie pre prístup k dátam boli presunuté do J2EE komponentov. Tieto umožňujú zaznamenanie udalostí viacerými používateľmi súčasne a tiež podporujú súčasnú prístup viacerých nástrojov tretích strán k dátam v úložisku.

Štrukturálne pohľady na systém ICDE

Obrázok 2-29 znázorňuje diagram komponentov API rozhrania systému ICDE, ktorý obsahuje rozhrania a závislosti jednotlivých komponentov:

- *ICDE Nástroj tretej strany* využíva rozhranie komponentu ICDE API Klient na volania API pre dopytovanie nad dátovým úložiskom, pre zápis do úložiska a pre prihlásenie sa na odber udalostí zverejnených pomocou JMS. Súčasne musí poskytovať rozhranie pre notifikačné volanie, ktoré využíva ICDE API Klient pre notifikáciu o zverejnených udalostiach.
- *ICDE API Klient* implementuje klientsku časť API rozhrania, pomocou ktorej prekladá požiadavky nástrojov tretích strán na EJB volania serverovej časti API. Súčasne transformuje výsledky EJB volaní serverovej časti a vracia ich nástrojom tretích strán. Pomocou zapuzdrenia všetkých J2EE volaní tento komponent izoluje nástroje tretích strán od zložitosti používania aplikačného serveru (lokalizácia, ošetrenie výnimiek a veľkých množín výsledkov). ICDE API Klient súčasne sprostredkuje prihlásenie sa nástrojov tretích strán na odber udalostí a notifikáciu cez JMS.
- Komponent *Služby ICDE API* sa skladá z bezstavových EJB session bean pre prístup k dátovému úložisku pomocou JDBC. Komponent Zápis umožňuje klientom definovať tému (*topic*), do ktorej bude vybraná udalosť zverejnená pomocou JMS.
- *ICDE Dátové úložisko* predstavuje ICDE v2.0 databázu.
- *JMS* predstavuje štandardnú J2EE službu pre posielanie správ (*Java Messaging Service*), pomocou ktorej sú zverejňované udalosti v danej množine tém.

Obrázok 2-30 znázorňuje diagram komponentov pre zber dát:

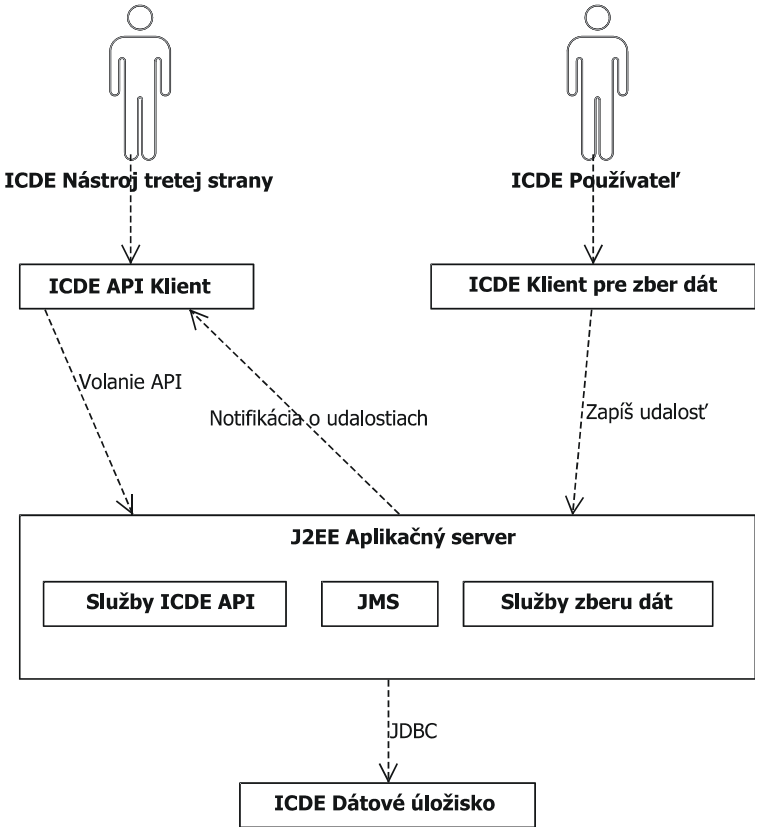
- *ICDE Klient pre zber dát* je súčasťou aplikácie na strane klienta, pričom zapuzdruje všetku interakciu s J2EE aplikačným serverom. Komponent zabezpečuje zaznamenanie udalostí na strane klienta pomocou volania Služieb pre zber dát cez príslušné API rozhranie.
- Komponent *Služby zberu dát* sa skladá z bezstavových EJB session bean, ktoré ukladajú informácie o udalostiach do dátového úložiska ICDE. Pre vybrané typy udalostí komponent zabezpečuje ich zverejnenie pomocou služby JMS.
- *Zverejnenie udalosti* zabezpečuje zverejnenie vybraných udalostí (nie všetky udalosti sú zverejnené, napr. pohyb myšou) do vopred definovanej množiny tém, pričom notifikácia na udalosť je zaslaná každému ICDE API Klient komponentu, ktorý je prihlásený na jej odber.

Priradenie jednotlivých komponentov na konkrétne uzly systému ICDE znázorňuje diagram rozmiestnenia (obrázok 2-31). Pre jednoduchosť diagram znázorňuje len jedného používateľa a jeden nástroj tretej strany aj keď aplikačný server umožňuje prácu viacerých klientov súčasne.

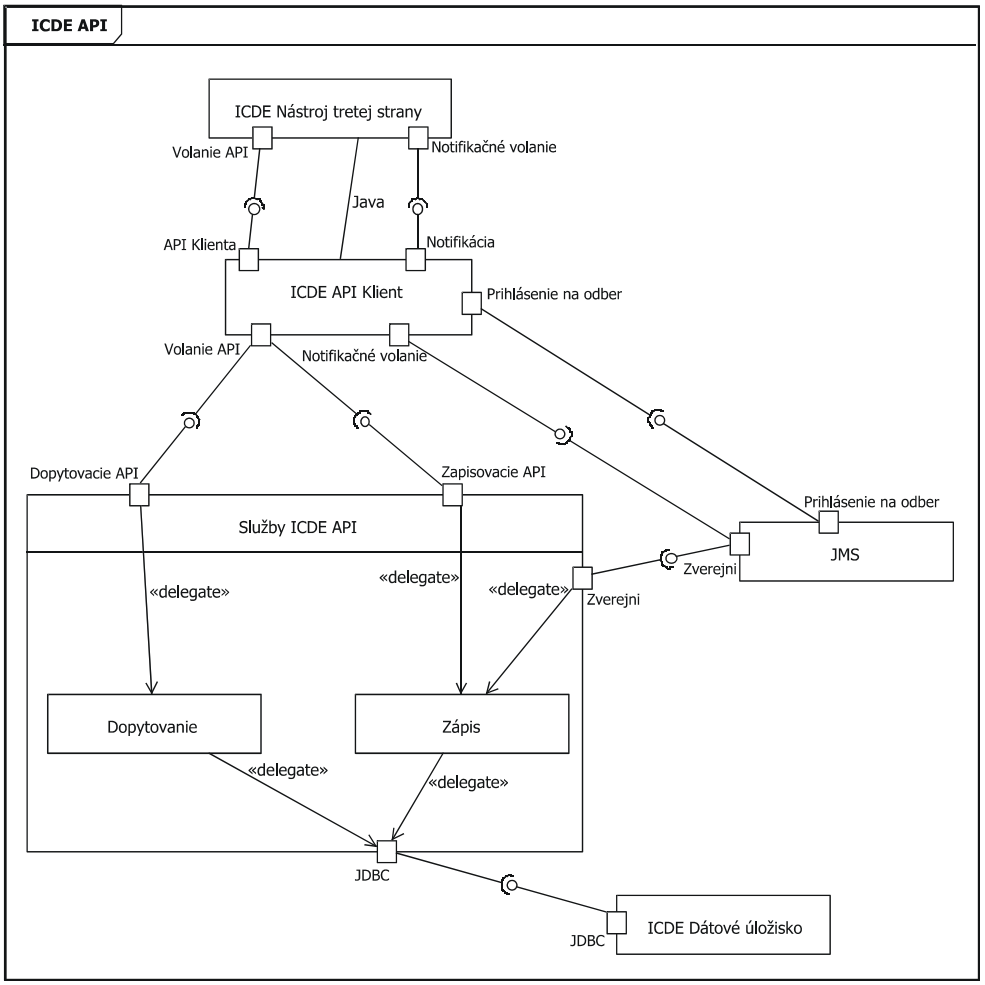
Diagram znázorňuje nasadenie nástrojov tretích strán na inom uzle ako je pracovná stanica klienta, avšak v praxi môžu byť tieto nasadené aj priamo na pracovnej stanici klienta. Každý ICDE nástroj obsahuje vlastnú inštanciu komponentu ICDE API Klient, ktorý je ako knižnica jar zahrnutý priamo v nástroji.

Procesné pohľady na systém ICDE

Správanie sa systému opíšeme pomocou sekvenčných diagramov. Obrázok 2-32 znázorňuje použitie dopytovacieho API. Nástroje musia API najprv explicitne inicializovať, čo zabezpečí inicializáciu EJB session bean v komponente ICDE API Klient pomocou J2EE adresárovej služby (JNDI).



Obrázok 2-28. Architektúra systému ICDE.



Obrázok 2-29. Diagram komponentov API rozhrania systému ICDE.

Po úspešnej inicializácii API umožňuje nástrojom vykonávať volania na dopytovanie nad dátovým úložiskom ICDE cez volanie služieb ICDE API. Tieto môžu potenciálne vrátiť veľké množstvo výsledkov v závislosti od konkrétneho dopytu a preto ICDE API Klient umožňuje postupne spracúvať výsledky veľkých dopytov po častiach, pomocou vzoru iterovania po stránkach výsledkov (*page-by-page iterator*) a súčasne zvyšuje spoľahlivosť pomocou včasného uvoľňovania zdrojov po každom dopyte.

ICDE API Klient pri volaní služieb ICDE API definuje maximálny počet výsledkov (typicky 1000) a počiatočný index, pričom pokiaľ zostali ešte nejaké nespracované dáta, vykoná opätovné volanie s vyšším indexom až po získanie všetkých dostupných dát. Agregovanú dátovú sadu následne vráti nástroju tretej strany na spracovanie, čím skrýva zložitosť získavania veľkého množstva údajov pred aplikačnými programátormi.

Obrázok 2-33 znázorňuje spôsob volania zapisovacieho API, pomocou ktorého môžu nástroje tretích strán zapísať údaje do dátového úložiska ICDE, pričom môže definovať, či a do akej témy má byť udalosť zverejnená.

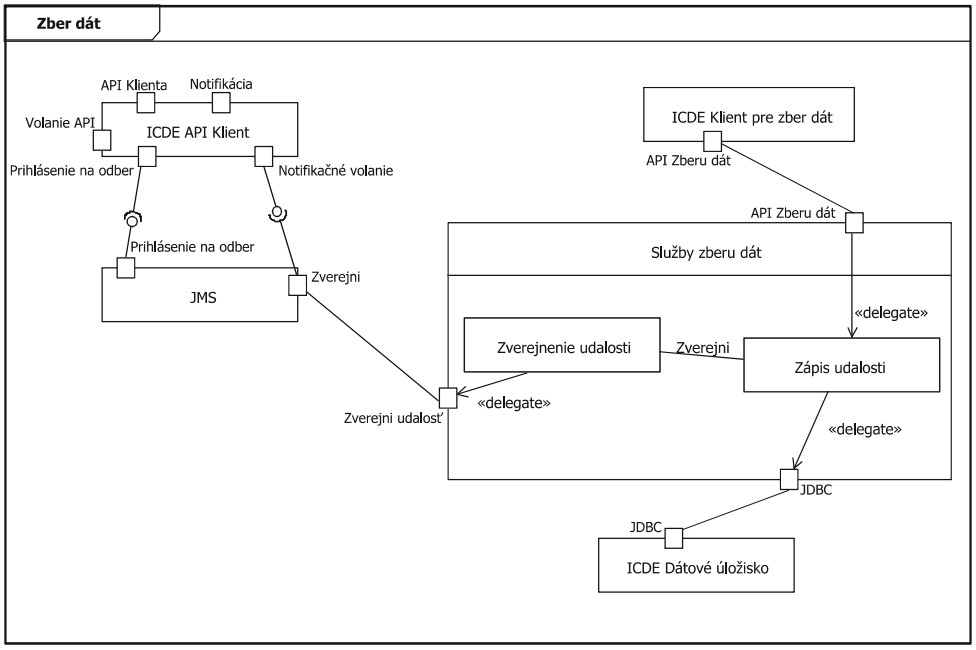
Obrázok 2-34 znázorňuje záznam udalosti zachytenej na strane klienta. Na zaznamenanie udalosti je nutné použitie transakcie (otvorenie a ukončenie transakcie), pretože zaznamenanie udalosti môže vyžadovať viacero samostatných JDBC volaní.

Ak je udalosť nastavená ako zverejňovaná, tak je po úspešnom zázname udalosti do dátové úložiska zverejnená pomocou služby JMS. Volanie JMS je vykonávané mimo transakcie, aby sa predišli réžii pri realizácii dvojfázového databázového volania (*two-phase commit*). Toto riešenie predstavuje kompromis v návrhu za účelom zlepšenia výkonnosti za cenu možného zlyhania zverejnenia udalosti cez službu JMS po jej zaznamenaní do databázy. Aj keď je tento stav neželateľný, nespôsobí vážne problémy v kontexte ICDE aplikácie (pravdepodobnosť zlyhania JMS je nízka).

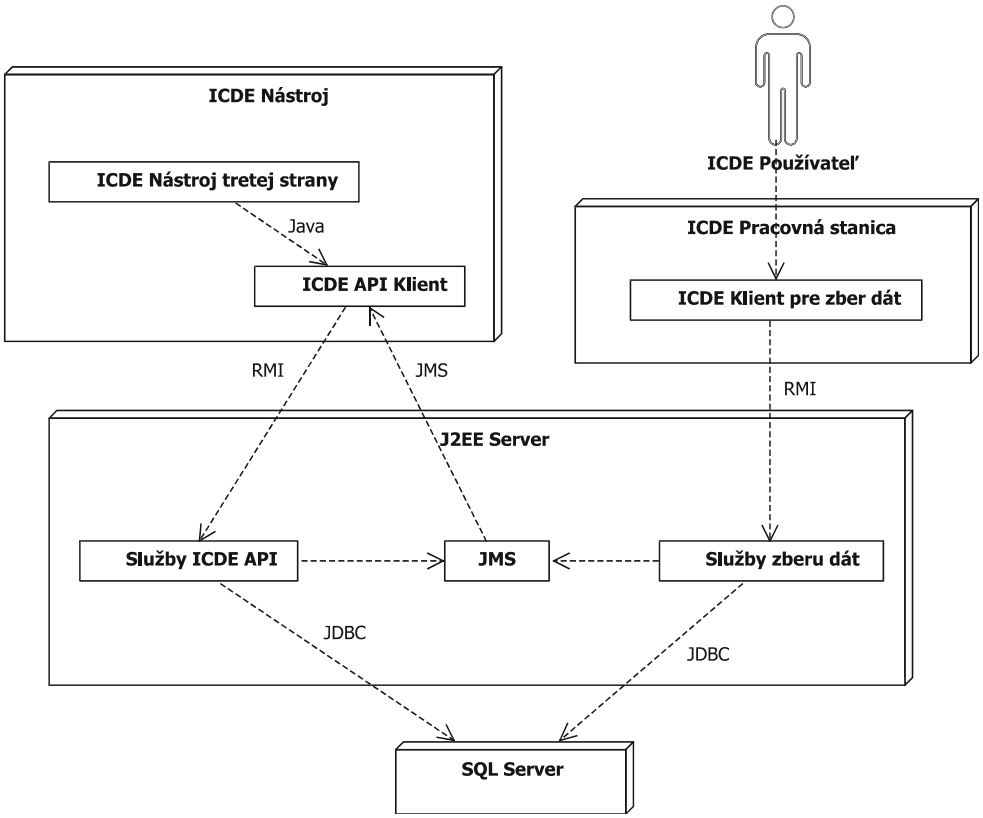
Implementačné aspekty

Ako implementačná platforma pre systém ICDE v2.0 bola vybraná Java 2 Enterprise Edition (J2EE), pretože je platformovo neutrálna čo spĺňa požiadavku na podporu heterogénnych platforiem. Súčasne existuje viacero kvalitných open-source verzií, ktoré sú vhodné pre minimalizáciu nákladov nasadenia, ako aj viacero výkonných komerčných verzií pre veľké nasadenia vyžadujúce vysokú spoľahlivosť. J2EE platforma obsahuje aj zabudovanú podporu pre distribuované systémy, notifikáciu a prístup k databázam. Medzi ďalšie implementačné aspekty patria:

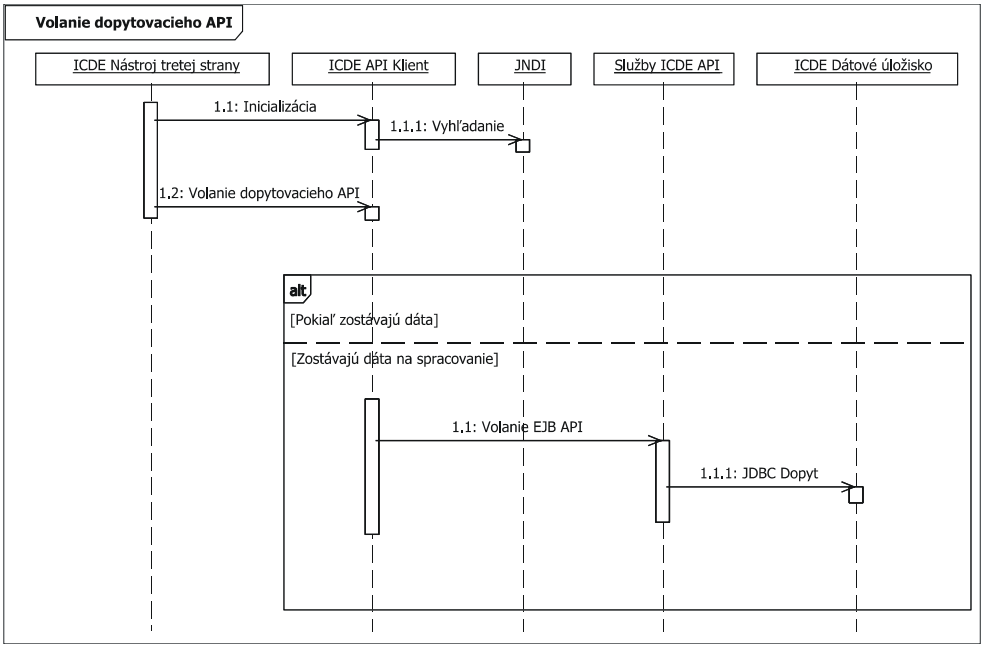
- *Viacvláknovosť:* Komponent ICDE API Klient by mal byť implementovaný tak, aby umožňoval súčasné vykonávanie viacerých vláken, čo umožní vývojárom nástrojov bezpečne vytvoriť viacero vláken a volať viacero súbežných volaní API.
- *Bezpečnosť:* Nástroje ICDE používajú na autentifikáciu používateľské meno a heslo. API rozhranie podporuje funkciu *login*, ktorá overí meno a heslo oproti údajom z databázy ICDE. Následne API umožní prístup k vybranej časti databázy ICDE tak ako v ICDE v1.0.



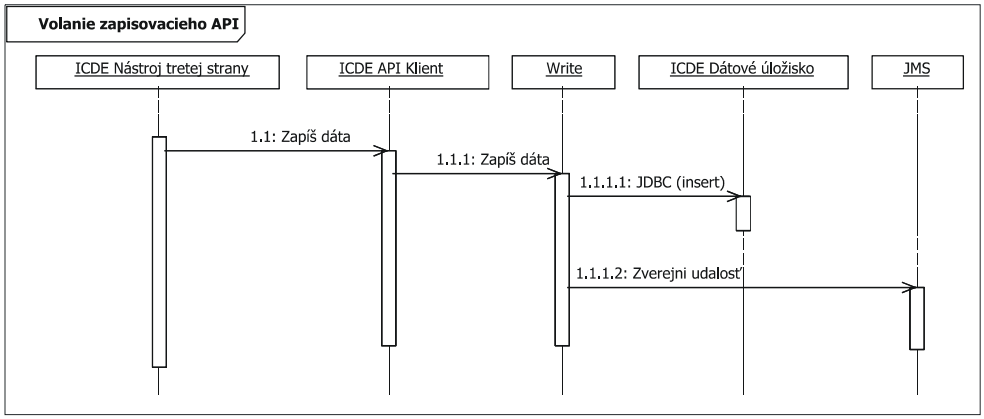
Obrázok 2-30. Diagram komponentov pre zber dát v systéme ICDE.



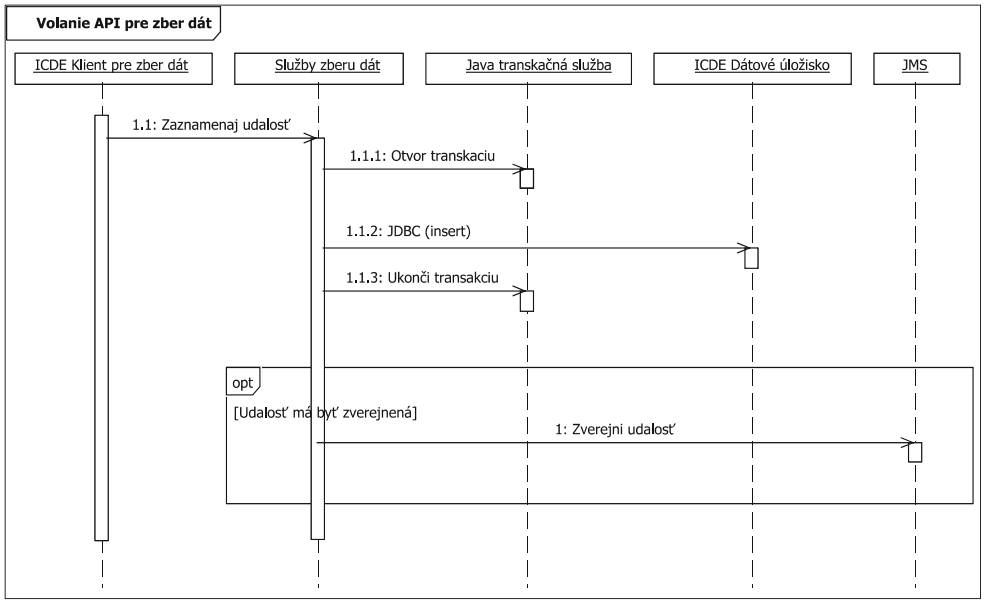
Obrázok 2-31. Diagram rozmiestnenia systému ICDE.



Obrázok 2-32. Sekvenčný diagram pre volanie dopytovacieho API.



Obrázok 2-33. Sekvenčný diagram pre volanie zapisovacieho API.



Obrázok 2-34. Sekvenčný diagram pre volanie API pre zber dát.

- *Enterprise Java Beans (EJB):* Session beany priamo pristupujú k databáze pomocou JDBC volaní. Tieto boli implementované v dvojvrstvovej architektúre ICDE v1.0 – ich priame použitie a refaktorizácia znižujú potrebné náklady.

2.5.4 Analýza architektúry systému ICDE

Pri analýze možných budúcich zmien v systéme ICDE sme zohľadnili tieto scenáre:

- *Zmena organizácie dátového úložiska:* Zmeny v databáze budú vyžadovať zmeny v EJB komponentoch na strane servera. Štrukturálne zmeny, ktoré nepridávajú nové dátové atribúty, sú celé zahrnuté v týchto komponentoch a neovplyvňujú ICDE API. Zmeny pridávajúce nové atribúty vyžadujú aj zmenu API rozhraní serverových komponentov. Použitie verziovania rozhraní a označovanie zasta-

raných metód (*deprecated method*) umožní riadenie vplyvu zmien na komponenty nástrojov tretích strán.

- *Zmena architektúry na iného J2EE dodávateľa:* Zmena platformy nepredstavuje závažný problém, pokiaľ bude ICDE aplikácia implementovaná podľa J2EE štandardov a nebude nepoužívať platformovo špecifické rozšírenia dodávateľa J2EE. Praktické skúsenosti ukazujú, že v takom prípade je možné preniesť aplikáciu z jedného aplikačného serveru na iný s minimálnym úsilím (do týždňa). Problémy sa najčastejšie vyskytujú pri konfigurácii a použití špecifických nastavení konfigurácie nasadenia (*deployment descriptor*).
- *Škálovanie nasadenia na 150 používateľov:* Škálovanie vyžaduje dôkladné plánovanie kapacity na základe špecifikácie dostupného hardvéru a sieťového vybavenia. J2EE server umožňuje jednoduché klastrovanie a replikáciu vďaka použitiu bezstavových session bean. Pre 150 používateľov bude nutné použiť výkonný databázový server, pričom môže byť vhodné rozdeliť dátové úložisko na dve fyzické databázy.

Medzi možné riziká pri zvládnutí zmien patria vysoké náklady potrebné na plánovanie kapacity pre veľké nasadenie a nevhodnosť API rozhrania pre meniace sa požiadavky nástrojov tretích strán.

Vhodným opatrením pre problémy s plánovaním kapacity je vykonanie skorého testovania výkonnosti a záťaže systému ICDE a poskytnutie konkrétnych údajov o výkonnosti klientom, čo umožní presnejšie plánovanie kapacity pre nasadenie systému ICDE.

Podobne zverejnenie API rozhrania a získanie skorej spätnej väzby od dodávateľov nástrojov tretích strán umožní prispôsobiť a rozšíriť API rozhranie a celkový návrh podľa požiadaviek jednotlivých zúčastnených strán.

2.5.5 Zhrnutie

Na príklade prípadovej štúdie systému ICDE sme demonštrovali prácu softvérového architekta. Opísali a zdokumentovali sme rozhodnutia pri návrhu systému ICDE v2.0, pričom našim cieľom bolo sprostredkovať čitateľovi zmýšľanie architekta a analýzu nutnú pre tvorbu samotného návrhu. Aj keď sme z priestorových dôvodov vynechali niekoľko detailov, uvedený rozsah dokumentácie návrhu systému je vhodný pre väčšinu bežných systémov. Samotný systém ICDE predstavuje aplikáciu strednej zložitosti a je tak ukážkovým príkladom každodennej práce softvérového architekta.

Použitá literatúra

CHUNG, L. NIXON, B., YU, E., MYLOPOULOS, J. (EDS.) (1999). *Non-Functional Requirements in Software Engineering Series:* The Kluwer International Series in Software Engineering. Vol. 5, Kluwer Academic Publishers. 1999.

RAMACHANDRAN, J. (2002). *Desighning Security Architecture Solutions*. Wiley & Sons, 2002.

GORTON, I., ZHU, L. (2005). *Tool Support for Just-in-Time Architecture Reconstruction and Evaluation: An Experience Report*. International Conference on Software Engineering (ICSE) 2005, St Louis, USA, ACM Press.

3 PRÍSTUPY K TVORBE ARCHITEKTÚRY SOFTVÉRU

3.1 Pohľad do budúcnosti

Softvérové technológie sú veľmi rýchle a neustále meniace sa odvetvie. Tak ako sa zlepšujú naše inžinierske vedomosti, metódy a nástroje v tejto oblasti, zlepšuje sa aj naša schopnosť pochopiť a riešiť stále zložitejšie a zložitejšie problémy. To znamená, že vytvárame „lepšie a väčšie“ aplikácie na hrane našich neustále sa zlepšujúcich inžinierskych zručností. Preto nie je prekvapujúce, že mnohí z odvetvia nepociťujú prínos nových a zlepšených vývojových prístupov ale majú pocit, že stojíme na mieste.

Je veľmi dôležité zamyslieť sa nad tým, aké budú hlavné výzvy a problémy tvorcov softvérových systémov v najbližších rokoch. Veľmi pravdepodobne bude pokračovať nárast zložitosti obchodných aplikácií mnohými smermi. Zložitosť treba preto chápať ako mnoho-rozmerný atribút. Ktorý aspekt zložitosti však bude ten, ktorý zásadne ovplyvní dizajn a vývoj novej generácie softvérových aplikácií?

Z pohľadu firiem ovplyvnia prácu softvérových inžinierov v najbližšej dekáde tieto aspekty:

- Spoločnosti budú od softvérovej infraštruktúry vyžadovať podporu čoraz komplexnejších obchodných procesov, čo povedie k zvýšenej efektivite organizácie a zníženiu nákladov.
- V mnohých spoločnostiach si intenzita zmien v prostredí obchodu vyžiada ľahko a rýchlo adaptovateľné softvérové systémy.
- Spoločnosti sa na jednej strane vždy usilovali a budú usilovať o zvýšenie prínosu softvéru a súčasne na strane druhej o zníženie nákladov na softvér.

V ďalšej časti textu postupne preberieme jednotlivé body spolu s ich možnými dôsledkami, špeciálne z pohľadu softvérovej architektúry.

3.1.1 Zložitosť obchodných procesov

Vo veľkých spoločnostiach zasahujú dôležité obchodné procesy do viacerých oblastí obchodovania podporované nezávislými aplikáciami vo vysoko heterogénnej IT infraštruktúre. V takomto prostredí nadobúdajú nástroje a technológie pre definovanie a ukotvenie obchodných procesov kriticky dôležitý význam. Inak povedané to znamená, že technológie orchestrácie obchodných procesov sa stávajú kritickými komponentmi v mnohých spoločnostiach.

Dnešné nástroje orchestrácie obchodných procesov sú dostatočne vyskúšané a efektívne technológie nasadené v praxi. Najvyspelejšie z nich podporujú požiadavky na zvýšenie záťaže a škálovateľnosti systémov. Existuje ale niekoľko fundamentálnych problémov, ktoré presahujú ich schopnosti. K riešeniu týchto problémov bude pravdepodobne kľúčové presunúť sa zo „statických“ na „dynamické“ procesy. Čo to presne znamená?

Veľmi atraktívnym zámerom pre obchodné procesy je dynamická kompozícia. Napríklad organizácia môže mať definovaný proces nákupu výrobných komponentov od dodávateľov. Nečakane jeden z dodávateľov skončí s obchodovaním a ďalší zvýši ceny nad hranicu, ktorú je spoločnosť ochotná zaplatiť. So súčasnými technológiami treba vykonať zmenu manuálne, aby mohla spoločnosť začať komunikovať s novým dodávateľom. Toto je nákladné a pomalé.

Ideálne by malo byť, aby bol obchodný proces schopný automatickej rekonfigurácie samého seba, dodržaním definovanej množiny obchodných pravidiel pre spojenie s novým dodávateľom a obnoviť nákupný vzťah. Toto všetko by sa vykonalo za pár sekúnd bez nutnosti väčšieho zásahu programátorov.

Rozvoj takéhoto typu dynamických obchodných procesov nie je veľmi náročný ak ide o extrémne obmedzenú a dobre zadefinovanú oblasť. Ak máme pevnú množinu potenciálnych partnerov u ktorých poznáme ich rozhrania, alebo ideálne ak sú ich rozhrania rovnaké, potom môžeme vytvoriť obchodný proces, ktorý dokáže reagovať na prípadné neočakávané zmeny (napr. náhla nedostupnosť rozhrania obchodného partnera). Avšak ako náhle odstránime obmedzenia v danej oblasti, celý problém bude exponenciálne zložitejší.

Ak nie sú vhodní obchodní partneri vopred známi, je nutné aby ich bol obchodný proces schopný vyhľadať. Táto podmienka vyžaduje existenciu adresára či registra, v ktorom sa dá vyhľadávať na základe viacerých vlastností. Ak výsledok vyhľadávania vráti viac ako jedného potenciálneho obchodného partnera, treba rozhodnúť ako a ktorého proces vyberie? Ako bude proces vedieť, ktorý obchodný partner bude poskytovať služby na požadovanej úrovni spoľahlivosti a bezpečnosti? Preto musí vzniknúť mechanizmus, ktorý dokáže opísať úroveň poskytovaných služieb a zabezpečiť dôveryhodnosť pre všetky spomínané mechanizmy.

Ihneď po selekcii dôveryhodného partnera na základe úrovne poskytovaných služieb je nevyhnutné určiť presnú komunikáciu s partnerom. Nikde nie je garantované, že každý možný partner bude mať rovnaké rozhranie, že bude akceptovať a chápať rovnakú množinu správ. Je preto nevyhnutné, zabezpečiť pre žiadajúci obchodný proces aby posielal svoje požiadavky v korektnom formáte.

Závažný problém je v tom, že rozhranie väčšinou opisuje iba formát odosielania a prijímania požiadaviek a nie sémantiku dát. Znamená to, že správa nám povie aká je cena predávaného výrobku, ale už nevieme určiť či je uvedená v dolároch alebo eurách. Ak je cena v eurách a my očakávame doláre, môže to byť pre nás nemilé prekvapenie.

Vo všeobecnosti nie sú načrtnuté problémy s vyhľadávaním, dôveryhodnosťou a sémantikou dát úplne vyriešené. O riešenie problému s vyhľadávaním a dôveryhodnosťou sa snažia webové služby, problém so sémantikou dát rieši zasa kolekcia prístupov, nástrojov a technológií známa pod názvom web so sémantikou. Podrobne sa týmto technológiám venujú ďalšie časti.

3.1.2 Agilnosť

Agilnosť je miera ako rýchlo dokáže spoločnosť adaptovať existujúce aplikácie na podporu nových potrieb obchodu. Ak dokáže spoločnosť sprevádzkovať nové obchodné služby skôr ako konkurencia, môže začať zarábať a konkurencia sa bude ešte len usilovať o jej dobehnutie.

Z technického pohľadu agilnosť silne súvisí s modifikovateľnosťou. Ak je podniková architektúra viazaná voľne, závislosť aplikácií a technológií abstrahovaná od citlivých rozhraní, potom nebude implementácia nových obchodných procesov až tak náročná.

Jednou z prirodzených prekážok agilnosti je heterogénnosť. Architektúra môže byť nádherne navrhnutá, ale ak napríklad bude zrazu nevyhnutné prepojiť novú .NET aplikáciu s existujúcou J2EE aplikáciou využívajúcou JMS, potom môže byť život nepríjemne komplikovaný. V skutočnosti celkový počet kombinácií nekompatibilných technológií v spoločnostiach nie je nič príjemné na zamyslenie sa.

Predsa je len na obzore riešenie. Samozrejme, že heterogénnosť nikdy nezmizne. Ale ani nemusí, ak ju dokážeme ukryť za štandardnú integračnú architektúru. V priebehu posledných rokov sa vynorili XML webové služby ako množina technológií, ktoré sú podporované každým dôležitým hráčom v oblasti. Definujú štandardný protokol a mechanizmus na prepojenie jednotlivých aplikácií v rámci, ale aj medzi spoločnosťami. Počas písania tohto textu sú niektoré časti štandardov ešte stále vo vývoji, ale aj tak je viac ako zrejmé, že ich vo svete IT čaká dlhý život.

Webové služby prinášajú vyššiu mieru agilnosti v štandardnej integrácií. Avšak integrácia nie je jediná prekážka vo zvýšení agilnosti spoločnosti v modifikovaní a nasadení nových aplikácií. Zlepšené vývojárske technológie, ktoré robia zmenu menej nákladnou a náročnou, takisto významne zvyšujú agilnosť spoločnosti. Máme na mysli najmä aspektovo-orientovanú architektúru a modelom riadenú architektúru, ktorým sa venujú ďalšie kapitoly.

3.1.3 Znižovanie nákladov

Veľký rozmach neskorých 90-tych rokov éry „dot.com“ a obrovské míňanie do IT sektora je zrejme minulosťou. V súčasnosti vyžadujú spoločnosti jasné určenie prínosov, ktoré im investície do IT prinesú a akú návratnosť môžu očakávať.

Ak chceme obmedziť míňanie a súčasne naďalej spĺňať naše obchodné ciele, musíme začať pracovať dômyselnejšie. Softvérový priemysel sa snaží hľadať spôsoby ako dosiahnuť sľubovanú zvýšenú efektivitu a nižšie ceny prispôsobením nových vývojových technológií a prístupov. Objektovo a komponentovo orientované technológie boli vymyslené, aby uľahčili návrh a tvorbu znovupoužiteľných komponentov, ktoré môžu byť použité v ďalších aplikáciách. Vytvor niečo raz a používaj v podstate bez nákladov mnohokrát. Túto ideu nemôže nikto odmietnuť a je dostatočne jednoduchá na to, aby jej porozumel aj manažment.

Pravda je taká, že softvérový priemysel zatiaľ viac-menej zlyhal v plnení sľubu znovupoužiteľnosti. Úspešné znovupoužitie bolo dosiahnuté pre široko-použiteľné systémy a to vo väčšej miere v komponentoch infraštruktúry akými sú spojovací softvér a databázy. Avšak pre systémy, ktoré sú využiteľné v menšej miere sa znovupoužiteľnosť nepodarilo uskutočniť. Dôvod je jednoduchý a podrobne opísaný v mnohých prácach v komunite softvérových inžinierov.

V podstate je oveľa nákladnejšie vytvoriť softvérový komponent tak, aby mohol byť použitý v kontexte, pre ktorý nebol pôvodne navrhnutý, lebo treba pridať viac vlastností zabezpečujúcich univerzálne použitie. Je nevyhnutné testovať všetky vlastnosti, zdokumentovať ich a vytvoriť príklady použitia ako komponent používať. Štúdie poukazujú, že náklady na vyprodukovanie kvalitného znovupoužiteľného komponentu sú 3 až 10 krát vyššie.

Samozrejme, že všetky tieto investície môžu byť veľmi hodnotné ak sa komponenty skutočne využívajú znovu a znovu. Ale ak nie? Potom sme jednoducho bez zmyslu investovali veľa času a úsilia do vývoja znovupoužiteľného komponentu.

Našťastie, sa niekoľko múdrych architektov pred pár rokmi nad týmto problémom zamyslelo. Uvedomili si, že úspešné znovupoužitie sa neudeje čakaním na zázrak, ale môže byť dosiahnuté ak sa správne pochopí a naplánuje produktová stratégia. A tak vznikol prístup s názvom „rada softvérových produktov“. Tomuto prístupu sa podrobne venuje ďalšia kapitola. Prezentuje množinu overených postupov, ktoré môžu byť adaptované a prispôsobené v spoločnosti na rozumné investovanie do softvérovej architektúry a komponentov.

3.1.4 Zhrnutie

Čo nás teda čaká v najbližších rokoch? Veľmi pravdepodobne bude pokračovať nárast zložitosti obchodných procesov a následne ich podporujúcich softvérových systémov. Dôraz sa bude klásť na automatickú kompozíciu procesov, čo si vyžiada rozšírenie schopností súčasne využívanej orchestrácie procesov a vytvorenie spoľahlivých mechanizmov na definovanie prostredia, rozhraní, vyhľadávanie obchodných partnerov a vyjadrenie sémantiky dát. Načrtnuté problémy sa snažia riešiť technológie ako webové služby a web so sémantikou.

Ďalším dôležitým aspektom, ktorý ovplyvní dizajn a vývoj novej generácie softvérových aplikácií je agilnosť. Prirodzenou prekážkou agilnosti je heterogénnosť. Architektúra môže byť nádherne navrhnutá, ale ak bude nevyhnutné prepojiť aplikácie dvoch rôznych nekompatibilných technológií, potom môže byť život nepríjemne komplikovaný. Riešením je ukryť heterogénnosť aplikácií za štandardnú integračnú architektúru. V priebehu posledných rokov sa ako riešenie tohto problému vynorili XML webové služby, ktoré sú podporované každým dôležitým hráčom v oblasti.

Doba bezhlavého míňania do IT sektora je už minulosťou – spoločnosti redukujú náklady na softvér a súčasne chcú zvyšovať prínosy zo svojej softvérovej infraštruktúry. Ak chceme obmedziť míňanie a súčasne naďalej spĺňať obchodné ciele je nutné začať pracovať efektívnejšie, napr. využívať znovupoužiteľnosť. Už v súčasnosti bolo znovupoužitie s úspechom využití pre široko-použiteľné systémy a to najmä v komponentoch infraštruktúry akými sú *spojovací softvér* a databázy. Pre špecializované systémy sa znovupoužiteľnosť vo väčšej miere nedarí využívať pre jej počiatočnú náročnosť vývoja.

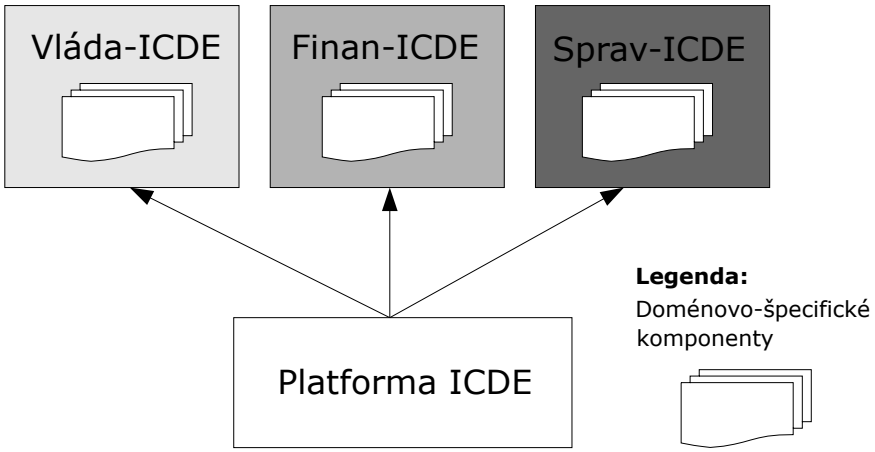
3.2 Rady softvérových produktov

3.2.1 Rady produktov ICDE

Systém ICDE, ktorý používame ako prípadovú štúdiu, je platforma pre zber a šírenie informácií, využiteľná v rôznych aplikačných doménach. Táto vlastnosť je silnou aj

slabou stránkou systému súčasne. Slabosť tkvie v skutočnosti, že je nevyhnutné pred nasadením do prevádzky prispôsobiť systém danej aplikačnej doméne a budúcim používateľom, čo stojí čas a peniaze a môže odradiť potenciálneho zákazníka od kúpy produktu.

Uvedomenie si tejto skutočnosti viedlo vývojový tím k rozhodnutiu vytvoriť rozšírené verzie ICDE platformy pre tri najdôležitejšie aplikačné domény, do ktorých sa systém najviac nasadzoval. Ide o domény finančná analýza, spravodajská činnosť a prieskum vládnej politiky. Každá z týchto rozšírených platforiem predstavuje odlišný produkt obsahujúci špecifické komponenty, ktoré robia základnú ICDE platformu viac používateľsky prívetivou pre danú aplikačnú doménu.



Obrázok 3-1. Základná architektúra doménovo-špecifických produktov ICDE platformy.

Pre dosiahnutie daného cieľa využil vývojový tím techniku brainstormingu, aby analyzoval niekoľko rôznych stratégií vývoja, ktoré by mohli minimalizovať návrh a vynaložené úsilie tvorby troch odlišných produktov. Základnou myšlienkou bolo použiť nezmenenú základnú platformu ICDE pre každý z troch produktov a nad ňou vytvoriť dodatočné doménovo-špecifické komponenty, pričom výsledné produkty by vznikali poskladaním základnej platformy a doménovo-špecifických komponentov. Základnú myšlienku architektúry znázorňuje obrázok 3-1.

Doterajšie úvahy vývojového tímu predstavujú prvý krok adaptácie spôsobu vývoja pomocou radu softvérových produktov pre ICDE technológiu. Rad produktov predstavuje stratégiu štrukturovania a manažovania súbežného vývoja kolekcie súvisiacich produktov vysoko efektívnym spôsobom. Významné zníženie nákladov a vynaloženého úsilia dosahuje táto stratégia aplikovaním širokej škály znovu-použiteľnosti architektúry, komponentov, testovacích scenárov a dokumentácie. Pre porovnanie by si oddelený vývoj jednotlivých rozšírených verzií ICDE platformy vyžiadal minimálne trojnásobné prostriedky.

3.2.2 Rady softvérových produktov

Znovu použiť softvér je jednoduché ak vieme, že robí presne to čo potrebujeme. Avšak softvér, ktorý nám poskytuje „skoro“ to čo potrebujeme, je pre nás zvyčajne úplne

nepoužiteľný. Preto je pre dosiahnutie všetkých výhod znovupoužiteľnosti softvéru potrebné vedieť efektívne narábať s variabilitou softvéru. Moderný prístup akým rady softvérových produktov sú, podporuje variabilitu softvéru vo veľkom a z praxe dokazuje, že predstavuje efektívny spôsob ako profitovať zo znovupoužiteľnosti a variability softvéru. Zavedenie radu softvérových produktov prinieslo mnohým spoločnostiam zníženie vývojových nákladov, zníženie času vývoja a zvýšenie kvality výsledného produktu.

Hlavná myšlienka spočíva v tom, že súbor súvisiacich produktov je vyvíjaný kombináciou základných a produktovo-špecifických aktív. Základné aktíva implementujú základnú funkcionalitu, ktorá je konštantná pre celú škálu produktov v rade a poskytujú podporu variabilných vlastností, ktoré môžu byť určené pre individuálne produkty. Variačné body základných aktív tvoria rozhranie, ktoré poskytuje prístup k jednotlivým vlastnostiam pre produktovo-špecifické aktíva, ktoré implementujú jedinečnú funkcionalitu produktu. Základné aktíva si môžeme predstaviť ako jadro každej aplikácie a preto budeme v nasledujúcej časti textu označovať základné aktíva aj ako jadro.

Vývoj radu softvérových produktov nezasahuje iba do architektúry, dizajnu a programovania, ale aj do existujúcich procesov životného cyklu produktu. Vyžaduje od nich nové schopnosti pre manažment znovupoužívaných základných aktív a produktov, ktorým sa budeme venovať v nasledujúcich častiach.

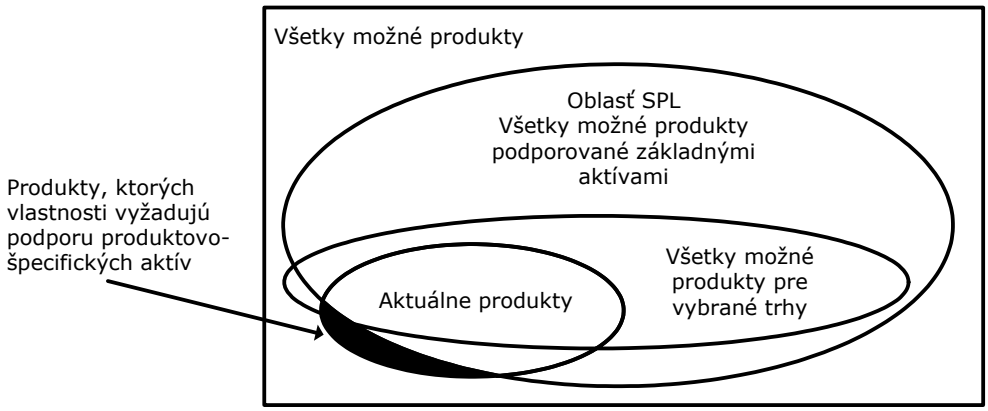
3.2.3 Prínosy

Pre organizácie vyvíjajúce produkty zdieľajúce rovnaké základy je prístup pomocou radu softvérových produktov (S*oftware Product Lines, SPL*) veľmi prospešný. Dokáže adresovať širokú oblasť trhu, pričom každý produkt z radu je zameraný na špecifický segment. Niektoré organizácie využívajú stratégiu SPL pre vývoj a údržbu rôznych variant jedného produktu pre každého individuálneho zákazníka.

Oblasť SPL predstavuje rozsah možných variácií výrobku podporovaných základnými aktívami (jadrom). Aktuálne produkty vyrábané a udržiavané spoločnosťou opisuje množina aktuálnych produktov, ktorá v SPL spadá vo väčšine prípadov do oblasti možných variácií podporovaných základnými aktívami (oblasť SPL). Avšak produktovo-špecifické aktíva poskytujú možnosť vývoja funkcionality za rámec oblasti SPL a preto môže množina aktuálnych produktov presahovať oblasť SPL. Pre dosiahnutie maximálneho zisku z SPL vývoja je potrebné aby sa oblasť SPL čo najpresnejšie zhodovala s oblasťou cieľových trhov a množinou aktuálnych produktov vyvíjaných danou spoločnosťou. Všetky tri typy množín produktov znázorňuje Vennov diagram na obrázku 3-2.

Najzrejmejší prospech z SPL vývoja je zvýšenie produktivity. Náklady na vývoj a údržbu základných aktív nevznikajú pri každom novom produkte, ale sú rozdelené do nákladov pre každý jeden produkt. Čím väčší počet produktov bude spoločnosť vyrábať, tým významnejšie ekonomické zhodnotenie môže dosiahnuť.

SPL vývoj má ale aj ďalšie významné prínosy. Ak sú dobre zavedené základné aktíva v SPL, potom je čas potrebný pre vytvorenie nového produktu podstatne nižší ako pri tradičnom spôsobe vývoja. Namiesto toho aby zákazníci čakali na vývoj celého produktu, čakajú len na vývoj špecifickej funkcionality podľa ich potrieb.



Obrázok 3-2. Oblasť radu softvérových produktov.

Prínosy z SPL vývoja sa prejavujú aj v podobe zvýšenej kvality produktov. Pri tradičnom spôsobe vývoja sa môže jedna chyba opakovať v niekoľkých produktoch, zatiaľ čo pri SPL vývoji stačí chybu v základných aktívach opraviť len raz, pričom z odhalenia a opravenia chyby jedného produktu bude profitovať každý produkt v SPL.

SPL vývoj má aj sekundárne prínosy – manažment základných a produktovo-špecifických aktív poskytuje jasný a jednoduchú pohľad na rozsah produktov udržiavaných spoločnosťou a umožňuje:

- Zjednodušenie aktualizácie produktov na novú verziu jadra.
- Určenie dôležitej funkcionality pre obchod, ktorá je daná vlastnosťami jadra.
- Jednotný pohľad na odlišnosti jednotlivých produktov.

3.2.4 Adaptácia

Adaptácia SPL vývoja predstavuje radikálny zásah do chodu spoločnosti, ktorý je zvyčajne vyvolaný krízou. Môže ísť o naliehavú požiadavku na rýchly vývoj viacerých nových produktov alebo o znižovanie nákladov na vývoj. V nasledujúcej časti textu poukážeme na niekoľko postupov a procesov dôležitých pre adaptáciu SPL vývoja.

Štartovacie body pre adaptáciu SPL vývoja

Existujú dva rôzne počiatočné stavy, v ktorých sa môže spoločnosť ocitnúť v procese adaptácie SPL vývoja:

1. *Štart na zelenej lúke* – na počiatku neexistuje žiadny produkt.
2. *Štart na pooranom poli* – existuje množina produktov, ktoré boli vyvinuté bez ohľadu na znovupoužiteľnosť.

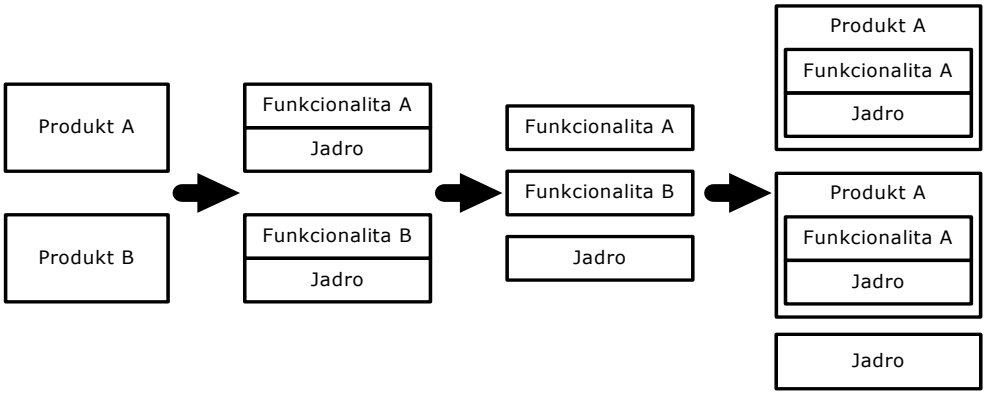
Štart na zelenej lúke

Vychádzajúc z Vennovho diagramu na obrázku 3-2 je pre organizáciu prvoradé určenie cieľových trhov, ktoré definujú množinu možných produktov. Ďalším krokom je odvodenie dlhodobého zamerania oblasti SPL. Nakoniec je potrebné zo vzniknutých množín určiť možný krátko až strednodobý plán produkcie výrobkov. Z dlhodobého hľadiska je veľmi dôležité určiť dostatočne široké zameranie oblasti SPL, čo je

pre začínajúcu spoločnosť, ktorá nie je danej oblasti expertom veľmi náročné. Ak sa zvolí príliš úzka oblasť zamerania SPL, podstatne sa znížia prínosy SPL vývoja. A naopak ak sa zvolí príliš široká oblasť zamerania SPL, vzrastá riziko tvorby komponentov, ktoré sa v budúcnosti vôbec nevyužijú, čo predraží a predĺži vývoj.

Štart na pooranom poli

Štart na pooranom poli je situácia kedy organizácia už má existujúce produkty, ktoré sa snaží pretransformovať na SPL. Podobne ako pri štarte na zelenej lúke aj tu treba určiť množinu cieľových trhov a oblasť zamerania SPL. Avšak, organizácia v tejto pozícií už má dostatok vedomostí o trhoch a potrebnej funkcionalite produktov. Oblasť zamerania SPL zväčša vychádza z funkcionality existujúcich a plánu budúcich produktov.



Obrázok 3-3. Dolovanie základných aktív z množiny existujúcich produktov.

Po zadefinovaní potrebných množín a zameraní je potrebné určiť čo je spoločné jadro a čo je špecifické pre každý existujúci produkt. Namiesto zahodenia všetkých existujúcich aktív produktov a štart z čistého stola, je možné použiť extrakčný prístup dolovania aktív z existujúcich produktov. Extrakcia základných aktív sa v mnohom ponáša na cvičenie refaktoringu. Kompletný proces dolovania základných aktív znázorňuje obrázok 3-3. Začína z pôvodnej množiny produktov a jeho cieľom je skončiť s identickými produktmi s jediným rozdielom, že budú všetky postavené na základnej množine aktív (jadre).

Adaptácia SPL pre ICDE

Vývojový tím ICDE platformy bol smerovaný do SPL vývoja predstavou vývoja troch produktov súčasne – bolo treba vytvoriť 3 nové produkty pre 3 špecifické trhy s využitím existujúceho produktu ako štartovacieho mostíka. Adaptácia SPL vývoja pre ICDE preto predstavuje scenár štartu na pooranej lúke, kde z existujúcej základne zdrojového kódu treba vydolovať znovupoužiteľné komponenty.

3.2.5 Pokračujúci vývoj radu softvérových produktov

Stratégia radu softvérových produktov musí byť efektívna nielen pre počiatočný vývoj nových produktov, ale aj pre pokračujúcu údržbu a rozširovanie. SPL vývoj má mnoho prínosov, ale je zložitejší ako normálny vývoj produktu. Preto je podpora rozšírených procesov výroby pre dosahovanie efektívnosti pokračujúceho SPL vývoja nevyhnutnosťou. Veľký dôraz treba klásť najmä na riadenie zmien a plánovanie nových vlastností produktov.

Riadenie zmien

Softvérové riadenie zmien sa dotýka plánovania, koordinácie, sledovania a manažovania dopadov zmien softvérových artefaktov (napr. zdrojového kódu). V každom type vývoja produktu, je produkt ovplyvňovaný niekoľkými účastníkmi, pričom každý účastník môže mať rôzne požiadavky na novú funkcionalitu, ba dokonca aj na nefunkcionálne vlastnosti produktu. Pod účastníkmi si môžeme predstaviť všetkých, ktorí môžu ovplyvniť vývoj produktu napr. jednotlivých zákazníkov, manažment alebo vývojárov. Ak vyvíjame jednotlivé produkty oddelene, sú oddeľujú aj požiadavky jednotlivých účastníkov. Pri SPL vývoji od základných aktív (jadra) závisí každý jeden produkt. Je preto nevyhnutné zohľadniť požiadavky všetky účastníkov. Riadenie zmien sa stáva pre SPL vývoj kľúčové a náročnejšie ako pri normálnom oddelenom vývoji.

Existuje množstvo stratégií ako riešiť riadenie zmien pre SPL vývoj, ale žiadna z nich neposkytuje dokonalé riešenie. Vyžaduje sa kombinovanie dlhodobého, strednodobého a krátkodobého manažmentu zmien.

Spôsob ako dlhodobo obmedziť konflikt riadenia zmien je vyvinúť vysoko kvalitné základné aktíva (jadro). Riadenie zmien je jednoduché ak zmeny nie sú potrebné – kvalitné základné aktíva môžu výrazne zredukovať ich potrebu. Je veľmi ťažké a bez skúseností v danej oblasti až nemožné, vytvoriť vysoko kvalitné aktíva. Na druhej strane sa kvalita základných aktív neustále počas SPL vývoja zvyšuje, pretože ich jednotlivé produkty využívajú rôznymi spôsobmi, čo vedie k ich dôkladnému testovaniu a odhaleniu vyššieho počtu chýb.

Vývojový plán *(roadmap)* je strednodobý spôsob manažovania zmien. Je to formálny plán zmien základných aktív, ktorý je užitočný na manažovanie očakávaní a plánov účastníkov produktu. Predchádza mylným očakávaniam účastníkov o časovom pláne uvoľňovania nových verzií a funkcionality.

Krátkodobé zmeny jadra môžu narúšať jeho konzistenciu a zvyšujú tak úsilie vynaložené na jeho údržbu a dlhodobejšie plánovanie. Preto sa odporúča ak je zmenu nevyhnutné uskutočniť, uskutočniť ju v produktovo-špecifickej časti. Ak sa zmena osvedčila a rátame s ňou aj v ďalších verziách, je nutné naplánovať presunutie zmeny do jadra vo vývojovom pláne, aby nedošlo k narušeniu dlhodobého plánu.

Plánovanie nových vlastností

V SPL vývoji prebieha neustáli vývoj produktovo-špecifických a základných aktív. Každý zásah do jadra môže ovplyvniť funkcionalitu všetkých produktov. Ako teda pridávať novú vlastnosť do radu softvérových výrobkov?

Poznáme tri spôsoby zavedenia zmien do jadra:

- *Proaktívne* – plánovanie nových vlastností a ich implementácia do jadra bez toho aby ich aspoň jeden produkt vyžadoval. Ide o dlhodobú investíciu, ktorá redukuje neskoršie konflikty jadra a skracuje čas pridania novej vlastnosti do prvého produktu. Vyžaduje však bohaté skúsenosti v danej oblasti.
- *Reaktívne* – implementácia novej vlastnosti priamo do jadra po vzniknutí prvej požiadavky.
- *Retroaktívne* – ak nastane požiadavka na implementáciu novej vlastnosti, implementuje sa do produktovo-špecifickej časti. Implementácia vlastnosti do jadra sa vykoná až keď bude danú vlastnosť vyžadovať viacero produktov. Redukujú sa tak konflikty zmien jadra a počet nevyužívaných vlastností.

Vo vývoji je možné využívať ktorýkoľvek prístup, pričom má každá zo stratégií rôznu réžiu, prínosy a riziká. Výber je ovplyvnený zámermi spoločnosti. Najčastejšie sa využíva kombinácia jednotlivých prístupov. Tabuľka 3-1 sumarizuje niektoré rozdiely jednotlivých prístupov.

Tabuľka 3-1. Porovnanie stratégií plánovania nových vlastností.

	Proaktívne	Reaktívne	Retroaktívne
Dlhodobá investícia	Áno	Nie	Nie
Obmedzenie rizika konfliktu zmien základných aktív (jadra)	Áno	Nie	Áno
Skrátenie času pridania vlastnosti do prvého produktu	Áno	Nie	Nie
Obmedzenie rizika nepotrebnej vlastnosti jadra	Nie	Áno	Áno

Rad výrobkov ICDE

Nábeh na SPL vývoj nebol pre ICDE tím len otázkou architektúry. Každý produkt mal skupinu ľudí riadenú zákazníkom, ktorá bola zainteresovaná do definovania požiadaviek pre nové produkty. Keďže každý produkt vychádzal z toho istého jadra bolo potrebné jednotlivé požiadavky skombinovať, aby nevznikali kolízie a ani naopak veľa požiadaviek nebolo rovnakých a nevznikali duplicity. Preto bolo nevyhnutné vytvoriť rozhranie pre zber požiadaviek. Spoločné rozhranie pre správu požiadaviek od jednotlivých zákazníkov nebolo možné realizovať, pretože tím nechcel aby napríklad zákazník z finančného sektora vedel detaily o spravodajskom systéme a naopak. Preto sa ICDE tím rozhodol zriadiť oddelené systémy pre zber požiadaviek zákazníkov. Tieto systémy boli priamo prepojené na interný systém pre zber požiadaviek, ktorý spravoval všetky požiadavky od vývojárov až po zákazníkov. Takto boli všetky požiadavky prístupné na jednom mieste, čím sa vytvoril efektívny nástroj na riešenie duplicít a konfliktov.

Stratégiou ICDE tímu bolo vydať z 3 pripravovaných produktov ten najjednoduchší ako prvý. Bol ním produkt pre vládny sektor, na ktorom krátko po vydaní našiel zákazník niekoľko chýb. ICDE tím dokázal reagovať veľmi pružne a nájdené chyby rýchlo odstránil. Dobrou správou bolo, že jedna z nájdených chýb bola chybou jadra, konkrétne v komponente *Data Collection*, ktorú využívajú všetky tri aplikácie a na ktorú by narazili aj ostatní zákazníci, čo bol jeden zo skorých prínosov vývoja SPL.

Zlá správa prišla ihneď po vydaní zvyšných produktov. Zákazník z finančného sektora objavil defekt produktu, ktorý bol spôsobený chybou v komponente *Data Analysis*. Chyba bola riešiteľná jednoduchým zásahom do jadra. Avšak zákazník z prostredia vlády mal dokončené akceptačné testy a produkt už využíval v praxi. Zmena jadra by znamenala aj zmenu aplikácie pre vládny sektor, čím by bol zákazník nútený ešte raz vykonať všetky akceptačné testy, čo by stálo čas a peniaze. Pritom nájdená chyba ovplyvňovala iba funkcionalitu produktu pre finančný sektor.

ICDE tím mal veľký záujem odstrániť chybu modifikovaním jadra a zároveň chcel uspokojiť každého zákazníka. Rozmýšľali o obídení modifikácie jadra modifikáciou

produktovo-špecifickej časti. Nakoniec sa však rozhodli, že chybu odstránia zmenou v jadre a aplikáciu zákazníka z vládneho prostredia ponechajú so starou verziou komponentu *Data Analysis*.

3.2.6 Zhrnutie

Stratégia vývoja rad softvérových výrobkov je efektívny spôsob ako profitovať zo znovupoužiteľnosti a variability softvéru. V súčasnosti využíva vývoj radu softvérových výrobkov mnoho spoločností ako nástroj pre zvýšenie produktivity, skrátenia času dodania softvéru na trh a zvýšenia kvality produktu. Nič však nie je zadarmo. Vývoj SPL je podstatne komplikovanejší ako bežný vývoj produktu a preto ak z neho chceme vyťažiť všetky spomínané výhody je nevyhnutné aplikovať sofistikovanejšie procesy podpory tvorby softvéru, najmä o riadenie zmien a plánovanie nových vlastností.

3.3 Aspektovo-orientovaný vývoj softvéru

Aspektovo-orientovaný vývoj softvéru predstavuje jednu z novších paradigiem, ktorá nadväzuje na iné prístupy, najmä na objektovo-orientované myslenie pri vývoji softvéru. Aj keď prvé podnety pre uvažovanie aspektovým spôsobom prišli už pred niekoľkými desiatkami rokov, aspektovo-orientované programovanie (AOP) vzniklo až v deväťdesiatych rokoch minulého storočia. Podnetom pre rozvíjanie aspektového prístupu bola najmä rastúca zložitosť softvéru z pohľadu vnútornej zviazanosti a súdržnosti. Kľúčový pojem, na ktorom je aspektovo-orientovaný prístup postavený, je výraz „pretínajúce záležitosti“.

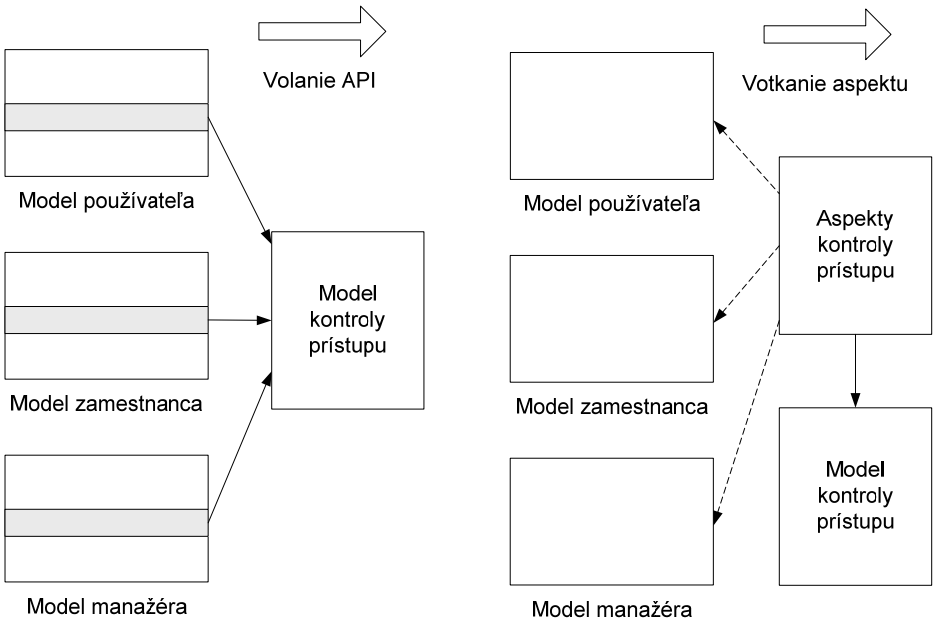
3.3.1 Pretínajúce záležitosti

Pojem pretínajúca záležitosť môžeme charakterizovať takto:

- špecifická požiadavka, ktorú je potrebné brať do úvahy v snahe naplniť ciele, ktoré má systém spĺňať,
- treba brať na ňu ohľad na viacerých miestach v systéme,
- rozoznávame funkcionálne záležitosti (doménovo závislé) a nefunkcionálne záležitosti (výkonnosť systému, bezpečnosť, kvalita),
- príklady pretínajúcich záležitostí sú kontrola transakcií, autorizácia, spracovanie chýb a výnimiek, ladenie softvéru.

Príklad implementácie softvérového systému dvoma rôznymi spôsobmi, čisto objektovým a aspektovým je na obrázku 3-4. Vďaka AOP sú pretínajúce časti kódu z modelu používateľa, zamestnanca a manažéra odstránené a ich funkcionalita je zoskupená do jedného modulu s názvom *aspekty kontroly prístupu*.

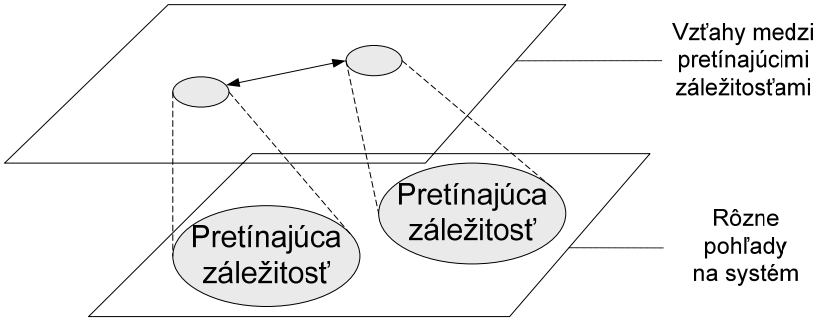
V aspektovo-orientovanom prístupe je možné pretínajúce záležitosti zoskupovať a zapuzdrovať v moduloch, ktoré sa nazývajú aspekty. Existujú dva spôsoby tvorby softvéru z pohľadu pretínajúcich záležitostí, ktoré sa označujú ako symetrický a asymetrický prístup.



Obrázok 3-4. Príklad implementácie kontroly prístupu ako pretínajúcej záležitosti.

Symetrický prístup v AOP

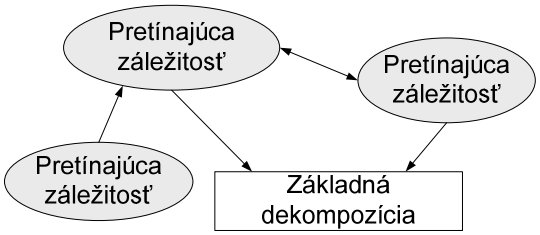
Pri symetrickom prístupe záležitosti vyjadrujú rôzne pohľady na softvérový systém, ktorý vzniká ich kompozíciou (Obrázok 3-5). Jednotlivé záležitosti sú si kvalitatívne rovnocenné.



Obrázok 3-5. Pretínajúce záležitosti pri symetrickom prístupe.

Asymetrický prístup v AOP

Pri asymetrickom prístupe sa softvérový systém skladá z dvoch odlišných častí. Jadro tvorí základná dekompozícia (napríklad kód v jazyku Java), ktorej funkcionalitu obohacujú a menia naviazané pretínajúce záležitosti, ktoré sa tiež môžu ovplyvňovať medzi sebou (Obrázok 3-6).



Obrázok 3-6. Pretínajúce záležitosti pri symetrickom prístupe.

Tento prístup reprezentuje najznámejší aspektový programovací jazyk, AspectJ (Kiczales, 2001), ktorý pre základnú dekompozíciu používa jazyk Java. Na tento vzťah však možno nazerať aj opačne, teda že program v jazyku základnej dekompozície, ktorý je samostatne preložiteľný a spustiteľný, je obohatený pomocou aspektového prístupu.

3.3.2 Základné pojmy aspektovo-orientovaného prístupu

V tejto časti sú opísané základné prvky jazykov AOP zabezpečujúce realizáciu pretínajúcich záležitostí.

Pretínanie

Technika umožňujúca manipuláciu so záležitosťami, ich zoskupovanie do modulov (aspektov) a ich zavádzanie do určených miest v kóde, sa nazýva pretínanie (*crosscutting*). Rozlišujeme dva typy pretínania:

- statické – mení statickú štruktúru komponentov (napr. vkladanie nových metód, atribútov),
- dynamické – votkávaním kódu ovplyvňuje vykonávanie a správanie sa objektov (napr. či sa po metóde A zavolá metóda B).

Body spájania

Dobre definované miesto v kóde aplikácie sa nazýva bod spájania (*join point*). Príkladom bodu spájania je volanie metódy či nastavenie premennej, body spájania nemôžu zahŕňať napr. cyklus.

Bodový prierez

Body spájania, v ktorých sa majú aplikovať pretínajúce záležitosti sa nazývajú bodové prierezy (*pointcuts*). Bodové prierezy umožňujú používanie zovšeobecňujúcich konštrukcií, nasledovný príklad bodového prierezu v jazyku AspectJ zachytáva ako body spájania všetky metódy triedy `ExampleClass`, ktorých názov začína výrazom `get` (Príklad 3-1).

```
* ExampleClass.get*(..)
```

Príklad 3-1. Bodový prierez v jazyku AspectJ.

Videnie

Vnútornú logiku pretínajúcej záležitosti vyjadrujú videnia (*advice*). Kód videnia je vykonaný pri výskyte zadaných bodových prierezov.

Aspekt

Aspekt predstavuje v AOP podobnú abstrakciu ako pojem trieda v objektovom prístupe. Aspekt zapuzdruje bodové prierezy a asociované videnia.

Votkávanie

Proces aplikovania aspektov v správnych miestach zdrojového kódu sa nazýva votkávanie. Cieľom je zaručiť, aby sa vykonateľné videnia votkali v správnych bodoch spájania. Existujú rôzne prístupy k realizácii votkávania, napr. v jazyku AspectJ sa väčšina votkávania deje počas kompilácie. Iné prístupy môžu využívať vyhradený pre- alebo post-procesor, prípadne môže votkávanie zabezpečovať virtuálny stroj.

3.3.3 Aspektová paradigma a životný cyklus softvéru

Aspektovo orientované uvažovanie nie je obmedzené len na fázu implementácie, identifikovať pretínajúce záležitosti možno už pri špecifikácii požiadaviek. Pri fázach analýzy a návrhu sa uprednostňuje symetrické uvažovanie, prechod na asymetrickú implementáciu však nepredstavuje vážnejší problém.

Aspektovo-orientovaná analýza softvéru – Theme/Doc

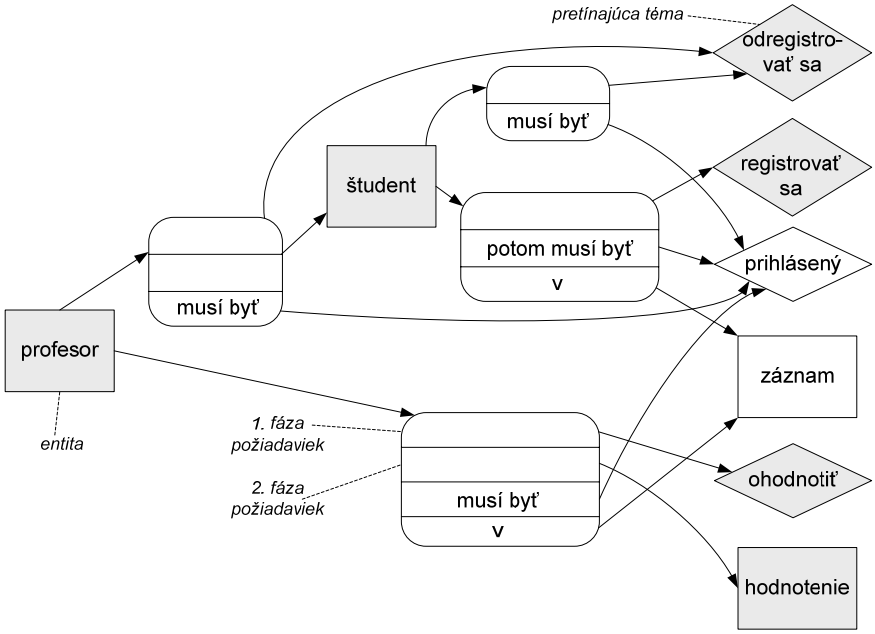
Prístup Theme/Doc (Baniassad, 2004) umožňuje identifikovať pretínajúce záležitosti na základe požiadaviek na vytváraný systém. Ústredný je pojem *téma*, ktorý predstavuje určitý pohľad na systém. Prístup Theme/Doc sa skladá z nasledovných krokov:

1. Identifikácia akcií a entít, predstavujúcich správanie systému. Akcie sa rozdelia podľa významu a dôležitosti na hlavné a vedľajšie. Hlavné akcie sa stanú témami, podružné akcie tvoria metódy v rámci tém.
2. Ďalej sa identifikujú vzťahy medzi témami, podstatné sú najmä požiadavky, ktoré zdieľa viacero tém. Príklad je na obrázku 3-7.
3. Po identifikovaní základným a pretínajúcich tém je možné tento model transformovať do Theme/UML.

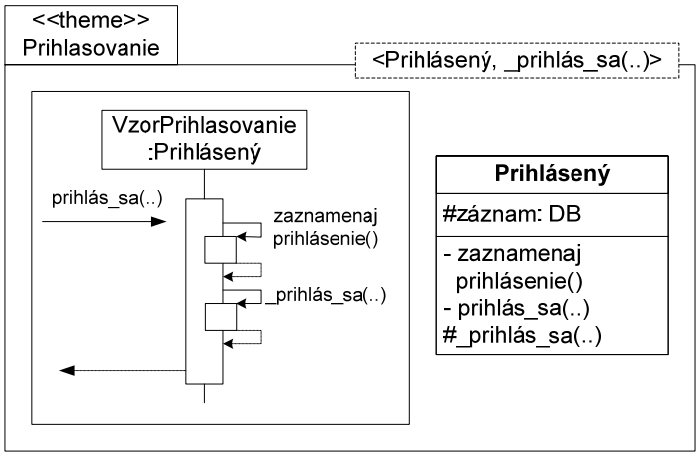
Aspektovo-orientovaný návrh softvéru – Theme/UML

Prístup Theme/UML predstavuje rozšírenie jazyka UML a umožňuje modelovanie rôznych pohľadov na systém. Pretínanie tém je vyjadrené v UML použitím šablón balíkov. Model pre jednu tému je na obrázku 3-8.

Pre skoršie fázy vývoja softvéru je určená technika modelovania bodov spájania – JPDD (*Join Point Designation Diagrams*) (Stein, 2004). Taktiež existuje rozšírenie modelovania prípadov použitia pre aspektovú paradigmu (Jacobson, 2004).



Obrázok 3-7. Pohľad na témy a vzťahy medzi nimi, príklad systému pre správu predmetov v škole.



Obrázok 3-8. Model pre tému „prihlásený“ v Theme/UML.

3.3.4 Rámec AspectWerkz

AspectWerkz[2] je rámec obohacujúci aplikácie vyvinuté v Jave o aspektovú paradigmu. Aspekty a videnia je možné písať tiež priamo v Jave bez znalosti iného programovacieho jazyka. Zároveň je však možné spúšťať aj aspekty vyvinuté v iných prostrediach (napr. Spring AOP, AOP Alliance a čiastočne aj AspectJ). Zavádzanie aspektov a deklarácia bodových prierezov sa realizuje pomocou XML. Rámec AspectWerkz tiež umožňuje používanie anotácií.

[2] Rámec AspectWerkz, http://aspectwerkz.codehaus.org

Príklad aspektu

Jednoduchý príklad[3] ukazuje triedu `HelloWorld` (Príklad 3-2), ktorá vypisuje známy výraz „Hello World“. Aspekt, naviazaný na túto triedu vypíše výraz „before greeting...” vždy pred výpisom „Hello World“. Z kódu v uvedenom príklade je kvôli zjednodušenie vyňatá deklarácia balíkov tried.

```
public class HelloWorld {

public static void main(String args[]) {
  HelloWorld world = new HelloWorld();
  world.greet();
}

public String greet() {
  System.out.println("Hello World!");
}
```

Príklad 3-2. Jednoduchá trieda v jazyku Java, na ktorú sa naviaže aspekt.

Aspekt, taktiež napísaný v jazyku Java ukazuje príklad 3-3. Telo metódy `beforeGreeting` slúži ako videnie, argumentom metódy je bodový prierez, ktorý je definovaný v XML súbore.

```
public class MyAspect {

  public void beforeGreeting(JoinPoint joinPoint) {
    System.out.println("before greeting...");
  }
}
```

Príklad 3-3. Aspekt obsahujúci jedno videnie v jazyku Java.

Votkanie aspektu do triedy `HelloWorld` zabezpečí definícia v XML súbore (pPríklad 3-4), ktorá určuje, že videnie v metóde `MyAspect.beforeGreeting` sa vykoná pred vykonaním metódy `HelloWord.greet`.

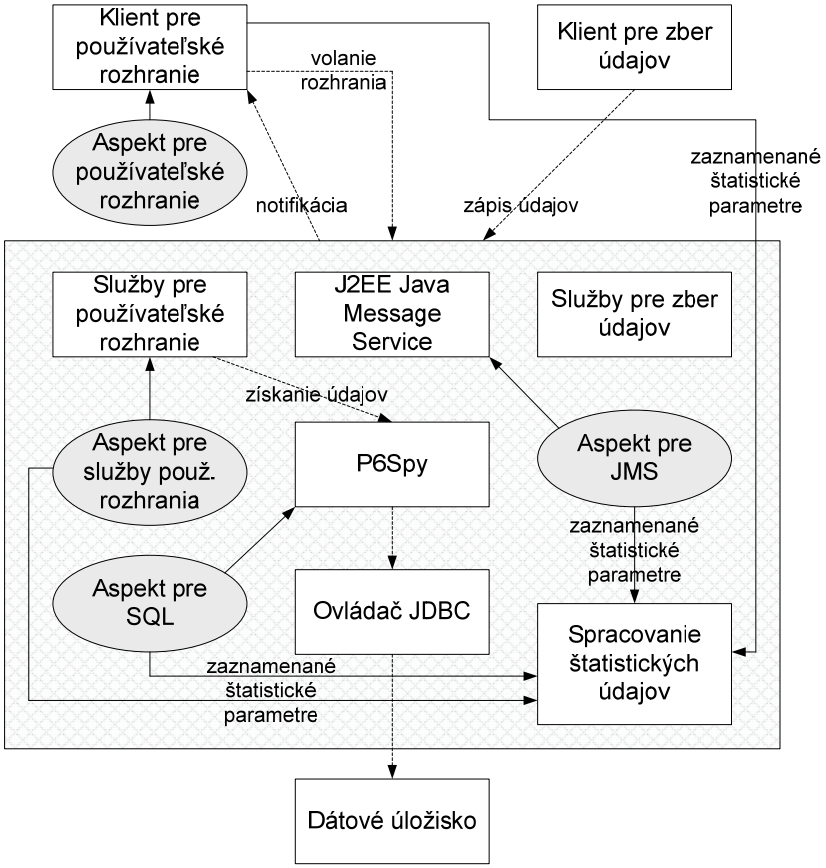
```
<aspectwerkz>
  <system id="AspectWerkzExample">
    <aspect class="MyAspect">

      <pointcut name="greetMethod"
         expression="execution(* HelloWorld.greet(..))"/>

      <advice name="beforeGreeting"
         type="before" bind-to="greetMethod"/>
    </aspect>
  </system>
</aspectwerkz>
```

Príklad 3-4. XML súbor zabezpečujúci votkanie.

[3] Príklad vychádza z návodu na stránke http://docs.codehaus.org/display/AW/Hello+World

Zavedenie aspektov do systému ICDE

Pomocou rámca AspektWerkz je zavedené monitorovanie parametrov do systému ICDE bez nutnosti rekompilácie celého kódu aplikácie (Obrázok 3-9). Sledovanie využívania a prípadného zahltenia jednotlivých modulov systému zabezpečujú aspekty prispôsobené konkrétnym modulom. Sledovanie dopytov do databázy zabezpečuje nástroj P6Spy[4].



Obrázok 3-9. Architektúra systému ICDE po zavedení aspektov pre monitorovanie systému pomocou AspectWerkz.

3.3.5 Zhrnutie

Aspektovo-orientovaná paradigma je podporovaná v mnohých vývojových prostrediach, keďže umožňuje redukovať zložitosť softvéru. Medzi štandardné oblasti v ktorých použitie AOP prináša výhody oproti iným prístupom je napr. logovanie, kontrola transakcií alebo autorizácia. Aspektovo-orientovaný prístup je však možné využiť aj pri mnohých iných úlohách, v práci (Bruschi, 2006) je analyzovaný prístup, ako pomocou AspectJ prekonať a modifikovať skompilovaný kód zabezpečený proti dekompilácii (*obfuscated code*). Pomocou AOP tiež možno nazerať z inej perspektívy na návrhové vzory a refaktorizáciu kódu (Hannenberg, 2003).

[4] P6Spy, http://www.p6spy.com/

Medzi nevýhody AOP patrí v súčasnosti nedostatočná vizualizácia modelovania pretínajúcich záležitostí. Použitie AOP pri vývoji softvérových systémov prináša tiež otázku ako určiť, ktorý konkrétny človek je zodpovedný za ktorý segment zdrojového kódu, keďže pomocou aspektov je možné výrazne ovplyvňovať inú časť kódu.

3.4 Modelom riadená architektúra

3.4.1 Čo je MDA?

Jednou z tém, ktorá sa neustále opakuje vo vývoji softvérového inžinierstva je využitie abstraktnejších formálnych jazykov pri modelovaní systémov. Pri vývoji softvéru sú abstraktné opisy, zapísané napríklad v jazyku Java alebo C# za pomoci rôznych nástrojov transformované do spustiteľnej podoby. Vývoj s využitím týchto abstraktných notácií prispieva k vyššej produktivite a znižuje chybovosť, pretože samotný preklad medzi abstraktnými a spustiteľnými podobami je realizovaný automatickými prekladačmi.

Je zrejmé, že vrcholom abstrakcie programovacích jazykov nie je ani Java, ani C# a dokonca ani žiadna z ich moderných alternatív. V histórii výskumu programovacích jazykov je však možné nájsť veľké množstvo návrhov nových vývojových jazykov, od úplne všeobecných až po úzko špecializované na najrôznejšie aplikačné domény. Väčšina z nich sa však v praxi nikdy nevyužije.

Modelom riadená architektúra (*Model Driven Architecture, MDA*) je novým prístupom, ktorý prináša ešte abstraktnejšiu špecifikáciu a zároveň aj vývojové nástroje určené pre IT trh. OMG[5] definuje MDA ako „*prístup k špecifikácii IT systému, ktorý oddeľuje špecifikáciu funkcionality od špecifikácie implementácie*“.

Už zo samotného názvu je zrejmé, že hlavným ťažiskom MDA je práve aplikačný model. Modelom v MDA je formálna špecifikácia funkcie, štruktúry a správania sa aplikácie alebo systému. Systém je teda pri MDA prístupe k vývoju najprv analyzovaný a následne špecifikovaný ako výpočtovo nezávislý model (*Computation Independent Model, CIM*), taktiež známy ako doménový model. CIM sa sústreďuje hlavne na prostredie, do ktorého bude systém zasadený a samotné požiadavky na systém. Výpočtové a implementačné detaily systému sú na tejto úrovni opisu systému skryté či dokonca ešte neznáme.

CIM je následne transformovaný na platformovo nezávislý model (*Platform Independent Model, PIM*), ktorý už síce obsahuje výpočtové informácie o aplikácii, ale žiadne informácie vzťahujúce sa k platforme, na ktorej bude samotný PIM implementovaný. PIM je až následne transformovaný na platformovo špecifický model (*Platform Specific Model, PSM*), ktorý už obsahuje všetky detailné opisy ku konkrétnej implementačnej platforme. Transformácie modelov sú zobrazené na obrázku 3-10.

Platforma je v MDA definovaná všeobecne ako akákoľvek množina podsystémov a technológii, ktoré poskytujú funkcionalitu cez rozhrania a vzory. MDA platformou teda môže byť napríklad všeobecný štandard ako CORBA či J2EE, ale môže ísť aj o konkrétne implementácie štandardov ako napríklad BEA WebLogic J2EE, či dokonca proprietárnu technológiu ako Microsoft .NET.

[5] Object Management Group, www.omg.org

Obrázok 3-10. Transformácie modelov v MDA.

MDA je zastrešené radom OMG štandardov, vrátane UML, MOF (*Meta-Object Facility*), XMI (*XML Metadata Interchange*) a CWM (*Common Warehouse Metamodel*) a taktiež obsahuje smernice a štandardy aj pre transformácie modelov a generické služby. Štandardy MDA tak spoločne definujú spôsob, ktorým je aj za pomoci nástrojov kompatibilných s MDA možné vyvíjať systémy.

Modely v MDA je nutné špecifikovať v modelovacom jazyku. Môže ísť o všeobecné modelovacie jazyky použiteľné vo viacerých doménach (napr. UML), ale aj o doménovo špecifické modelovacie jazyky. MOF tu zohráva úlohu meta-modelovacieho jazyka, ktorý umožňuje špecifikovať iné modelovacie jazyky a zároveň definuje spôsob ukladania modelov vytvorených v týchto jazykoch do XML dokumentov. Akýkoľvek existujúci modelovací jazyk je teda po vytvorení jeho MOF reprezentácie už možné považovať za kompatibilný s MDA. Príkladmi takto definovaných modelovacích jazykov sú, aj keď pomerne všeobecné, UML a CWM, ktoré sú zároveň súčasťou balíka MDA štandardov. Zatiaľ čo UML sa sústreďuje skôr na objektové modelovanie, CWM sa naopak zameriava na dátové modelovanie.

Spojenie XMI a MOF navyše umožňuje automatickú serializáciu modelov do štandardizovaných XML dokumentov, čím sa zvyšujú možnosti ďalšieho spracovania modelov v rôznych nástrojoch. Príkladom takéhoto úspešného využitia XMI je štandardizovaná XML schéma pre UML modely slúžiaca ako výmenný formát medzi modelovacími nástrojmi.

MDA sa snaží pokryť všetky fázy procesu vývoja softvéru, od doménových modelov, cez analytické a návrhové modely až po modely samotného kódu, pričom dôraz sa kladie na štandardizáciu výmenných formátov samotných modelov.

3.4.2 Výhody MDA

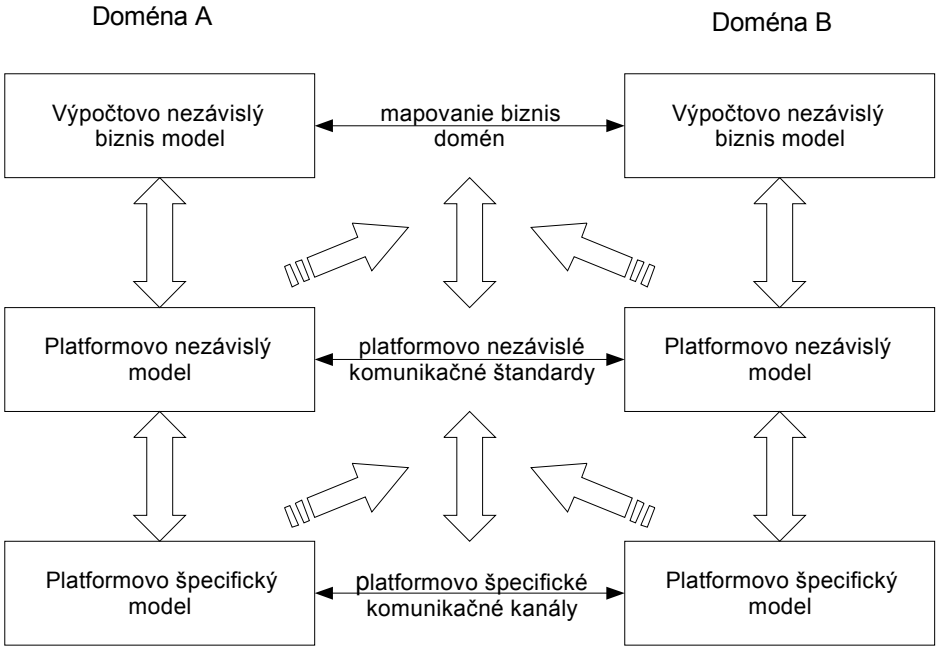
Modely sú vo všeobecnosti abstrakciami systému, ktoré umožňujú nahliadať na systém z množstva rôznych pohľadov. MDA už vo svojej podstate rozdeľuje systém na tri rôzne modely (CIM, PIM, PSM), ktoré sú svojou povahou veľmi rozdielne a vyžadujú inú úroveň špecializácie. Každý z týchto modelov je však možné vyvíjať pomerne nezávisle od ostatných, čím sa vývoj systému zefektívňuje a špecializáciou aj skvalitňuje. Medzi tri hlavné ciele MDA patrí prenosnosť, interoperabilita a znovupoužiteľnosť.

Prenosnosť

Prenosnosť v MDA je zrejmá už zo samotnej definície, kde sa oddeľuje špecifikácia funkcionality systému od implementačných detailov systému. Vysokoúrovňové modely CIM a PIM tak neobsahujú detaily platforiem, na ktorých bude systém nasadený vo forme PSM. Ak sa platforma vyvíja, PIM a CIM zostávajú nezmenené a menia sa len transformácie medzi PIM a PSM. Týmto je zabezpečená aj portabilita smerom k budúcim platformám, kde je nutné vytvoriť iba transformácie z PIM modelov na PSM novej platformy.

Interoperabilita

Komunikácia s inými aplikáciami je bežnou súčasťou veľkých aplikácií. Na obrázku 3-11 sú zobrazené dve skupiny CIM, PIM a PSM modelov, ktoré medzi sebou komunikujú. Interakcie medzi CIM a PIM modelmi je možné analyzovať bez ohľadu na komunikačné kanály, ktorými bude táto komunikácia realizovaná na úrovni PSM modelov. Pretože vertikálne mapovania medzi CIM, PIM a PSM modelmi sú dané explicitne je možné vysokoúrovňovú komunikáciu medzi doménami automaticky transformovať na nižšie komunikačné štandardy či dokonca konkrétne platformovo špecifické kanály.



Obrázok 3-11. Mapovanie horizontálnych vzťahov medzi modelmi.

Znovupoužiteľnosť

Znovupoužiteľnosť je kľúčom k zvýšeniu produktivity a kvality. MDA preto kladie veľký dôraz na kvalitu transformácií medzi modelmi, ktorých výsledkom je kvalitný výstup s využitím dobre overených praktík a vzorov. Takto kvalitný výstup, je ľahšie pochopiteľný, zapamätateľný a v konečnom dôsledku teda aj častejšie použiteľný.

3.4.3 Kritika MDA

Väčšina kritiky MDA pochádza hlavne z úst popredných predstaviteľov agilných metód vývoja softvéru.

Nevhodnosť UML

Jedným z ťažísk kritiky MDA je už samotné využitie vizuálneho UML ako nosného modelovacieho jazyka na opis správania systému. Sekvenčné diagramy, ktoré sú bežne využívané na opis správania, sa totiž zdajú byť pre človeka ťažšie pochopiteľné ako pseudokód, ktorý modelujú. Takéto diagramy vrhajú tieň na efektivitu MDA prístupu

k vývoju softvéru. Navyše je až zarážajúce, že aj časť zo samotných stúpencov MDA považuje UML a OMG štandardy za nevhodné a neefektívne (Fowler, 2004). UML sa vo svojej podstate považuje za nástroj na modelovanie a nemal by slúžiť ako implementačný jazyk, pričom vizuálne jazyky sú nutne doménovo špecifické, čo je v priamom rozpore s využitím UML v MDA (Thomas, 2003).

Otázka nezávislosti modelov

Ďalším diskutabilným miestom MDA je úplná platformová nezávislosť modelov. Takáto nezávislosť je vždy nevyhnutne spojená s určitými kompromismi vo výkone a integrácii systémov, ktoré môžu byť pri veľkých systémoch neželané (Cook, 2004). Ak navyše neexistujú obojsmerné transformácie medzi modelmi, agilnosť a jednoduchosť celého vývojového procesu systému sa vytráca, pretože transformácie je potrebné robiť manuálne.

Vágnosť OMG štandardov

Je tiež zarážajúce ako nejasne sú zadefinované niektoré pojmy OMG štandardov, ako napríklad v otázke definície platforiem, ktoré sú základom celej MDA. Taktiež tvrdenia OMG o priamej podpore radov softvérových výrobkov nie sú v štandardoch dostatočne formálne zachytené (Cook, 2004).

3.4.4 MDA a softvérová architektúra

Väčšina modelov v MDA predstavuje reprezentáciu softvérovej architektúry – v širšom slova zmysle je možné doménové modely a modely systémov považovať za abstrakcie a rôzne pohľady na modely architektúry softvéru.

Architektúra softvéru môže byť opísaná aj pomocou jazykov priamo špecializovaných na opis architektúry (*Architecture Description Language*, *ADL*). V posledných rokoch vzniklo množstvo rôznych ADL s vyjadrovacou schopnosťou zameranou na rôzne aspekty softvérových systémov a rôzne aplikačné domény. Mnohé z užitočných vlastností ADL boli buď zapracované v revíziách samotného UML, alebo špecifikované ako menšie (cez UML profily) či väčšie (MOF) rozšírenia UML. V MDA je teda UML možné považovať za formu ADL.

Niektoré pokročilejšie formalizmy a dynamické charakteristiky určitých ADL však nie je vždy úplne možné zapísať len pomocou UML. Rastúci záujem o MDA a UML v priemysle spolu s dostupnosťou vysoko kvalitných nástrojov na modelovanie architektúry a UML však vo väčšine domén prevažuje nevýhodu, ktorou je práve spomínaná limitácia modelovacích schopností UML.

3.4.5 MDA a nefunkcionálne požiadavky

Nefunkcionálne požiadavky sú veľkým problémom v softvérovej architektúre, pretože sú väčšinou spojené s ťažko merateľnými atribútmi kvality ako výkonnosť, modifikovateľnosť, znovupoužiteľnosť, interoperabilita či bezpečnosť. Aj keď MDA sa nevenuje každému z týchto atribútov zvlášť, nepriamo podporuje a pomáha k dosiahnutiu týchto čiastkových cieľov, z nasledujúcich dôvodov:

- Určitý stupeň interoperability, znovupoužiteľnosti a prenosnosti je zahrnutý už v samotných modeloch, pretože dochádza k vynútenému oddeleniu funkcionality od implementácie.

- MOF a profilovanie UML umožňujú pridávaním nových prvkov rozširovať UML o modelovanie požiadaviek priamo súvisiacich s nefunkcionálnymi požiadavkami. Príkladom takéhoto UML profilu je profil pre výkonnosť, plánovanie a časovanie.

3.4.6 Transformácie modelov a architektúra softvéru

Veľká časť návrhu architektúry softvéru sa venuje problému ako navrhnúť a overiť architektúru softvéru tak, aby spĺňala požiadavky zákazníka a zároveň bola v návrhu systému korektne zapracovaná. Je veľmi zložité systematicky overiť či architektúra spĺňa požiadavky na systém, pretože priame mapovanie medzi požiadavkami a návrhom nie je formalizovaný proces.

V MDA sú všetky modelovacie jazyky po syntaktickej ako aj sémantickej stránke veľmi presne zadefinované meta-modelom. Transformácie medzi týmito modelmi sú taktiež systematické procesy, ktoré sa riadia explicitne danými pravidlami. Táto explicitnosť a potenciálna automatizovateľnosť môže teda veľmi vylepšiť kvalitu a efektívnosť vyhodnocovania architektonického modelu. Príkladom takejto automatickej transformácie sú existujúce implementácie obojsmerných mapovaní medzi PIM a PSM modelmi. Tieto mapovania obsahujú skryté znalosti doménových expertov, zaužívané praktiky a návrhové vzory čím nemalou mierou prispievajú aj k celkovému zvyšovaniu kvality modelov architektúry.

3.4.7 Plánovanie kapacít v ICDE

Jeden z problémov, s ktorými sa stretol vývojový tím ICDE je spojený s plánovaním kapacít nových ICDE inštalácií. V prípade, že ICDE inštalácia podporuje viacero používateľov, záťaž požiadavkami sa zvyšuje a hardvér by mal byť schopný takúto záťaž zvládnuť. Ako sa však hardvér vyťažuje, nemusí byť už schopný spracovávať ďalšie udalosti generované používateľmi čím môže dôjsť k strate dôležitých údajov. Táto situácia je navyše komplikovaná nasledujúcimi skutočnosťami:

1. Každá aplikačná doména a každá jednotlivá inštalácia, v každej doméne bude využívať ICDE iným spôsobom a teda aj generovať inú priemernú záťaž požiadavkami od používateľov.
2. Rôzne inštalácie ICDE budú nasadené na rôzne hardvérové platformy, pričom každá z nich je schopná spracovať rôzny počet používateľov.
3. Systém ICDE bude nasadený na rôzne J2EE servery, pričom každý z nich môže mať iné výkonnostné charakteristiky.

Plánovanie kapacít je určovanie veľkosti inštalácie, v zmysle hardvérových a softvérových kapacít tak, aby bola schopná spracovať predpokladanú záťaž. Na odhad výkonnosti inštalácie sa niekedy používajú matematické modely, no typickejšie je použitie výkonnostných testov na prototypoch alebo kompletných inštaláciách.

ICDE tím sa rozhodol, že vykoná testovacie zaťaženie každej nasadenej inštalácie, čím overí schopnosť tejto inštalácie spracovávať predpokladanú reálnu záťaž. Pre každú inštaláciu je teda potrebné zadefinovať očakávanú charakteristiku záťaže, čo sa týka počtu používateľov ako aj zohľadniť špecifiká správania sa používateľov domény, v ktorej bude inštalácia nasadená. Takéto testy sa následne a najlepšie aj automaticky spustia na hardvéri, ktorý je čo možno najviac podobný tomu, na ktorom bude inštalácia reálne spustená. Výsledkom týchto testov by mali byť charakteristiky priepustnosti a časovej odozvy systému.

3.4.8 MDA v ICDE

Hlavné dôvody prečo sa vývojový tím ICDE rozhodol využiť MDA prístup k vývoju boli:

- V rôznych J2EE aplikačných serveroch sa menia len platformovo závislé podporné kódy a detaily nasadenia. Pomocou MDA je možné navrhnúť všeobecný aplikačný model, z ktorého by konkrétne platformovo špecifické implementácie vygenerovali automatickými transformáciami.
- Generovanie platformovo špecifického podporného kódu a konfigurácií nasadení pre veľa rôznych J2EE aplikačných serverov je dostupných v množstve MDA projektov s otvoreným zdrojovým kódom. Tieto generátory kódu sú bežne spravované veľkými a aktívnymi komunitami a produkujú vysoko kvalitné kódy.
- ICDE tím má rozsiahle skúsenosti s testovaním výkonnosti a záťaže. Refaktoringom ich existujúcich knižníc je možné získať znovupoužiteľný rámec jednoducho využiteľný naprieč rôznymi J2EE platformami. Rozšírením UML s použitím stereotypov je možné tieto testy aj vizuálne definovať. Takýto vizuálny model je následne automaticky transformovaný do spustiteľného testovacieho kódu naviazaného na testovací rámec pre danú platformu.

ICDE tím navrhol UML profil a nástroj, ktorý automatizuje generáciu kompletných testov pre systém ICDE. Vstupom je množina UML diagramov, ktoré vizuálne definujú záťažové testy. Tím sa rozhodol využiť na opis záťaže od používateľa existujúci profil *UML 2.0 Testing Profile*. Výstupom je výkonnostný test spolu s monitorovacími, profilovacími a zobrazovacími nástrojmi. Po spustení takto vygenerovaného testu sa používateľovi zobrazia zozbierané výkonnostné údaje na analýzu spolu s automaticky vygenerovanými grafmi výkonnostných charakteristík.

3.4.9 Zhrnutie

Štandardizácia modelom riadeného vývoja softvéru v podobe prístupu MDA sa v praxi ukázala byť užitočná a ďalej sa pokračuje v jej vývoji. MDA núti tímy vytvárať formálne modely ich aplikácií použitím striktne definovaných modelovacích jazykov a podporných nástrojov. MDA predstavuje ďalšie zvýšenie abstrakcie modelov architektúry, ktoré môže znamenať možnosť zvýšenia produktivity v IT priemysle.

MDA sa stala tiež terčom kritiky z mnohých strán, ktoré kritizujú najmä jej obmedzenia, ktoré sú niekedy až tak zásadné a principiálne, že ich nebude možné odstrániť bez veľkých zmien. Microsoft sa navyše rozhodol nepodporiť MDA štandardy a ide svojou vlastnou cestou modelovacích jazykov vo *Visual Studio IDE*.

Aj napriek týmto faktom sa MDA ukazuje ako zaujímavá alternatíva, ktorá dokáže efektívne a jednoducho vyriešiť niektoré problémy pomocou nových prístupov.

3.5 Architektúry a technológie orientované na služby

V posledných rokoch sa pomaly, ale isto dostávajú vo viacvrstvových architektúrach do popredia riešenia využívajúce webové služby (*Web Services*). Táto architektúra je časťou väčšej a rozsiahlejšej oblasti s názvom Architektúra orientovaná na služby (*Service Oriented Architecture, SOA*).

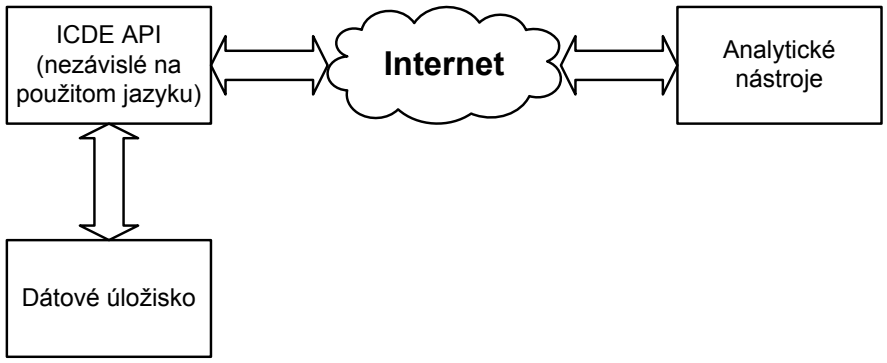
3.5.1 Architektúra orientovaná na služby v prostredí ICDE

Pôvodný návrh ICDE ponúkal Java API, ktoré umožňovalo dodávateľom tretích strán jednoducho a efektívne pristupovať k dátam zozbieraným komponentom pre zber dát a písať aplikácie pre ICDE, ktoré tieto dáta analyzujú a využívajú. Tento prístup vyhovoval väčšine dodávateľom, avšak mal dve veľké nevýhody:

- aplikácie museli byť písané v Jave (alebo musel byť použitý nejaký spôsob ako volať Java funkcie z iných jazykov),
- dopyty na tieto metódy nie sú za bežných okolností povoľované na bezpečnostných bránach (*firewall*). Inými slovami tieto Java metódy boli prístupné len z intranetu. Nanešťastie toto nevyhovovalo riešeniam nasadeným v prostrediach kde sa nedalo zasahovať do sieťovej infraštruktúry, prípadne kde požiadavky museli prechádzať cestou cez cudzie prostredie.

Vývojársky tím sa rozhodol tieto obmedzenia odstrániť aby vyšiel v ústrety dodávateľom ktorí z akýchkoľvek príčin neboli ochotní/schopní písať svoje aplikácie v Jave (napr. C/C++, C#), a ktorých aplikácie museli pristupovať k dátam cez Internet (Obrázok 3-12).

Architektúra, ktorá rieši obe uvedené nevýhody sa nazýva webové služby. Webové služby poskytujú aplikáciám možnosť komunikovať medzi sebou vymieňaním si XML dokumentov pomocou SOAP (*Simple Object Access Protocol*).



Obrázok 3-12. Vzdialený prístup analytických nástrojov k ICDE.

Webové služby sú posledným výkrikom módy vo veľkých a stredne veľkých systémoch kde sa predpokladá interoperabilita medzi viacerými architektúrami.

Tradičné riešenia, ako napríklad J2EE aplikačné servery a posielanie správ je vynikajúce riešenie pre vnútropodnikové aplikácie. Často však padá ak sa ho snažíme prepojiť s iným riešením, ktoré je prepojené s našou aplikáciou cez Internet.

Vo svojej podstate webové služby nie sú ničím novým. Volanie vzdialeného kódu Java podporuje pomocou svojich komponentov. Webové služby však prinášajú možnosť komunikácie rôznych platforiem medzi sebou bez potreby poznania druhej strany. Každá z platforiem, či už ide o Javu, .NET alebo inú implementačnú platformu, dokáže poskytnúť vzdialené služby iným aplikáciám, avšak len ak pracujú na rovnakej platforme. Taktiež majú tieto riešenia problém s komunikáciou cez Internet.

Ďalším problémom je to, že aplikácie ktoré neboli nikdy navrhované tak aby prekročili hranice spoločnosti sú zrazu nútené komunikovať so systémami mimo organizácie.

3.5.2 Systémy využívajúce služby

Prechod na systémy využívajúce služby je výsledkom toho, že organizácie rastú, vyvíjajú sa, spájajú sa, potrebujú nové a nové systémy na podporu svojich biznis procesov a potrebujú tieto systémy rozumným spôsobom prepájať. Potreba integrácie softvérových systémov je rovnako stará ako samotné softvérové systémy.

Prepájanie takýchto systémov je často robené formou papierových dokumentov. Časom sa síce tento proces zautomatizoval, ale len do tej formy že papiere sa prestali posielať poštou. Posielajú sa elektronicky, ale na druhej strane je potrebné tieto údaje znova ručne prepísať do cieľového systému. Cieľový systém ich za pár sekúnd spracuje, ale na to aby sa výsledok vrátil systému ktorý tento proces inicioval je potrebné tieto informácie znova ručne poslať späť. Proces posielania a prepisovania informácií sa zopakuje.

Ako už bolo spomenuté, webové služby sú len jedným z mnohých prístupov, ktoré umožňujú prepájať systémy medzi sebou. Webové služby však umožňujú prepájať systémy na rôznych platformách a cez existujúcu infraštruktúru Internetu.

Jednou z veľkých výhod tohto prístupu je, že pri návrhu a implementácií takéhoto systému je potrebné zaoberať sa len minimálne tým, ako je navrhnutý a implementovaný poskytovateľ resp. používateľ služieb.

Základné princípy na ktorých sú systémy orientované na služby založené nie sú nové. Väčšinou len odrážajú roky skúseností s návrhom, implementáciou, ale hlavne s prevádzkovaním veľkých a stredne veľkých aplikácií, ktorých životný cyklus si vynútil prepájanie a komunikáciu s inými systémami často s úplne odlišnou štruktúrou a postavených na úplne inej platforme.

Môžeme ich rozdeliť nasledovne:

- jednoduchosť,
- nezávislosť,
- zdieľanie rozhraní, nie implementačných detailov,
- kompatibilita založená na pravidlách.

Jednoduchosť

Zásada číslo jedna vraví že webové služby sú z pohľadu klientov nezávislé aplikácie, nie metódy zviazané s aplikáciou klienta. Metóda služby sa vykonáva mimo aktívny proces na klientovi. Najčastejšie dokonca úplne mimo hardvér klienta, požiadavka musí cestovať po sieti na vzdialený server, kde sa vykoná a následne vráti výsledok. Taktiež vývojári aplikácií a poskytovatelia služieb môžu byť geograficky vzdialený.

Kvôli spomenutým obmedzeniam je výhodné klásť veľký dôraz na jednoduchosť. Keďže vývojári nemôžu robiť predpoklady o správaní sa druhej strany iné ako je uvedené v špecifikácii, je vhodné robiť rozhrania čo najjednoduchšie.

Nezávislosť

Webové služby sú autonómne, nezávislé aplikácie. Nie metódy, funkcie, triedy alebo komponenty integrované do klientskych aplikácií. Cieľom je webové služby sprístupniť na sieti kde ich klientske aplikácie môžu nájsť a použiť. Webové služby nemu-

sia vedieť nič o klientskych aplikáciách. Klientske aplikácie však musia presne dodržať špecifikáciu rozhrania jednotlivých služieb.

Webové služby je možné vyvíjať samostatne, nezávisle od aplikácie. Pokiaľ vývojári webových služieb dodržia rozhrania s klientskymi aplikáciami, môžu ľubovoľne meniť implementáciu, algoritmy a klientske aplikácie to nemajú ako zistiť. Z princípu ich to ani nemusí zaujímať.

Istým problémom môže byť kompatibilita jednotlivých verzií tej istej služby. Ak vezmeme v úvahu neznáme množstvo klientskych aplikácií, ku ktorým nemáme prístup, úprava týchto klientov neprichádza do úvahy.

Riešením je spätná kompatibilita. Ak dodržíme už raz zverejnenú a používanú funkcionalitu, nič nám nebráni ďalšie vlastnosti a správanie pridať do novej verzie tej istej webovej služby. Aplikácie, ktoré používajú stále tú istú starú funkcionalitu nebudú obmedzené a služba môže súčasne obsluhovať iné aplikácie s inými požiadavkami.

Keďže webové služby sú jednak nezávislé aplikácie a jednak zo svojej podstaty „slúžia“ nielen dôveryhodným klientskym aplikáciám (vnútropodnikové nasadenie) ale často aj úplne cudzím (Internet), treba pri ich implementácií a nasadení brať väčší ohľad aj na bezpečnosť, robustnosť ďalšie nefunkcionálne vlastnosti.

Zdieľanie rozhraní, nie implementačných detailov

Návrh a implementácia veľkých systémov so sebou prináša vyššiu zložitosť ale tiež komplexnosť. Technológie a architektúry orientované na služby tento problém riešia dobrým oddelením klientskych aplikácií a samotných služieb a jednoduchosťou. Čím jednoduchšia myšlienka, tým je menej náchylná na chyby (prípadne tým ľahšie sa potenciálne chyby odhalia). Tiež čím je systém webových služieb jednoduchší, tým viac kapacít ostáva na riešenie naozaj zložitých oblastí.

Klientska aplikácia a webová služba by mali zdieľať len svoje rozhranie. Aplikácia by mala vedieť *aké* správy má poslať webovej službe ak požaduje nejakú službu, prípadne v *akom poradí* má poslať jednotlivé správy.

Kompatibilita založená na pravidlách

Ak chcú byť klienty kompatibilné s webovými službami, nestačí vedieť iba aké vstupy služby očakávajú a aké výstupy produkujú, ale aj ďalšie podrobnosti, napr. či musia byť správy šifrované, či záleží na poradí v akom služba správy prijme (pre prípad ak by správy nedorazili v poradí akom boli odoslané).

Tieto nefunkcionálne požiadavky sú definované pravidlami. Nie sú len zmienkou v dokumentácii ku webovej službe. Pravidlá sú písané tak aby sa dali strojovo spracovať a aby sa na ich presadenie dal implementovať nástroj – program. Ak sa napríklad dodávateľ rozhodne, že na šifrovanie symetrického kľúča mu už nestačí 1024 RSA šifra ale potrebuje 2048 RSA, stačí keď zmení pravidlo. Klientska aplikácia si ho prečíta a bez zásahu (a často aj bez vedomia) používateľa/vývojára miesto RSA 1024 začne používať RSA 2048.

3.5.3 Webové služby

Ako už bolo povedané, webové služby sú veľmi podobné existujúcim technológiám z danej oblasti. Na rozdiel od iných podobných riešení majú jednoduchú architektúru a podporujú viacero platforiem. Možno práve neutralita z pohľadu podporovaných

platforiem webovým službám zabezpečila priazeň viacerých popredných softvérových spoločností ktoré ich začali podporovať. Alebo aspoň ich hneď na začiatku nepochovali a neútočili na ne len kvôli tomu že tento nápad nepochádzal z ich dielne – čo sa stalo v histórii osudným už viacerým dobrým nápadom.

Keď porovnáme ostatné riešenia ktoré v tejto oblasti existujú, zistíme že v skutočnosti všetky poskytujú len 4 základné funkcie:

- nájdenie vhodnej služby (UDDI),
- nájdenie technickej špecifikácie (WSDL),
- použitie služby (SOAP),
- špeciálne požiadavky (WS-* štandardy).

3.5.4 SOAP

Jedná sa o prvý pôvodný štandard z oblasti webových služieb a doteraz je najpoužívanejší. Skratka SOAP pôvodne znamenala *Simple Object Access Protocol*. Niektorí ju dodnes mylne prekladajú ako *Service Oriented Architecture Protocol*. Pravda je taká, že v dnešnej dobe sa už SOAP považuje za bežné slovo a netreba v ňom okrem historického pôvodu hľadať skratku.

SOAP je jednoduchý protokol – klient pošle požiadavku vo forme XML dokumentu a odpoveď dostane taktiež vo forme XML dokumentu. Štruktúru SOAP správy znázorňuje obrázok 3-13. SOAP správa sa skladá z nepovinnej hlavičky a z povinného samotného tela správy. Okrem toho že je hlavička nepovinná, nie je ani definovaná jej štruktúra. To dáva dostatočnú voľnosť pri vytváraní nových štandardov. Hlavička aj telo správy sú obalené v tzv. obálke.

SOAP správy môžu byť prenášané ľubovoľným protokolom. Najčastejšie sa využíva HTTP, ale je možné využiť ktorýkoľvek z rodiny protokolov TCP/IP.

Obrázok 3-13. Štruktúra SOAP správy.

3.5.5 WSDL

Webové služby sú opísané pomocou WSDL (*Web Services Description Language*). WSDL predstavuje technickú špecifikáciu vo forme XML dokumentu, ktorá obsahuje

informácie o rozhraniach služby (vstupy, výstupy, dátové typy). Reprezentácia WSDL pomocou XML je vhodná pre strojové spracovanie – niektoré vývojové nástroje dokážu automaticky z metód vygenerovať WSDL dokument a naopak pomocou existujúceho WSDL dokumentu priamo pomôcť vývojárovi pri použití služby.

3.5.6 UDDI

UDDI (*Universal Description, Discovery and Integration*) adresár slúži na vyhľadávanie medzi webovými službami v prípade že hľadáme konkrétnu službu, ktorá spĺňa naše požiadavky. Tento systém sa však ukázal ako nesprávna cesta, lebo vývojári často pre potreby svojich aplikácií vyvíjajú vlastné webové služby, prípadne priamo komunikujú s dodávateľom, ktorý tieto služby vyvíja na mieru. V dnešnej dobe sa už UDDI nepoužíva.

3.5.7 Bezpečnosť, transakcie a spoľahlivosť

Medzi najčastejšie doplnkové požiadavky patrí napríklad bezpečnosť. Keďže SOAP správy často cestujú nezabezpečeným prostredím Internetu, vyplynula požiadavka šifrovať posielané XML dokumenty a autentifikovať odosielateľa požiadavky.

Tieto problémy rieši štandard WS-Security podporou kryptografie, digitálneho podpisu príbuzných technológií. Práve tieto informácie je možné ukladať do voliteľnej hlavičky SOAP protokolu.

Podobne ako na bezpečnosť, aj na iné problémy existujú štandardy, resp. sa tieto dajú implementovať. Plánom je vo finálnej verzii podporovať transakcie a tiež spoľahlivé doručovanie správ, ktoré zabezpečí aby služba vedela rozlíšiť v akom poradí boli správy odoslané.

3.5.8 Použitie webových služieb

Webové služby majú 2 základné oblasti nasadenia:

- veľké aplikácie, ktoré sa skladajú z viacerých prepojených systémov a použitie webových služieb je pre ne vhodným riešením,
- malé vnútropodnikové aplikácie.

V prvom prípade webové služby nemajú okrem niektorých proprietárnych riešení žiadneho konkurenta. Webové služby však vytláčajú z tohto segmentu ostatné riešenia kvôli svojej otvorenosti, štandardizácii a svojim ďalším výhodám a vlastnostiam.

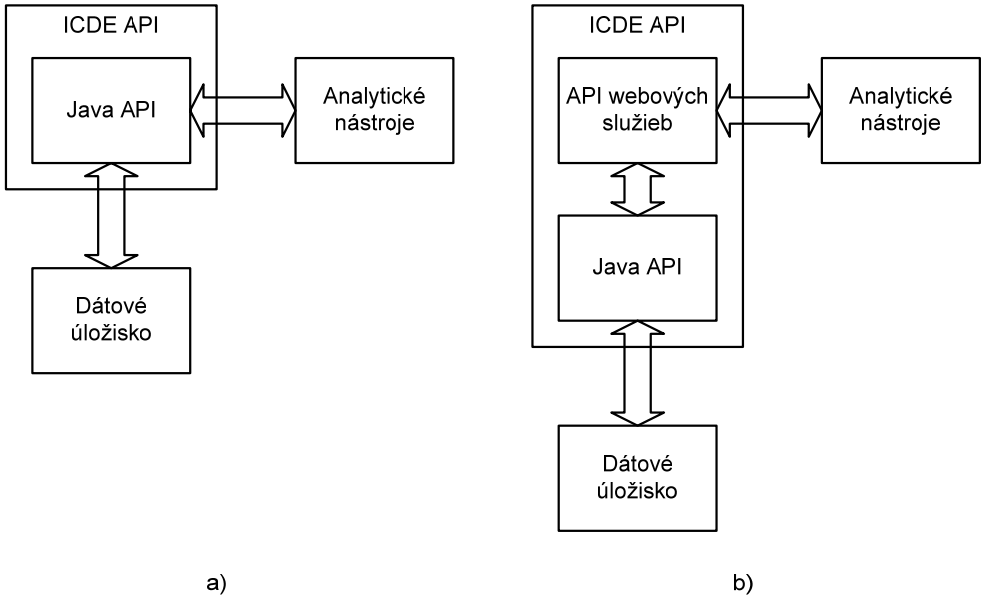
V druhej oblasti okrem webových služieb existujú riešenia využívajúce napr. Javu/.NET, ktoré sú síce bohatšie na funkcionalitu, ale nedisponujú vlastnosťami webových služieb, ktoré sa ukázali byť výhodné v určitých prípadoch. Jedná sa predovšetkým o interoperabilitu v heterogénnom prostredí a jednoduchosť.

3.5.9 Webové služby v ICDE

Ako už bolo spomenuté, použitie Java API v systéme ICDE sa neukázalo byť šťastnou voľbou. Z nepochopiteľných príčin sa tento prístup nepozdával dodávateľom, ktorí by radi vyvíjali aplikácie pre ICDE, ale nevenovali sa vývoju v Jave, prípadne už mali predtým hotové analytické nástroje implementované v inom jazyku.

Bolo potrebné navrhnúť a implementovať systém ktorý by umožnil aj tejto skupine dodávateľov vyvíjať aplikácie pre ICDE a webové služby sa ukázali ako vhodný prístup. Java priamo podporuje webové služby (WSDL aj SOAP), takže stačilo existujúce metódy QueryAPI a WriteAPI z pôvodného Java API sprístupniť cez webové služby. Výsledkom tejto malej úpravy bolo riešenie oboch problémov spomenutých na začiatku tejto kapitoly.

Na obrázku 3-14a možno vidieť architektúru ICDE pred použitím webových služieb – ICDE API tvorí len Java API rozhranie. Obrázok 3-14b zobrazuje modifikované rozhranie vrátane vrstvy webových služieb, ktorá sa jednoducho včlenila medzi analytické nástroje a samotné Java API. Samotné webové služby nevykonávajú žiadnu inú činnosť ako volanie pôvodných metód Java API, čo umožnilo ponechať pôvodnú implementácia nedotknutú.



Obrázok 3-14. ICDE API pred a po použití webových služieb.

Na lokálnej sieti ICDE používalo autentifikáciu pomocou používateľského mena a hesla. V otvorenom priestore Internetu je však potrebná silnejšia ochrana. WS-* štandardy webových služieb podporujú viacero možností vrátane štandardov X-509 a Kerberos. Taktiež sú podporované silné kryptografické algoritmy, ktoré sú v súčasnosti postačujúce na šifrovanie prenášaných dát.

3.5.10 Zhrnutie

Webové služby riešia problém interoperability medzi systémami. Predstavujú jednoduchú a efektívnu architektúru postavenú na existujúcich myšlienkach, ktorá rieši dva základné nedostatky – platformovú závislosť a nedostupnosť medzi sieťami.

Ich veľkou výhodou je, že sa na nich zhodli najväčší hráči v počítačovom priemysle ako Microsoft, IBM, Sun a ďalší. Nemalo by teda dochádzať ku konkurenčným bojom,

ale sily by sa mali sústrediť na vylepšovanie existujúcich a vývoj nových štandardov pre webové služby, čo možno považovať za jednu z ich najväčších výhod.

Webové služby sú v súčasnosti primerane zdokumentované samotnými štandardami, pričom každý dodávateľ vývojových nástrojov má tiež kvalitne (v rámci svojich zvyklostí) zdokumentované implementačné detaily na svojej platforme.

Ďalšie informácie o problematike webových služieb možno nájsť v (Zimmermann et al., 2004, Alonso et al., 2004, Chatterjee, 2004).

3.6 Web so sémantikou

3.6.1 Motivácia pre web so sémantikou v ICDE

Aplikácia ICDE predstavuje platformu pre spoluprácu nástrojov tretích strán. Jeden nástroj môže implementovať komplexné postupy získavania znalostí z dostupných databáz a výsledky uložiť v dátovom úložisku ICDE. Iný nástroj môže tieto dáta využiť a vykonať ďalšie analýzy, ktoré poskytnú používateľovi iný pohľad na dostupné dáta.

Aby takéto scenáre boli možné, musia byť nástroje schopné efektívne zdieľať dáta, čo spočíva najmä v správnej interpretácii dát z rôznych zdrojov. Najčastejšie sa tento cieľ dosahuje vytvorením dátovej štruktúry, ktorá je spoločná pre všetky aplikácie, ktoré pracujú s dátami a ktorá okrem formátu zdieľaných dát definuje aj ich význam.

Jednou možnosťou riešenia v platforme ICDE je zadefinovať štruktúru tabuliek v dátovom úložisku ICDE a požadovať od všetkých spolupracujúcich nástrojov jej využívanie na zdieľanie dát. Nemôžeme však predpokladať, že takáto štruktúra bude vyhovovať všetkým nástrojom pre uloženie ľubovoľných dát, ktoré by sa ich tvorcovia rozhodli zdieľať. V konečnom dôsledku by veľa nástrojov muselo byť integrovaných individuálne za pomoci vývojového tímu ICDE, teda zdĺhavo a neflexibilne.

Flexibilnejší prístup umožňuje nástrojom využívať ľubovoľné dátové štruktúry podľa potreby. Predpokladom je, aby boli ďalšie nástroje schopné dynamicky objavovať tieto štruktúry a pochopiť ich obsah bez toho, aby v nich táto znalosť bola natvrdo naprogramovaná.

V súčasnosti už bežne využívame dátové štruktúry, ktoré sú samoopisné a ktoré program dokáže spracovať a používať. Príkladom je meta-jazyk XML (*eXtensible Markup Language*), ktorý umožňuje flexibilne zapísať dátové štruktúry, ale sám o sebe nedefinuje žiaden význam zapísaných dát. Program teda dokáže dáta prečítať, ale ich význam, potrebný pre spracovanie musí mať natvrdo zadaný. Dôvodom je, že XML umožňuje zapísať tú istú informáciu rôznymi spôsobmi, pričom každý z nich je korektný a bez problémov interpretovateľný človekom (typickým príkladom je rozličné pomenovanie XML značky, ktorá kóduje tú istú informáciu z hľadiska významu – použitie iného slovníka).

Ak by sme nástroje integrované v ICDE nútili používať jeden slovník XML značiek, stratili by sme flexibilitu prístupu a dostali sa k rovnakým problémom ako pri prvej predstavenej možnosti zdieľania cez štruktúry tabuliek. Potrebujeme mechanizmus prepojenia rôznych slovníkov na úrovni významu, ktorý by programom umožnil programovo nachádzať výrazy, ktoré opisujú rovnaké koncepty. Tento mechanizmus

programového prekladu pojmov do slovníka konkrétneho nástroja umožní flexibilné zdieľanie a prácu s dátami z viacerých zdrojov.

3.6.2 Web so sémantikou

Riešenie problému automatizovaného objavovania významu dát predstavuje súbor technológií webu so sémantikou, ktorý umožňuje opísať dáta a explicitne tak definovať ich význam. Základom sú metadáta v spojení s formálnymi jazykmi a protokolmi.

Dáta o dátach – metadáta

Metadáta predstavujú jeden z kľúčových prvkov webu so sémantikou. XML a technológie, ktoré s týmto jazykom súvisia poskytujú flexibilný mechanizmus opisu dát z hľadiska označenia entít a ich častí avšak poskytujú iba slabé možnosti opisu vzťahov medzi nimi.

V zápise XML z príkladu 3-5 vidíme opis osoby *Person* a transakcie *Transaction*, ktorej sa táto osoba zúčastnila. Na príklade je vidieť problém identifikácie rôznych vzťahov medzi osobou a transakciu. Program, ktorý chce odhaliť adresu elektronickej pošty klienta transakcie musí vedieť, že značka *Client* má rovnaký význam ako značka *Name*. Ani v tom prípade nemusí uspieť, ak je obsah značiek rozdielny. Človek však s interpretáciou príkladu a odhalením adresy el. pošty problém nemá.

```
<example>
  <Person id="123">
    <Name>J Doe</Name>
    <Email>doe@myplace</Email>
  </Person>
  <Transaction transID="567">
    <Client>Josef Doe</Client>
    <Downtick>500</Downtick>
  </Transaction>
</example>
```

Príklad 3-5. Príklad zápisu transakcie v XML.

Riešenie problému reprezentácie vzťahov predstavuje rámec pre opis zdrojov (*Resource Description Framework, RDF*). Pomocou RDF dokážeme modelovať vzťahy medzi ľubovoľnými entitami v trojiciach {subjekt, predikát, objekt}.

Základom RDF je označenie entít a vzťahov medzi nimi pomocou URI (*Uniform Resource Identifier*). URI je textový reťazec, ktorý je pre každú entitu svetovo unikátny a jednoznačne ju identifikuje. Poznáme URI vo forme URL (*Uniform Resource Locator*), ktoré identifikuje zdroj na základe jeho umiestnenia (napr. webová stránka prístupná protokolom HTTP). Špeciálnym typom URI je aj URN (*Uniform Resource Name*), ktorý identifikuje zdroj jeho pomenovaním (napr. ISBN knihy). Je potrebné si uvedomiť, že URI je vo svojej podstate stále iba jednoznačný identifikátor (vo forme URL) a preto nemusí byť platným odkazom na umiestnenie zdroja.

RDF predstavuje iba všeobecný spôsob zápisu vzťahov medzi entitami a neobsahuje žiadny mechanizmus, ktorý by umožňoval automatizovanú identifikáciu vzťahov alebo aplikáciu reštrikcií na entity zúčastnené vo vzťahu. Algoritmus, ktorý spracúva čisté RDF nemá ako overiť konzistenciu zapísaných tvrdení, prítomnosť všetkých vyžadovaných atribútov a pod.

Riešením je použitie tzv. schema jazykov. Tak ako sú XML dokumenty napojené na svoje XMLSchema definície vieme napojiť aj RDF dokument na RDFSchema (RDFS). RDFS umožňuje definovať triedy a ich vzťahy (napr. hierarchie), definovať rôzne vlastnosti a prepojiť ich s triedami. Následne v RDF možno vyjadriť príslušnosť zdroja do niektorej triedy a vyjadriť tak základnú sémantiku opisovaného zdroja. Tento prístup sa ponáša na štandardnú objektovo-orientovanú paradigmu, avšak má tú výhodu, že jeden zdroj (jedna inštancia) môže byť viacerých typov (inštancia viacerých tried). Ďalšou výhodou je definovanie vlastností tried nezávisle od samotných tried. Každá vlastnosť má definovaný svoj definičný obor (*domain*) a obor hodnôt (*range*) a môže byť znovupoužitá medzi ľubovoľnými dvoma triedami.

Vzťah medzi RDF a RDFS môžeme chápať tak, že RDFS definuje ontológiu, zatiaľ čo RDF opisuje inštancie ontológie. Pod slovom ontológia označujeme terminológiu (slovník), všetky podstatné koncepty a vzťahy v určitej doméne. RDFS nám umožňuje robustným spôsobom definovať metadáta, avšak na kompletný opis konkrétnej domény potrebujeme ešte silnejšie vyjadrovacie schopnosti, ktoré umožňujú dedukcie a odvodzovanie nad metadátami.

Ontológie

Slovo ontológia ma v informatike dva rôzne významy. Jedna interpretácia definuje ontológiu ako ľubovoľný model, ktorým dokážeme vyjadriť vzťahy medzi doménovými konceptmi. V takom prípade ich môžeme porovnávať z hľadiska sémantickej bohatosti v tzv. spektre ontológií (Obrázok 3-15). Ako postupujeme v obrázku z ľavého dolného rohu doprava hore, tak sa zvyšuje vyjadrovacia sila modelu z hľadiska sémantiky. Na „slabšej“ strane sa dajú vyjadriť iba veľmi jednoduché významy, zatiaľ čo na „silnejšej“ strane sa dajú vyjadriť ľubovoľne komplikované významy.

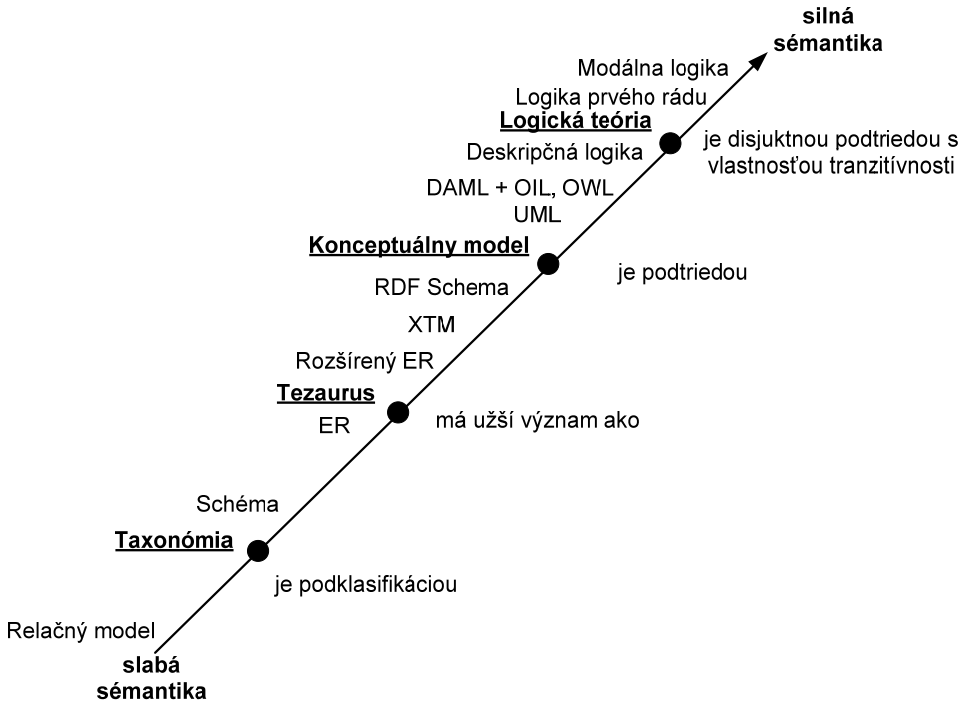
Ontológia teda môže byť jednoduchá taxonómia (čiže klasifikácia znalostí v hierarchickej štruktúre), ale aj logická teória s možnosťou definovania komplexných vzťahov medzi znalosťami).

Na obrázku 3-15 si môžeme všimnúť aj umiestnenie jazykov OWL a RDF (resp. RDF Schema) v spektre ontológií. Pod slovom ontológia sa čoraz častejšie chápe práve model vyjadrený v jazyku OWL, keďže práve tento jazyk sa stal oficiálnym jazykom iniciatívy webu so sémantikou. V ďalšom texte teda pod slovom ontológia budeme chápať práve model vyjadrený v jazyku OWL.

Jazyk OWL stavia na slovníku RDFS a pridáva doň ďalšie elementy a atribúty. Rozdeľuje vlastnosti na dátové a objektové a pridáva k nim ďalšie atribúty – umožňuje definovať vlastnosť ako symetrickú, inverznú, tranzitívnu a pod. OWL umožňuje špecifikovať požadovanú kardinalitu vlastnosti, resp. minimálnu alebo maximálnu možnú kardinalitu. Veľmi dôležitou je z hľadiska možností zdieľania dát možnosť označiť niektoré triedy, vlastnosti alebo inštancie za ekvivalentné. OWL teda priamo počíta so vzájomným mapovaním významu viacerých označení toho istého konceptu. Asi najpodstatnejšou je však možnosť zapisovať v jazyku OWL logické pravidlá, ktoré umožňujú automatické usudzovanie nad informáciami a odvodzovanie nových faktov.

Ako jednoduchý príklad môžeme zobrať model ľudských vzťahov, v ktorom definujeme osoby a vzťahy rodič – potomok medzi jednotlivými osobami. V prípade, že chceme model rozšíriť o ďalšie vzťahy medzi osobami (napr. starý rodič, strýko, súrodenec) alebo spresniť už existujúce vzťahy (namiesto rodič mama alebo otec, či namiesto súrodenec brat alebo sestra) môžeme postupovať tak, že tieto vzťahy

explicitne vložíme pre každú inštanciu. Lepší spôsob je zadefinovať vzťahy na meta-úrovni pomocou logických pravidiel – pridať do modelu pravidlá definujúce kedy je osoba strýkom inej osoby a pod. Keďže odvodzovací mechanizmus dokáže tieto pravidlá vyhodnotiť, dostaneme v konečnom dôsledku komplexnejší model, ktorý obsahuje aj fakty, ktoré sme doň explicitne nezadali, ale ktoré môžeme použiť v dopytoch a systém na ne bude vedieť odpovedať.



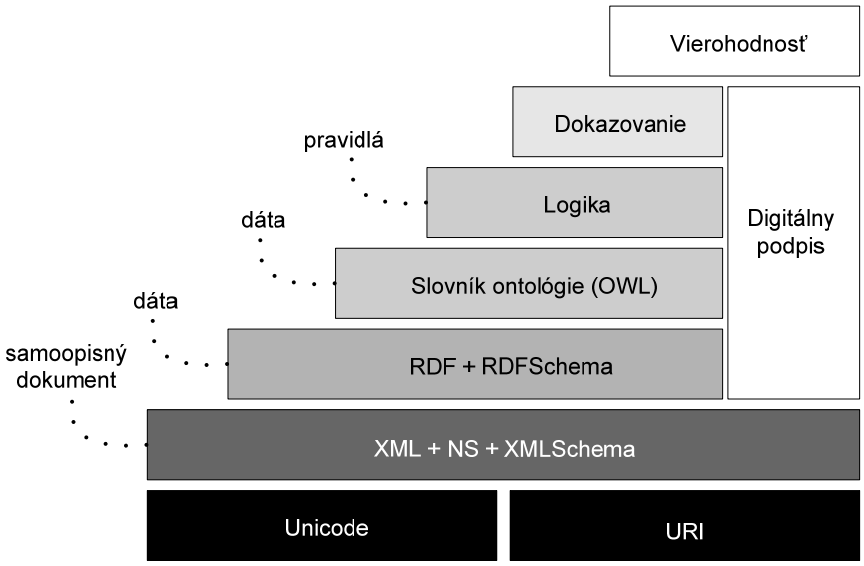
Obrázok 3-15. Spektrum ontológií, podľa (Studer, 1998).

Vrstvy webu so sémantikou

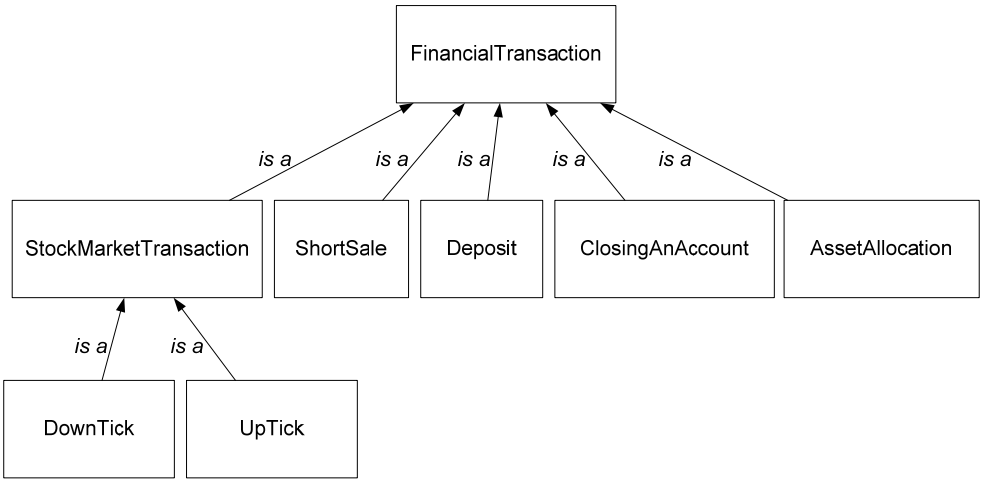
Web so sémantikou je implementovaný vo vrstvách webových technológií a štandardov. OWL využíva a rozširuje slovník RDFS a je zapísaný pomocou výrokov RDF. Samotné RDF je zase vyjadriteľné vo formáte XML s využitím URI. Na obrázku 3-16 vidíme umiestnenie ontologickej vrstvy v celkovom kontexte. Vrstvy nad OWL sú predmetom aktuálneho výskumu, ktorý smeruje k tomu, aby stroje vedeli autonómne posúdiť pravdivosť a vierohodnosť informácií na webe a vyhľadávať relevantné a spoľahlivé informácie.

3.6.3 Ontológie v ICDE

Systém ICDE na riešenie problémov integrácie nástrojov tretích strán mohol využiť technológie webu so sémantikou a najmä ontológií, ktoré definujú spoločný slovník v doméne. Tou je pre ICDE napr. analýza finančných transakcií. Na obrázku 3-17 je zobrazená časť finančnej ontológie, ktorá definuje hierarchiu typov transakcií a zavádza tak v modeli potrebnú abstrakciu. Jeden nástroj tak môže pracovať s *DownTick* a *UpTick* transakciami, pričom druhý nástroj pracujúci s finančnými transakciami tieto pojmy nemusí poznať ale vie využiť to, čo majú spoločné s abstraktnou *FinancialTransaction.*

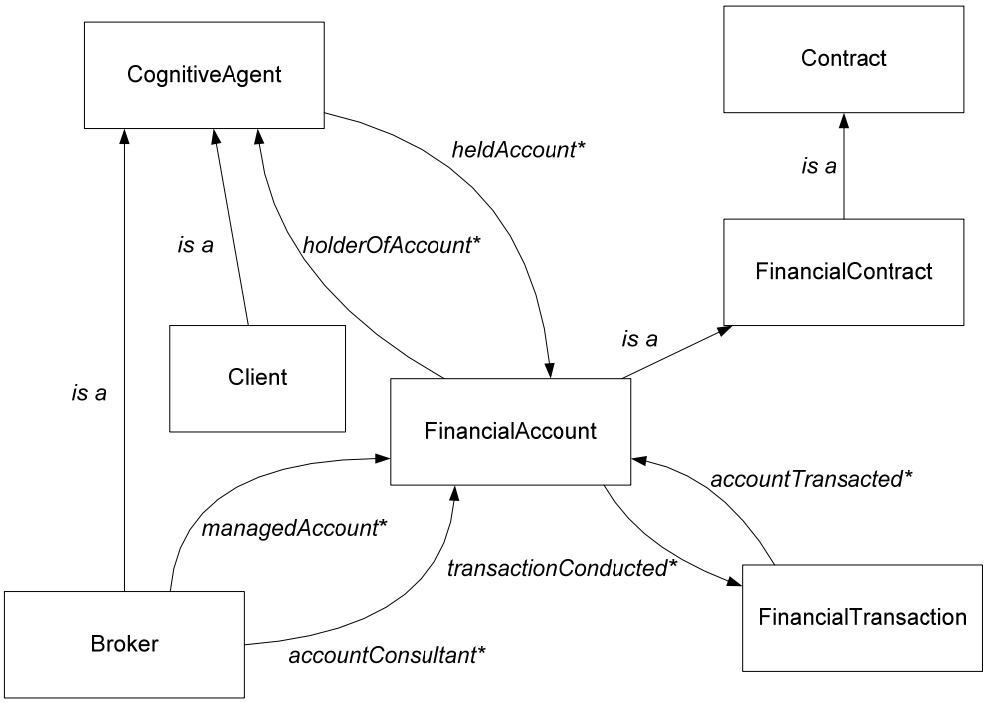


Obrázok 3-16. Vrstvy webu so sémantikou, podľa (Berners-Lee, 2001).



Obrázok 3-17. Taxonómia finančných transakcií.

Ontológia nepozostáva iba z taxonómie tried, ale aj z definícií rôznych vzťahov medzi nimi. Tieto vzťahy môže použiť odvodzovač (*reasoner*) na objavovanie nových informácií o jednotlivých entitách. V príklade na obrázku 3-18 sú uvedené vzťahy medzi účtami, ich vlastníkmi, transakciami a burzovými maklérmi. Odvodzovač môže tieto vzťahy použiť pre odvodenie nových vzťahov medzi konkrétnym klientom a maklérom, ktoré v dátach nie sú explicitne zadané (klient a maklér sú prepojení cez účet, ktorého sa transakcia týkala). Zdieľaná ontológia teda môže prispieť k objaveniu nových informácií a vzťahov, ktoré by možno ostali nepovšimnuté.



Obrázok 3-18. Vzťahy medzi triedami v ontológii.

3.6.4 Sémantické webové služby

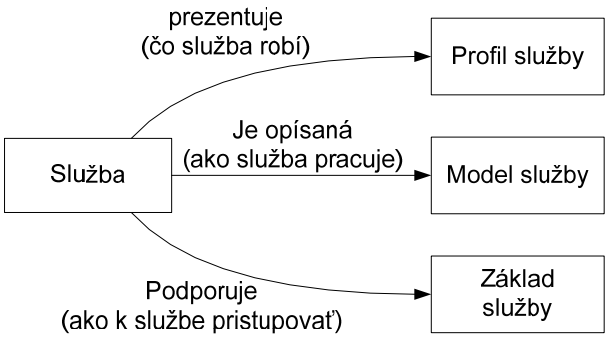
Webové služby a servisne orientovaná architektúra predstavujú dôležitý aspekt webu so sémantikou. Technológie opísané v tejto kapitole boli primárne navrhnuté s cieľom poskytnúť ľuďom a programom možnosti sprístupňovania dát na základe ich významu, teda sémantické vyhľadávanie informácií. S postupným vývojom webu so sémantikou sa čoraz viac kladie dôraz aj na sémantické vyhľadávanie služieb. Webové služby, ktoré takéto vyhľadávanie podporujú označujeme prívlastkom sémantické.

Fungovanie webu so sémantikou závisí od ochoty ľudí vytvárať a spravovať metadáta o informáciách publikovaných na webe. Z obrovského množstva informácií, ktoré sú dostupné na webe je zatiaľ anotovaných iba veľmi málo, aj keď existujú nástroje, ktoré túto činnosť v určitej miere podporujú (či už počas vytvárania samotného obsahu alebo až „ex post“). Ľudia často nevidia priamu výhodu vytvárania metadát o dátach, nevedia o možnostiach, ktoré sa tým otvárajú a nie sú nijako nútení ich vytvárať.

Naproti tomu webové služby už vo svojej podstate pracujú s metadátami. Tieto metadáta však opisujú iba správy, ktorými sa služba volá a ktorými odpovedá, bez explicitného definovania toho, aký je presný význam správ a čo vlastne služba vykonáva. Existujúce opisy webových služieb dostupné na webe sú neštruktúrované a určené pre ľudí, nie pre programy. Plne automatizovaná orchestrácia webových služieb do väčších systémov je teda stále viac snom ako realitou, avšak poskytovatelia webových služieb si uvedomujú prítomnosť metadát a sú motivovaní vytvárať čo najlepšie opisy svojich služieb tak, aby boli vyhľadateľné a zakomponovateľné do iných systémov. To dáva predpoklady pre použitie technológií webu so sémantikou na opis webových služieb.

Opisy sémantických webových služieb špecifikujú kompozíciu služieb (služba sa skladá z viacerých služieb alebo je naopak časťou väčšieho celku), opisujú biznis logiku služby na abstraktnejšej úrovni a poskytujú bázu znalostí pre odvodzovacie systémy, ktoré dokážu inteligentným pospájaním služieb vytvoriť ucelený softvér.

Jedným z jazykov vyvíjaných pre opis sémantických webových služieb je OWL-S, ktorý stavia na jazyku pre opis ontológií OWL. Opis webovej služby v jazyku OWL-S sa skladá z troch častí – profilu služby, procesného modelu a protokolu zasielania správ, ktorý predstavuje základ (*grounding*) služby (Obrázok 3-19).



Obrázok 3-19. Ontológia webovej služby (najvyššia úroveň).

Profil služby

Profil služby slúži pri vyhľadávaní sémantickej webovej služby, špecifikuje „čo služba robí“ spôsobom, ktorý dokážu využiť vyhľadávacie agenty a ktorý im umožňuje rozhodnúť o vhodnosti služby z hľadiska ich potrieb. V profile sú uvedené aj obmedzenia z hľadiska aplikovateľnosti a kvality služby. Profil môže definovať aj požiadavky, ktoré musí spĺňať konzument služby, aby mohol službu úspešne použiť.

Model služby

Model služby (nazývaný aj procesný model) hovorí konzumentovi služby o tom, ako službu používať definovaním významu jednotlivých požiadaviek, podmienok, pri ktorých sa získajú určité výstupy (výsledky) a tam, kde je to potrebné aj jednotlivé kroky, ktoré vedú k daným výstupom. Na sémantickej úrovni teda definuje spôsob, ako službu zavolať a proces, ktorý sa po zavolaní vykoná. Pre komplikované služby môže byť tento opis použitý v najmenej štyroch rôznych prípadoch:

- Vykonanie podrobnejšej analýzy, či služba naozaj spĺňa požiadavky;
- Poskladanie opisov rôznych služieb pre vykonanie špecifickej úlohy;
- Koordinácia aktivít jednotlivých účastníkov počas priebehu ustanovenia služby;
- Sledovanie vykonávania služby.

Základ služby

Základ služby špecifikuje detaily prístupu agenta k službe. Typicky definuje komunikačný protokol, formáty správ a ďalšie detaily špecifické pre konkrétnu službu (napr. čísla portov). Základ služby musí špecifikovať jednoznačný spôsob výmeny dátových elementov pre každý sémantický typ vstupu a výstupu špecifikovaný v modeli služby (použitú serializačnú techniku).

OWL-S nie je kompletnou špecifikáciou ani stabilným štandardom v oblasti sémantických webových služieb, ale predstavuje dobrý základ pre pridávanie významu k webovým službám. V súčasnosti však ešte neumožňuje plne automatizované objavovanie a kompozíciu webových služieb.

3.6.5 Zhrnutie

Web so sémantikou sa stal v posledných rokoch populárnou výskumnou témou, avšak nie všetky projekty stavajúce na technológiách webu so sémantikou sledujú pôvodné ciele vytvárania, vyhľadávania a spracúvania sémanticky bohatých metadát spracovateľných strojom.

Jedným z dôvodov, prečo sa web so sémantikou v praxi zatiaľ príliš neujal je problém správy dát a najmä metadát. Kto vytvorí podrobné metadáta o existujúcich službách a informáciách na webe? Ako bude zabezpečená integrita, presnosť a úplnosť metadát?

Úplne iným problémom je problém ochrany súkromia, zabezpečenia a dôveryhodnosti. Najviditeľnejšie je to v prípade webových služieb. Potrebujeme mechanizmy, ktoré vyhodnotia poskytovateľa webovej služby nielen z funkcionálneho pohľadu (poskytuje službu, ktorú potrebujeme?), ale aj z pohľadu jeho spoľahlivosti a dôveryhodnosti (deje sa naozaj s našimi dátami iba to, čo poskytovateľ služby deklaruje?).

Napriek problémom predstavuje web so sémantikou v podobe použitých technológií zaujímavé riešenie pre softvérového architekta. Web so sémantikou možno chápať ako veľmi voľne zviazanú konglomeráciu spolupracujúcich softvérových nástrojov a technológií. Práve táto voľná zviazanosť je to, čo nás z hľadiska architektúry softvéru zaujíma, keďže zvyšuje šance na znovupoužiteľnosť a výrazne zlepšuje udržiavateľnosť systému.

3.7 Softvérové agenty

Použitie metafory agentov pri tvorbe softvérových systémov pomáha lepšie reprezentovať komplexné javy a riešiť zložité úlohy. Agentové technológie majú pôvod v oblasti distribuovanej umelej inteligencie, avšak rozšírili sa do ďalších odvetví informatiky a informačných technológií.

3.7.1 Softvérový agent

Výraz softvérový agent (tiež konateľ) nemá jednoznačnú definíciu. Neraz sa možno stretnúť s označením agenta prívlastkom inteligentný či autonómny. S pojmom agent sa spájajú nasledovné definície, ktoré zároveň slúžia ako vymedzenie agentov od programov, objektov a expertných systémov:

- agenty sa nachádzajú v určitom prostredí,
- agenty sú schopné prispôsobivo a samostatne reagovať s cieľom splniť požiadavky.

Prispôsobivosť agentov je významná črta ich správania, ktorá v sebe zahŕňa tieto pojmy:

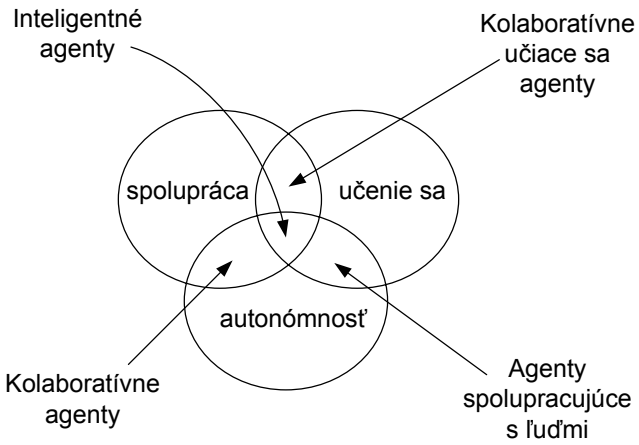
- odozva – agenty vnímajú svoje okolie a reagujú na zmeny, ktoré sa v ňom dejú,
- aktivita – snaha agentov dosiahnuť stanovený cieľ,

- sociálna interakcia – agenty spolupracujú v snahe dosiahnuť svoj cieľ a pomôcť ostatným agentom.

Softvérové agenty sú zvlášť vhodné pre aplikácie, kde údaje, riadenie a využívanie zdrojov sú distribuované a agenty poskytujú prirodzenú metaforu pre reprezentáciu funkcionality systému.

Typológia agentov

Agenty je možné deliť podľa rôznych kritérií; medzi najčastejšie uvádzané typy agentov patria kolaboratívne, mobilné, reaktívne, informačné, inteligentné a hybridné agenty. Jednou typológiou agentov je ich rozdelenie podľa spolupráce, učenlivosti a autonómnosti (Obrázok 3-20).



Obrázok 3-20.Časť typológie agentov podľa (Nwana, 1999).

Agenty typu Belief-Desire-Intention

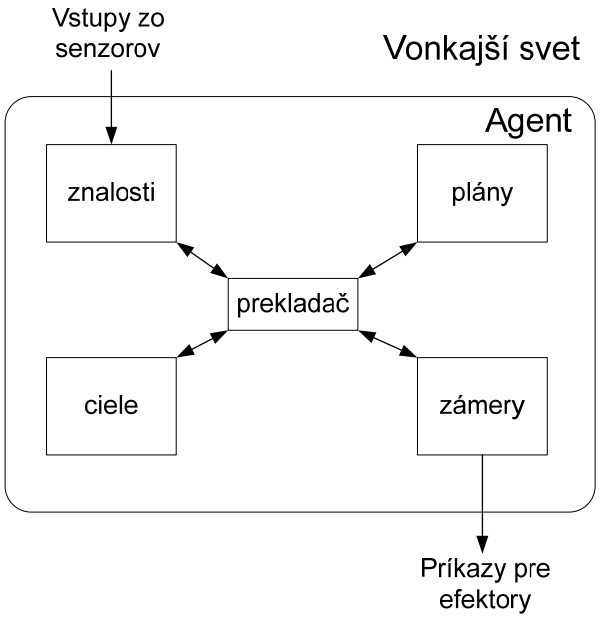
Belief-Desire-Intention (BDI) agent vychádza z modelu vytvoreného na základe ľudského odvodzovania (Bratman, 1999). Architektúru agenta tvoria znalosti (domnienky), ciele (želania), zámery a interakcie:

- Znalosti reprezentujú agentov model sveta. Tieto vedomosti sú subjektívne, nemusia byť ani úplné, ani správne.
- Ciele sú vyjadrené stavmi (situáciami), do ktorých by sa agent rád dostal. Podľa (Schmotzer, 2001), pri type agenta BDI je dôležité, že agent nemusí byť schopný splniť všetky ciele, dokonca aj keď splniteľné sú. Podmnožina cieľov, ktoré sa agent podujme splniť sa nazýva zámery.
- Zámery sú situácie a plány, ktoré sa agent aktuálne snaží splniť.

Obrázok 3-21 zobrazuje schému BDI agenta a jeho komunikáciu s prostredím (Rao, 1995).

3.7.2 Agentovo-orientované programovanie

Pomocou agentovo-orientovaného programovania možno rozšíriť objektovú paradigmu na výpočtový systém, ktorý sa skladá z komunikujúcich modulov, každý s vlastným spôsobom spracovania správ (Shoham, 1993). Porovnania agentovo-orientovaného programovania a objektovej paradigmy je v tabuľke 3-2.



Obrázok 3-21. Schéme BDI agenta.

Tabuľka 3-2. Porovnanie objektovo-orientovaného prístupu a agentovo-orientovaného programovania.

	Objektová paradigma	**Agentová paradigma**
základný typ	objekt	agent
komunikácia	metódy	správy
výmena informácií	akákoľvek	typy správ - informačné, požiadavky, ponuky, zamietnutia
obmedzenia metód	žiadne	konzistentnosť

3.7.3 Agentové technológie – JACK Intelligent Agents

Jedným z komerčne dostupných prostredí pre vývoj a aplikovanie agentov je technológia JACK Intelligent Agents [6], samotní tvorcovia hovoria o agentovo-orientovanom vývojovom prostredí, ktoré predstavuje nadstavbu nad jazykom Java.

Typy v jazyku JACK Agent Language:

- agent – predstavuje agenta, jeho správanie, typy správ, schopnosti, udalosti ktoré spracúva a plány, ktoré vykonáva v snahe dosiahnuť cieľ,
- plán – určuje správanie (logiku) agenta,
- udalosť – zvyčajne správa, ktorú agent získa a spracuje,
- databáza – báza znalostí agenta (privátna, globálna, zdieľaná),

[6] JACK Intelligent Agents, http://www.agent-software.com/

- schopnosť – zoskupenie viacerých funkcionálnych komponentov (plán, udalosť, databáza).

Vzorový kód agenta znázorňuje príklad 3-6.

```
agent AgentType extends Agent
{
  //báza znalostí agenta
  #private databaseDbType db_name(arg_list);
  #private data DataType data_name(arg_list);

  //získavané a spracovávané udalosti
  #handles event EventType;
  #posts event EventType reference;
  #sends event EventType reference;

  //plány agenta
  #uses plan PlanType;

  //schopnosti agenta
  #has capability CapabilityType reference;
}
```

Príklad 3-6. Príklad kódu agenta v prostredí JACK Intelligent Agents.

Alternatívnym open-source vývojovým prostredím agentov je Jadex[7], ktorý využíva jazyk Java a XML.

3.7.4 Agenty a vlastnosti systému

Pri požiadavke vytvárať softvér využívajúci softvérové agenty je rozumné uvažovať o vplyve tohto prístupu na viaceré vlastnosti systému. Bližšie rozoberieme tri hlavné vlastnosti typické pre softvérové agenty – súperenie/zdieľanie zdrojov, škálovateľnosť systému vzhľadom na množstvo agentov a ich komunikáciu a mobilita agentov.

Zdieľanie zdrojov a konkurencia

Agentové systémy pristupujú k spoločným zdrojom, čo si vyžaduje správne radenie udalostí a adekvátne zabezpečenie zdieľaných prostriedkov. Agenty, ktoré sú nezávislé a kolaboratívne zároveň si konkurujú a treba zabezpečiť explicitnú synchronizáciu udalostí agentov. Napr. v prostredí JACK Intelligent Agents možno použiť príkaz `@wait`.

Škálovateľnosť

Disciplinované vytváranie nových agentov (teda nových procesov) pomáha vyvarovať sa nečakanému zníženiu výkonu systému. Je tiež potrebné uvažovať nad vhodným riešením komunikácie medzi agentmi pomocou správ. Ak napríklad správy prechádzajú cez jeden centrálny bod, hrozí jeho zahltenie.

V širších súvislostiach je vhodné mať na pamäti, čo môže spôsobiť používanie agentov vôbec. Čo ak by každý človek pri počítači s konektivitou používal agenty, ktoré prehľadávajú web?

[7] Jadex, http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

Migrácia agentov

Presunom na vzdialenejší počítač agent jednoduchšie (rýchlejšie) dosiahne zdroje cieľového počítača, tieto výhody však musia prevážiť náklady na presun agenta. Existuje viacero druhom migrácie agentov, napríklad tzv. „ťažká“ migrácia, kedy sa presúva stav agenta a aj jeho báza znalostí. Problém predstavuje možná vyťaženosť cieľového počítača a tiež otázka bezpečnosti údajov – jednak údajov agenta a tiež údajov v cieľovom počítači.

Jedna z technológií podporujúca migráciu agentov medzi počítačmi je open-source platforma Aglets[8] od IBM. Príklad 3-7 ukazuje časť kódu agenta v prostredí Aglets, ktorý sa z domovského počítača presunie na hostiteľský, kde vypíše správu a vráti sa naspäť (príklad je zjednodušený o problém rozhrania výpisu).

```
public class HelloAglet extends Aglet {

    // inicializácia
    public void onCreation(Object init) {
           setMessage("Hello World!");
           home = getAgletContext().getHostingURL().toString();
    }

    // presun na vzdialené miesto
    public synchronized void startTrip(Message msg) {
           // cieľová adresa
           String destination = (String)msg.getArg();

           // presun na zadané miesto a inicializácia výpisu
           itinerary.go(destination, "sayHello");
    }

    // samotný výpis "sayHello" u hostiteľa a návrat domov
    public void sayHello(Message msg) {
           setText(msg);// výpis "sayHello"

           // pokus agenta o návrat domov
           setText("I'll go back to.. " + home);

           // návrat domov a inicializácia výpisu, že už je doma
           itinerary.go(home, "atHome");
    }

    //výpis, že agent je už doma a zrušenie agenta
    public void atHome(Message msg) {
           setText("I'm back.");
           dispose();
    }
}
```

Príklad 3-7. Agent v prostredí Aglets, presúvajúci sa medzi domovským a hostiteľským počítačom.

[8] IBM Aglets, http://www.trl.ibm.com/aglets/

3.7.5 Zhrnutie

Použitie agentových technológií predstavuje ďalší stupeň abstrakcie pri vytváraní softvéru. Agenty ako spolupracujúce, distribuované a aktívne zasahujúce objekty sú vhodné v prípadoch, keď je od vyvíjaného softvéru požadované prispôsobivé správanie v zložitých distribuovaných prostrediach.

Zložitosť sveta, v ktorom agenty existujú a konajú môže spôsobiť problém pri správnom rozhodovaní o proaktívnosti/reaktívnosti agenta – ak sa agent správa príliš proaktívne, môže ho zamestnať plnenie bezvýznamných cieľov. Je preto dôležité určiť, ktoré ciele má agent plniť a za akých okolností (Jennings, 2000). Keďže agenty predstavujú pomerne vysokú abstrakciu, je ťažké predpovedať ich správanie v budúcnosti vzhľadom na ich zložité rozhodovacie mechanizmy a spôsoby interakcie.

Predstavené riešenia sú náznakom, kam sa môže vývoj softvéru uberať v budúcnosti, keďže metafora agentov je stále považovaná za vyvíjajúcu sa a prepojenie softvérového inžinierstva a agentových systémov je len v začiatočnej fáze vývoja. Jedným z príkladom vznikajúcej podpory softvérových agentov v skorších fázach vývoja softvéru je rozšírenie jazyka UML nazývané Agent UML (Bauer, 2001).

Použitá literatúra

ALONSO, G. – CASATI, F. – KUNO, H. – MACHIRAJU, V. *Web Services Concepts, Architectures and Applications*. Springer-Verlag 2004.

BANIASSAD, E. – CLARKE, S. (2004) Theme: *An Aproach for Aspect-Oriented Analysis and Design*. Trinity College Technical Report TCD-CS-2003-40.

BAUER, B. – MÜLLER, J. P. – ODELL J. (2001). *Agent UML: A Formalism for Specifying Multiagent Interaction*. Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds., Springer-Verlag, Berlin, pp. 91-103, 2001.

BERNERS-LEE, T. – HENDLER, J. – LASSILA, O. (2001) The Semantic Web. In: *Scientific American*, Vol. 284, No. 5, pp. 34-43.

BRATMAN, M. E. (1999). *Intention, Plans, and Practical Reason*. CSLI Publications. ISBN 1-57586-192-5.

BRUSCHI, D. – DE WIN, B. – MONGA, M. (2006) *Introduction to Software Engineering for Secure Systems: SESS06 – Secure by Design*. Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems, p.1-2, May 20-21. Shanghai, China.

COOK, S. (2004). *Domain-Specific Modeling and Model Driven Architecture*. MDA Journal.

CHATTERJEE, S. – WEBER, J. Developing Enterprise Web *Services: An Architect's Guide*. Prentice-Hall, 2004.

FOWLER, M. (2004). *Model Driven Architecture*. http://www.martinfowler.com/bliki/ModelDrivenArchitecture.html

HANENBERG S. – OBERSCHULTE C. – UNLAND R. (2003) *Refactoring of Aspect-Oriented Software*. In 4th Int. Conf. on Object-Oriented and Internetbased Technologies, Concepts, and Applications for a NetworkedWorld.

JACOBSON, I. – NG, P.W. (2004) *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley.

JENNINGS, N. – WOOLDRIDGE, M. (2000). *Agent-Oriented Software Engineering*. In Handbook of Agent Technology (ed. J. Bradshaw) AAAI/MIT Press.

KICZALES G. – HILSDALE E. – HUGUNIN J. – KERSTEN M. – PALM J. – GRISWOLD W. (2001) *An Overview of AspectJ*. In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18–22 June.

NOVÁK, P. (2006). *Programovacie jazyky pre vývoj inteligentných agentov*. Seminár z umelej inteligencie, Computational Intelligence Group, Clausthal University of Technology, Nemecko.

NWANA, H. S. (1999). *Software Agents: An Overview*. Knowledge Engineering Review, Vol. 11, No 3, pp.1-40.

RAO, A. – GEORGEFF, M. (1995). *BDI Agents from Theory to Practice*. Technical Note 56, AAII, April.

SCHMOTZER, M. (2001). *dMARS*. Interná výskumná správa, Univerzita Pavla Jozefa Šafárika v Košiciach, Prírodovedecká fakulta.

SHOHAM, Y. (1993). *Agent-oriented programming*. Artificial Intelligence 60, 1 (Mar.), 51-92.

STEIN, D. – HANENBERG, S. – UNLAND, R. (2004) *Modeling Pointcuts*. Early Aspects 2004: Workshop on Aspect-Oriented Requirements Engineering and Architecture Design, AOSD 2004, Lancaster, UK, March 22.

STUDER, R. – BENJAMINS, R. –FENSEL, D. (1998) Knowledge Engineering: Principles and Methods. In *Data Knowledge Engineering*, Vol. 25, No. 1-2, pp. 161-197.

THOMAS, D. (2003). *UML – Unified or Universal Modeling Language?,* In: *Journal of Object Technology*, Vol. 2, No. 1, pp. 7-12.

ZIMMERMANN, O. – TOMLISON, M. R. – PEUSER, S. *Perspectives on Web Services Applying SOAP, WSDL and UDDI to Real World-Projects*. Springer-Verlag 2004.

DIEL II:
VYBRANÉ TÉMY PROGRAMOVÝCH A INFORMAČNÝCH SYSTÉMOV

4 VÝSKUM VLASTNOSTÍ SÚBOROVÝCH SYSTÉMOV

Narastajúcou informatizáciou spoločnosti a rýchlym nárastom kapacít diskových médií vzniká množstvo problémov, ktoré pred niekoľkými rokmi ešte neexistovali. Jedným z hlavných problémov súčasnosti je, že je k dispozícii veľmi veľké množstvo dát, z čoho užitočné informácie tvoria len nepatrnú časť. Problém s veľkým množstvom dát sa prejavuje ako v lokálnom rozmere (osobný počítač používateľa) tak ako v globálnom rozmere (vyhľadávacie služby, klastre). Tieto dáta je potrebné zálohovať, je potrebné v nich vyhľadávať, v niektorých prípadoch uchovávať ich verzie, triediť podľa aktuálnych požiadaviek, prípadne nad nimi vykonávať iné požadované operácie. Na ukladanie dát sa na logickej úrovni vo všeobecnosti využívajú súborové systémy (prípadne databázové systémy). V súčasnosti existuje veľké množstvo súborových systémov, pričom každý z nich má svoje špecifiká (rýchlosť prístupu, spoľahlivosť, kapacita) charakteristické pre dobu, keď boli vyvinuté a taktiež vzhľadom k účelu ich vzniku. Charakteristickou črtou týchto systémov je, že dáta, ktoré spravujú, v nich vystupujú v pasívnej forme – sú ukladané na fyzické médiá s rôznou úrovňou optimálnosti a spoľahlivosti.

Pohľad na dáta ako na pasívne entity prináša okrem rôznych výhod, ako jednoduchá manipulácia, jednotné rozhranie alebo nízke výpočtové nároky, aj rôzne nedostatky. Azda hlavným nedostatkom súčasných (pasívnych) súborových systémov je, že dáta sú „bezmocné“ voči vonkajším vplyvom akými sú napríklad: poškodenie média, nechcené zmazanie dôležitých údajov a podobne. Ďalšou nevýhodou je, že sa súčasné súborové systémy obmedzujú len na poskytnutie možností nastavenia rôznych práv a atribútov pre adresáre a súbory (súbor len na čítanie, systémový súbor, ...), ktoré však nezahŕňajú vlastnosti resp. atribúty ako „súbor, ktorý sa vysokou pravdepodobnosťou nestratí ani pri poškodení časti disku“, „súbor, ktorý sa bude sám zálohovať (bez ďalšej interakcie používateľa)“ a podobne.

Táto práca sa pokúsi priniesť nový pohľad na súbory, presnejšie - pokúsi sa nahradiť pasívne súbory aktívnymi, t.j. takými, ktoré môžu sami iniciovať operácie potrebné pre plnenie určitých tried úloh, ktoré sú na ne kladené.

Prvá časť práce sa zaoberá základným konceptom súčasných súborových systémov. Zameriava sa hlavne na súborové systémy, ktoré sú z hľadiska problematiky tejto práce zaujímavé, prípadne, ktoré sú príbuzné koncepcii nového pohľadu na súborový systém navrhnutom v závere práce spoločne s cieľmi ďalšieho výskumu.

4.1 Súborový systém

Súborový systém je podsystém operačného systému, ktorého cieľom je poskytovať perzistentné úložisko dát (Levy & Silberschatz, 1990).

Súborový systém by mal poskytovať dva odlišné komponenty (Tanenbaum, 2001):

- súborové služby
- adresárové služby

Prvý komponent sa týka operácií nad jednotlivými súbormi, akými sú: čítanie, zápis alebo pripájanie obsahu k súborom, zatiaľ čo druhý sa týka vytvárania a manažovania adresárov, pridávania a rušenia súborov z adresárov a podobne. V nasledujúcich dvoch podkapitolách budú opísané rozhrania súborových a adresárových služieb.

4.1.1 Súborové služby

Pre každú súborovú službu je základnou otázkou: „čo je súbor?“. Existuje viacero definícií súborov, pričom jedna z nich je nasledujúca: Súbory predstavujú abstraktný mechanizmus, ktorý poskytuje spôsob na ukladanie informácií na disk a ich neskoršie čítanie (Silberschatz & Galvin, 1998). Vo väčšine operačných systémov ako napríklad UNIX a MS-DOS je súbor neinterpretovaná postupnosť slabík. Význam a štruktúra informácií v súboroch je úplne ponechaná na aplikačné programy, pre operačný systém je neznáma.

V starších (mainframe) systémoch existuje veľa rôznych typov súborov, pričom každý typ má rôzne vlastnosti. Súbor môže byť koncipovaný napríklad ako sekvencia záznamov (podobne ako v operačných systémoch OS/400 a IBM i5/OS (Scholerman et al., 1993)), pričom operačný systém tieto záznamy číta a zapisuje. Záznam je zvyčajne špecifikovaný svojim číslom záznamu (pozícia v súbore) alebo hodnotou niektorej z položiek záznamu. Operačný systém môže udržiavať súbory tiež ako binárne stromy alebo iné vhodné údajové štruktúry, prípadne používať hešovacie tabuľky na rýchle nájdenie záznamu. Väčšina súčasných súborových systémov uprednostňuje ponímanie súborov ako sekvenciu slabík pred sekvenciou záznamov s kľúčmi.

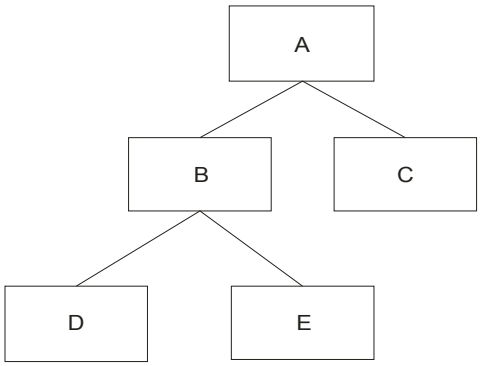
Súbory môžu mať atribúty, ktoré predstavujú časti informácií o nich, avšak netvoria ich súčasť. Typickými atribútmi sú napríklad vlastník, veľkosť súboru, dátum vytvorenia a prístupové práva. Súborové služby poskytujú primitívy na čítanie a zápis týchto atribútov. Umožňujú napríklad zmeniť prístupové práva súboru, nie však zmeniť jeho veľkosť (bez zmeny obsahu). Niektoré novšie systémy poskytujú okrem štandardných atribútov aj operácie na vytváranie a manipuláciu s prístupovými zoznamami – ACL (Grünbacher, 2003) ako aj s používateľmi definovanými atribútmi. Tieto prístupové zoznamy spolu s používateľmi definovanými atribútmi podstatne rozširujú bezpečnosť systému a jeho možnosti využitia.

V operačných systémoch ako napríklad UNIX a Linux existujú aj iné, ako len údajové (regulárne) súbory. V týchto systémoch predstavujú niektoré druhy súborov abstrakciu nad zariadeniami počítača. Pravdepodobne najznámejším takýmto súborovým systémom je súborový systém DevFS (Kroah-Hartman, 2003). Okrem reprezentácie blokových a znakových zariadení tu môžu súbory reprezentovať aj pomenované dátovody alebo sokety.

4.1.2 Adresárové služby

Ďalšou časťou súborových služieb sú adresárové služby, ktoré poskytujú operácie na vytvorenie a rušenie adresárov, operácie pomenovania a premenovania súborov a adresárov, a ich presun z jedného adresára do iného. Adresárové služby definujú istú abecedu a syntax pre vytváranie mien súborov a adresárov (Tanenbaum, 2001). Mená súborov sa môžu typicky skladať z jedna a viac (vždy existuje horné ohraničenie) písmen, čísel a vopred definovaných špeciálnych znakov. Niektoré systémy oddeľujú mená súborov na dve časti, zvyčajne oddelenými bodkou, ako napríklad *program.c* pre zdrojový súbor jazyka C alebo *udaje.txt* pre textový súbor. Druhá časť mena súboru zvaná *prípona súboru* identifikuje typ súboru. Iné systémy používajú na tento účel explicitné atribúty namiesto pripájania prípon k menám súborov.

Adresár obsahuje bežne množinu súborov pre jeden projekt, ako napríklad rozsiahly program alebo dokument. Keď je obsah (pod)adresára vylistovaný, sú zobrazené len relevantné súbory, ostatné súbory v ďalších podadresároch sa nezobrazujú. Podadresáre môžu obsahovať svoje vlastné podadresáre, čím sa zavádza strom adresárov, často nazývaný *hierarchický súborový systém* (Silberschatz & Galvin, 1998). Obrázok 4-1. znázorňuje príklad obsahu hierarchického súborového systému – strom s piatimi adresármi.



Obrázok 4-1. Hierarchický súborový systém s piatimi adresármi.

V niektorých súborových systémoch je možné vytvárať odkazy alebo ukazovatele na ľubovoľný adresár. Tieto môžu byť uložené v ktoromkoľvek adresári, čo umožňuje vytváranie ľubovoľných adresárových grafov (cyklických aj acyklických), ktoré sa vo všeobecnosti považujú za výkonnejšie v porovnaní zo stromovou štruktúrou.

4.1.3 Pomenovanie súborov a adresárov

Väčšina súborových systémov používa nejakú formu dvojúrovňového pomenovania súborov a adresárov. Súbory (a ostatné objekty) majú symbolické mená, ako napríklad *program.c* (niektoré operačné systémy priamo vynucujú zloženie mena z dvoch častí) pre jednoduché použitie používateľmi, majú však tiež interné – binárne mená pre používanie systémom samotným. Adresáre v skutočnosti prevádzajú mapovanie medzi týmito dvomi úrovňami pomenovania (Tanenbaum, 2001, Levy & Silberschatz, 1991). Pre používateľov a programy je pohodlné používanie symbolických (ASCII) mien, pre použitie systémom samotným sú však tieto mená príliš dlhé a ťažkopádne. V prípade, že používateľ otvorí súbor alebo sa naň inak odkazuje použitím

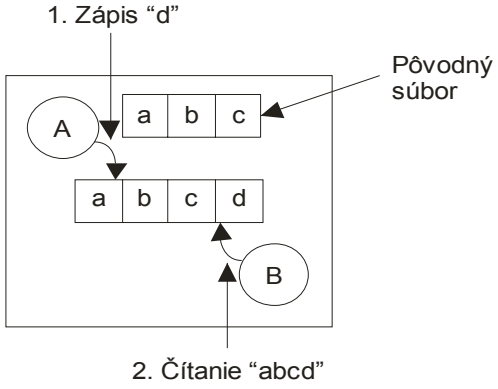
symbolického mena, systém okamžite vyhľadá dané symbolické meno v adresárovej štruktúre z ktorej potom získa binárne meno, ktoré použije na prístup k súboru. V niektorých systémoch sú binárne mená viditeľné aj pre používateľov (UNIX, Linux, MacOS X), nie je to však pravidlom (Windows, MS-DOS).

4.1.4 Prístup k súborom

Zo začiatku poskytovali operačné systémy len jediný druh prístupu k súborom – *sekvenčný prístup*. V týchto systémoch mohol proces čítať slabiky alebo záznamy súboru od začiatku po koniec, nemohol však vynechať nejakú časť a prekročiť tak poradie čítania. Tento prístup sa často používal v prípadoch, keď bola úložným médiom magnetická páska (Prabhakar et al., 1997). Príchodom diskov ako médií na ukladanie dát sa umožnilo čítanie obsahu súborov v ľubovoľnom poradí. Tento prístup sa nazýva *náhodný prístup*. Pri náhodnom prístupe sa pri operácii *Read* buď vždy uvedie pozícia miesta v súbore, z ktorého sa má čítať, alebo samotná operácia *Read* číta sekvenčne a používa sa operácia posunu v súbore, kedy ďalšia operácia *Read* číta z miesta, kde bol vykonaný posledný posun prostredníctvom operácie *Seek*.

4.1.5 Sémantika zdieľania súborov

Pokiaľ dvaja alebo viacerí používatelia zdieľajú rovnaký súbor, je pre predídenie možných problémov a konfliktov nutné precízne definovať sémantiku čítania a zápisu. V jedno-procesorových systémoch, ktoré povoľujú procesom zdieľanie súborov (UNIX, Linux, Windows) táto sémantika bežne stanovuje, že pokiaľ operácia *Read* nasleduje po operácii *Write*, tak *Read* vráti hodnotu zapísanú *Write* – tak ako je to znázornené na obrázku 4-2. Podobne, keď dve operácie *Write*, v krátkom časovom okamžiku po sebe, sú nasledované operáciou *Read*, tak hodnota po prečítaní bude identická s poslednou zapísanou hodnotou (posledný *Write*). V skutku systém vynucuje absolútne usporiadanie času pre všetky operácie a vždy vracia najčerstvejšiu hodnotu. Tento model je často označovaný ako *UNIX sémantika* (Silberschatz & Galvin, 1998). Je veľmi jednoducho pochopiteľný a priamočiaro implementovateľný.



Obrázok 4-2. UNIX sémantika zdieľania súborov.

4.1.6 Diskové súborové systémy

Väčšinu súborových systémov s ktorými sa bežne stretávame možno zaradiť medzi diskové súborové systémy. Sú to „jednoduché“ súborové systémy navrhnuté na ukladanie súborov na dátové úložisko, ktoré je reprezentované fyzickým médiom

ako napríklad: disky, CD/DVD nosiče, diskety a podobne. Medzi najznámejšie a najpoužívanejšie diskové súborové systémy patria FAT, NTFS, HFS, HFS+, Ext2, Ext3, ReiserFS, ISO 9660. Niektoré diskové súborové systémy ponúkajú rôzne rozšírenia, ako napríklad ukladanie záznamov do denníka na špeciálne miesto na disku (alebo do súboru), ktoré zvyšuje odolnosť proti poškodeniu súborového systému v prípade nečakaných udalostí. Tomuto denníku sa často hovorí žurnál (žurnálový súbor). Ďalším častým rozšírením je použitie verziovania súborov v súborovom systéme.

Použitie mechanizmu žurnálovania

Od súborových systémov sa očakáva extrémna spoľahlivosť a súčasne čo najväčšia možná rýchlosť. Poruchy na počítačoch sa objavujú pomerne často a nečakane, a to ako hardvérové, tak aj softvérové a z tohto môžu vznikať rôzne problémy. Po nečakanom reštarte počítača môže zabrať obnovenie konzistentného stavu súborového systému pomerne dlhý čas. Pri súčasných veľkostiach súborových systémov môže byť serióznym problémom, že obnovenie súborového systému do konzistentného stavu trvá aj niekoľko hodín. Nie vždy si však môžeme takýto – niekoľko hodinový – výpadok dovoliť. Napriek tomu, že rýchlosti diskov rastú každým rokom je toto zrýchlenie zanedbateľné v porovnaní s nárastom ich kapacity za rovnaké časové obdobie. Pri tradičných technikách opravy súborových systémov trvá pri zvýšení ich kapacity na dvojnásobok aj oprava súborového systému približne dvojnásobný čas. Pri riešení problému obnovy obsahu súborového systému treba brať ohľad na rôzne aspekty (Tweedie, 1998), akými sú:

Udržiavanie: dáta, ktoré sú uložené na disku pred poruchou systému, by nemali byť poškodené. Nepoškodenie súborov, ktoré sa zapisovali v čase poruchy zrejme zaručiť nie je možné, je však nutné, zabezpečiť to, aby pri procese obnovy súborového systému zostali nedotknuté súbory, ktoré už boli bezpečne uložené.

Predikovateľnosť: spôsoby poškodenia z ktorých sa systém musí zotaviť by mali byť predpovedateľné, aby sa zaručilo spoľahlivé obnovenie súborového systému.

Atomickosť: množstvo operácií vykonávaných na súborovom systéme vyžaduje významný počet oddelených vstupno-výstupných operácií (príkladom môže byť presun súboru z jedného adresára do iného). Obnovenie sa považuje za atomické pokiaľ sa operácie na súborovom systéme buď úplne ukončia, alebo je ich možné po ukončení obnovy súborového systému plne vrátiť do počiatočného stavu.

Jednotlivé implementácie mechanizmu žurnálovania na súborovom systéme využívajú jeden alebo viacero z týchto aspektov na dosiahnutie bezpečného a rýchleho obnovenia súborového systému po jeho neočakávanom poškodení.

Použitie mechanizmu verziovania

Prvý súborový systém podporujúci udržiavanie viacerých súčasných verzií súborov bol prvý krát opísaný v operačnom systéme TENEX (Murphy, 1970). Za hlavný prínos tohto súborového systému sa však považoval manažment symbolických mien súborov a zavedenie prístupových práv (Bobrow et al., 1972). Symbolické mená súborov sa v tomto operačnom systéme mohli skladať z piatich častí, ktoré vlastne reprezentovali strom s maximálnou hĺbkou päť. Týchto päť častí reprezentovalo zariadenie, meno adresára, meno súboru, príponu súboru a číslo verzie súboru. Po každom zápise do už existujúceho súboru na súborovom systéme vznikla nová verzia tohto súboru, pričom staršie verzie mohol odstrániť používateľ alebo ich mohol odstraňovať samotný systém.

V súčasnosti je dostupných viacero súborových systémov podporujúcich udržiavanie súborov spoločne s ich verziami, sú to napríklad: Ext3cow (Peterson & Burns, 2005), PersiFS (Ports et al., 2005) a WinFS. Princípom týchto súborových systémov je vytvorenie kópie súboru, ku ktorému sa pristupuje na zápis (prípadne vytvorenie kópie v pravidelných časových intervaloch). Keďže v prípade vykonania veľmi malých zmien v súboroch vzniká pri veľkom množstve prístupov na zápis problém neefektívneho využívania diskového priestoru, všetky tieto súborové systémy zabezpečujú mechanizmy, ktoré sa snažia takýmto situáciám predchádzať.

Takýmito mechanizmami sú:

- obmedzenie maximálneho počtu verzií súborov, ktoré sa budú uchovávať
- obmedzenie vytvárania nových verzií podľa času (nie je možné vytvoriť novú verziu častejšie ako *N*)

Súborový systém Reiser 4

Súborový systém Reiser 4[9] predstavuje zaujímavý koncept medzi diskovými súborovými systémami. V súčasnosti je všeobecne považovaný za najrýchlejší súborový systém (Benchmarks Of ReiserFS Version 4, 2006).

Medzi jeho významné vlastnosti patria:

- účinnejšie žurnálovanie prostredníctvom preskupujúcich sa logových záznamov
- rýchlejšia práca s adresármi obsahujúcimi veľké množstvá súborov
- účinnejšia podpora pre malé súbory a to, ako z pohľadu šetrenia diskového priestoru, tak aj z pohľadu rýchlosti
- dynamicky optimalizované rozvrhnutie disku prostredníctvom oneskorenej alokácie podobne ako v súborovom systéme XFS (Sweeney et al., 1996)
- podpora pre atomické transakcie

Najvýznamnejším (aj keď zatiaľ nie moc viditeľným) prínosom tohto súborového systému však je zavedenie infraštruktúry zásuvných modulov do súborového systému. Autori súborového systému sľubujú, že okrem súčasných základných zásuvných modulov (pre súbory, adresáre a pod.) bude v budúcnosti možné prostredníctvom týchto modulov zavádzanie špeciálnych typov metadát, šifrovanie, kompresia a množstvo ďalších funkcií. V súčasnosti sú k dispozícii len moduly, ktoré poskytujú funkcionalitu, ktorá sa vyskytovala aj v predchádzajúcej verzii tohto súborového systému – Reiser 3.

4.1.7 Databázové súborové systémy

Väčšina súčasných súborových systémov patrí medzi hierarchické súborové systémy, alebo aspoň hierarchický pohľad na svoje dáta poskytuje. Používanie týchto súborových systémov sa síce osvedčilo, avšak v súčasnosti, pri diskoch veľkostí stoviek gigabajtov, na ktorých sú ukladané milióny súborov sa stáva manažment týchto súborov čoraz obtiažnejší. Dôsledky a hlavne nevýhody používania hierarchických súborových systémov sa v súčasnosti už plne prejavujú. V súčasnosti sa využíva na vyhľadávanie súborov, prípadne ich obsahov čoraz viac procesorového času, ktorý by

[9] Reiser 4, http://www.namesys.com/v4/v4.html (last accessed in November 2006)

sa dal využiť oveľa efektívnejšie. Toto plytvanie procesorového času je aj dôsledkom používania súčasných hierarchických súborových systémov. Snahou databázových súborových systémov je odstrániť nevýhody hierarchických súborových systémov v čo možno najväčšej miere. Typickým príkladom databázového súborového systému je Database File System.

Súborový systém Database File System

Súborový systém Database File System (DBFS) je nový typ súborového systému, ktorý mení základnú koncepciu súborových systémov tak, ako ich pozná väčšina používateľov. V tomto súborovom systéme sa nenachádzajú adresáre a používateľ pri ukladaní súborov nevyberá žiadne miesto, kde by chcel súbor uložiť (Gorter, 2004, Dbfs, 2007). Namiesto názvu *súborový systém* by bolo v prípade DBFS vhodnejšie použiť výraz *dokumentový systém*. DBFS v skutočnosti nie je ani databázovým systémom, je to skôr taxonomický systém. Je to súborový systém vybavený na obsluhu používateľa a jeho cieľom je, čo možno najväčšie zjednodušenie jeho života.

DBFS používa namiesto adresárov kľúčové slová. Kľúčové slová pracujú podobne ako adresáre, avšak umožňujú oveľa viac. Kľúčové slová sú v ponímaní tohto súborového systému nadmnožinové adresáre.

Súčasná implementácia DBFS predstavuje klient/server architektúru (obrázok 4-3), ktorá poskytuje DBFS ako vrstvu nad hierarchicky založeným súborovým systémom. DBFS v súčasnosti neukladá súbory, namiesto toho udržuje referencie na súbory uložené v hierarchicky založenom súborovom systéme. Rozhranie k tomuto súborovému systému bolo implementované pre prostredie KDE[10] (K Destop Environment, 2007), kde toto rozhranie kompletne nahrádza hierarchický prístup k súborom. Stav implementácie DBFS je len čiastočný, avšak pre jeho pozitívne vlastnosti a najmä vďaka tomu, že bol vydaný pod licenciou GPL na ňom pracuje množstvo ľudí z Open Source komunity.

4.1.8 Distribuované (sieťové) súborové systémy

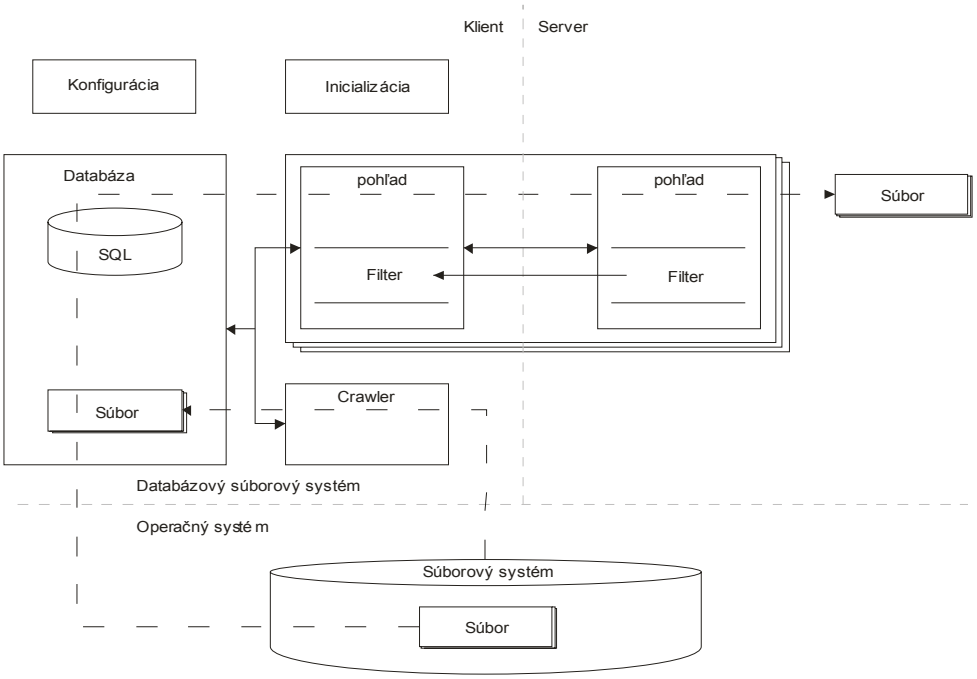
Ako v obyčajných systémoch, tak aj v distribuovaných systémoch je úlohou súborového systému ukladanie programov a dát, a udržiavanie ich dostupnosti pre neskoršie použitie. Množstvo aspektov distribuovaných súborových systémov (DFS) je podobných konvenčným súborovým systémom, preto týmto aspektom ďalej nebude venovaná pozornosť. Miesto toho budú uvedené tie aspekty DFS, ktoré sú odlišné od centralizovaných súborových systémov, pričom je nutné dodať, že z pohľadu používateľa by sa distribuovaný súborový systém mal (v ideálnom prípade) javiť tak, ako obyčajný centralizovaný súborový systém.

Základné vlastnosti distribuovaných súborových systémov

Medzi základné vlastnosti distribuovaných súborových systémov patria (Levy & Silberschatz, 1990):

Sieťová transparencia, táto vlastnosť umožňuje klientom pristupovať ku vzdialeným súborom použitím rovnakej množiny operácií ako pri prístupe k lokálnym súborom. Úlohou DFS je v tomto prípade lokalizovať súbory a zabezpečiť prenos ich dát.

[10] K Destop Environment – Conquer your Desktop!, http://www.kde.org/ (last accessed in January 2007)



Obrázok 4-3. Architektúra DBFS.

Mobilita používateľov je ďalším aspektom transparencie, ktorý zabezpečuje, aby používateľ nebol viazaný na konkrétny počítač ale sa mohol prihlásiť na ľubovoľnom počítači DFS. Transparentný DFS teda zabezpečí rovnaký prístup k dátam používateľa bez ohľadu na miesto jeho prihlásenia.

Výkon DFS, ktorý sa najčastejšie meria v čase potrebnom na vykonanie požadovaného dopytu môžeme považovať za ďalší rozmer transparencie. V konvenčných systémoch je tento čas rovný prístupovému času k disku a využitému procesorovému času pre tento prístup. V DFS musíme počítať aj s časom na doručenie požiadavky na server, ako aj časom za ktorý sa odpoveď na požiadavku vráti späť ku klientovi.

Odolnosť proti poruchám je chápaná v širšom slova zmysle, ako chyba komunikácie, chyba počítačov alebo chyba zariadení na ukladanie dát. Napriek tomu, že sa tieto chyby môžu zdať závažné, v DFS je možné ich v určitých prípadoch s časti tolerovať. Distribuované systémy odolné voči poruchám môžu pokračovať vo svojej práci ďalej aj navzdory týmto chybám, i keď v degradovanom stave.

Škálovateľnosť systému je chápaná ako schopnosť adaptácie systému na zvýšené zaťaženie. Škálovateľnosť je relatívna vlastnosť. Škálovateľný systém môže napríklad reagovať v porovnaní s neškálovateľným systémom v prípade vysokého zaťaženia elegantnejšie, degradácia výkonnosti systému môže prebiehať miernejšie alebo prostriedky v takomto systéme môžu dosiahnuť saturovaný stav neskôr ako v neškálovateľných systémoch. Napriek akokoľvek dobrému návrhu distribuovaného systému nie je možné poňať ustavične rastúce zaťaženie. Pridaním nových zdrojov je síce možné vyriešiť problém zaťaženia, avšak súčasne sa tým môže generovať ďalšie, nepriame zaťaženie, na iné zdroje (pridanie počítačov do distribuovaného systému môže zaťažiť sieťovú komunikáciu a zvýšiť zaťaženie služieb). Expandovanie systému

môže dokonca vyvolať potrebu zmeny návrhu DFS. Ideálny škálovateľný systém by mal poskytovať potenciál na svoje rozšírenie bez týchto problémov. V DFS je špeciálne dôležité poskytnúť možnosti jednoduchej škálovateľnosti, keďže rozšírenie siete pridaním nových počítačov alebo prepojenie viacerých sietí navzájom je veľmi časté. Odolnosť voči poruchám a škálovateľnosť sú si navzájom príbuzné zo všetkými ostatnými vlastnosťami DFS. Často zaťažený komponent sa napríklad môže stať paralyzovaným a môže sa správať ako chybný komponent. V takýchto prípadoch môže splniť cieľ presunutie záťaže z chybného komponentu na záložný komponent. Vo všeobecnosti je využívanie záložných prostriedkov dôležité pre dosiahnutie spoľahlivosti ako aj pre ošetrenie krátkodobých extrémnych zaťažení systému.

Služby v DFS

Súborové služby špecifikujú čo súborový systém poskytuje pre svojich klientov. Opisujú dostupné primitívi, ich parametre a akcie ktoré vykonávajú. Súborové služby pre klientov jednoznačne definujú s ktorými službami môžu klienti pracovať, ale nehovoria o tom, ako majú byť príslušné služby implementované. Súborové služby v podstate definujú rozhranie medzi súborovým systémom a klientom.

Súborový server je proces vykonávaný na niektorom alebo niektorých z počítačov, ktorý napomáha pri implementácii súborových služieb. Systém pri tom môže mať jeden alebo viacero súborových serverov. Súborové služby nemusia vedieť koľko súborových serverov je dostupných a aké sú ich geografické lokácie alebo funkcie každého z nich. Všetko čo potrebujú vedieť je, že pokiaľ zavolajú procedúru špecifikovanú v súborových službách, tak bude požadovaná akcia nejakým spôsobom vykonaná a bude vrátená požadovaná odpoveď. V skutočnosti by klienti nemali ani vedieť o tom, že sú súborové služby distribuované.

Klient je proces, ktorý využíva súborové služby prostredníctvom množiny operácií, ktoré vytvárajú rozhranie klienta. Rozhranie klienta pre súborové služby je tvorené množinou súborových operácií. Medzi tieto operácie patria napríklad vytvorenie a zrušenie súboru, čítanie a zápis do súboru a podobne.

Návrh distribuovaných súborových systémov

Rovnako ako pri diskových súborových systémoch, aj v distribuovaných súborových systémoch je nutné poskytovať aspoň základné – súborové a adresárové služby. V nasledujúcich podkapitolách budú uvedené tieto služby z pohľadu distribuovaných súborových systémov, pričom pre ne platia všetky vlastnosti súborových (kapitola 4.1.1) a adresárových (kapitola 4.1.2) služieb, ktoré už boli opísané v tejto práci.

Rozhranie súborových služieb

Dôležitým aspektom modelu súborových služieb je, či môžu byť súbory po ich prvotnom vytvorení aj modifikované. Bežne je táto modifikácia umožnená, ale v niektorých DFS tomu tak nie je; v týchto systémoch existujú len operácie *Creat* a *Read* (Tanenbaum, 2001). Po tom, čo sa v týchto systémoch súbor vytvorí už nemôže byť zmenený. O týchto súboroch sa hovorí, že sú takzvané *immutable* súbory. Majúc takéto súbory, je umožnené vytvoriť oveľa jednoduchšiu podporu pre využívanie lokálnych medzipamätí a replikáciu, keďže eliminujú všetky problémy súvisiace s aktualizáciou všetkých svojich kópií vždy, keď v nich nastane zmena.

Mechanizmy zabezpečenia v distribuovaných systémoch využívajú v podstate rovnaké techniky ako v jedno procesorových systémoch, presnejšie systém spôsobilostí

(capability) a prístupové zoznamy (ACL), prípadne sú doplnené o ďalšie bezpečnostné mechanizmy ako napríklad Kerberos a podobne.

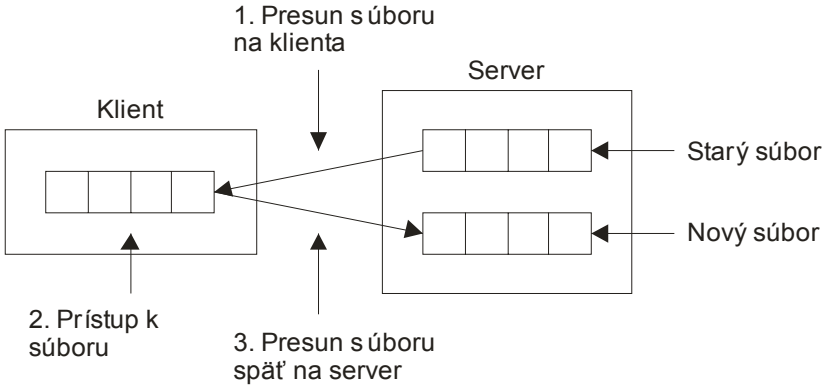
V prípade systému spôsobilostí vlastní každý používateľ určitý druh tiketu zvaného spôsobilosť pre každý objekt, ku ktorému má prístup. Spôsobilosť špecifikuje druh prístupov, ktoré sú na daný objekt povolené (napríklad povolený zápis ale nie čítanie a podobne).

Prístupové zoznamy asociujú s každým súborom zoznam používateľov, ktorý majú prístup k súboru súčasne aj s typom prístupu. Schéma UNIXu s kontrolou prístupových bitov na čítanie, zápis a vykonávanie pre každý súbor zvlášť pre vlastníka, vlastníkovu skupinu a ostatných je v podstate len zjednodušený prístupový zoznam.

Súborové služby môžeme deliť na dva typy v závislosti od toho, či podporujú takzvaný upload/download model alebo model vzdialeného prístupu. V *upload/download* modely zobrazenom na obrázku 4-4. poskytujú súborové služby len dve hlavné operácie (Tanenbaum, 2001, Levy & Silberschatz, 1990):

- čítanie zo súboru
- zápis do súboru

Prvá operácia prenáša celý súbor zo súborového servera na počítač klienta, ktorý oň požiadal. Druhá operácia prenáša súbor opačným smerom, teda od klienta na server. Tento konceptuálny model teda presúva súbor oboma smermi. Súbory môžu byť uložené v pamäti alebo na lokálnom disku podľa potreby.

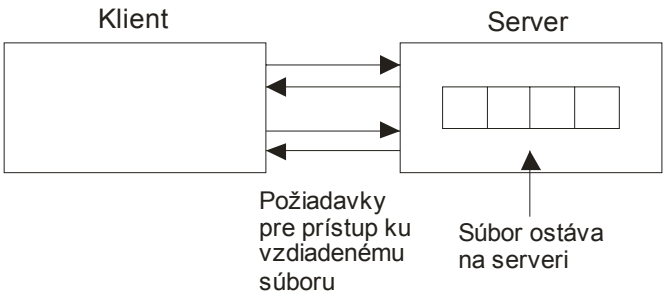


Obrázok 4-4. Upload/download model súborových služieb.

Výhodou upload/download modelu je jeho konceptuálna jednoduchosť. Aplikačné programy si sprístupnia súbory v prípade potreby a potom s nimi pracujú lokálne. Každý modifikovaný alebo novo vytvorený súbor je neskôr po ukončení programu zapísaný späť na server. Pri tomto modely nie je nutné vytvárať zložité rozhranie pre súborové služby. Okrem toho je transfer celých súborov veľmi účinný, avšak na strane klienta musí existovať dostatočná kapacita úložného priestoru na ukladanie požadovaných súborov. Nevýhodou tohto modelu je plytvanie v prípade prenosu celého súboru napriek tomu, že sa môže pracovať len s jeho malým fragmentom.

Iným druhom súborových služieb je *model vzdialeného prístupu* (Tanenbaum, 2001, Levy & Silberschatz, 1990), ktorý je znázornený na obrázku 4-5. Pri tomto modely

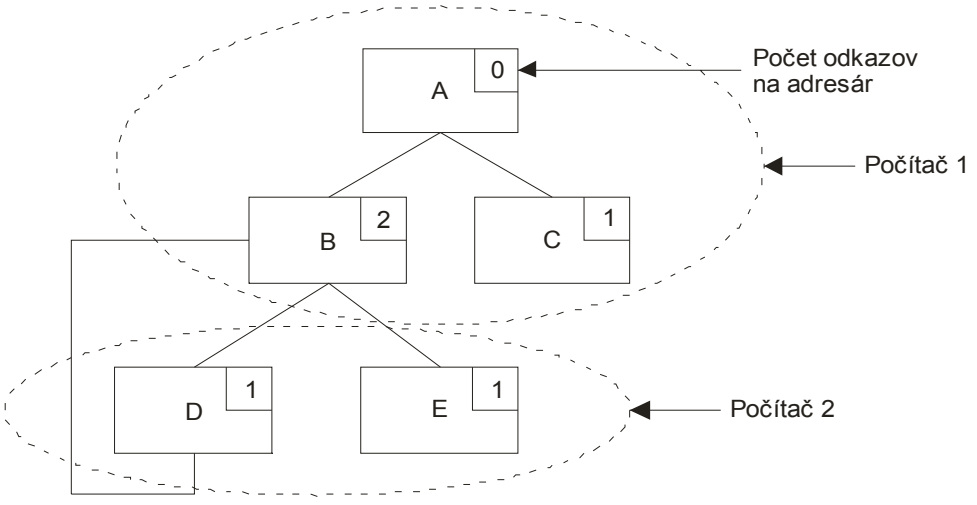
poskytujú súborové služby veľké množstvo operácií ako: otvorenie, zatvorenie súboru, čítanie a zápis častí súborov, posun v rámci súborov (*Seek*), zistenie a zmenu atribútov súboru a množstvo ďalších. Zatiaľ čo pri upload/download modely súborové služby poskytovali len fyzické úložisko a jednoduchý transfer súborov, pri tomto modely je súborový systém vykonávaný na strane servera a nie na strane klienta. Výhodou je, že nie je nutné mať k dispozícii veľké úložisko na strane klienta ako aj eliminácia problému presunu celého súboru v prípade, že je potrebné pracovať len s jeho malým fragmentom.



Obrázok 4-5. Model vzdialeného prístupu súborových služieb.

Rozhranie adresárových služieb

V prípade diskových súborových systémov nemá rozdiel v koncepte stromovej a grafovej reprezentácie súborov a adresárov veľký dosah na konzistenciu súborového systému. Rozdiel medzi stromovou a grafovou reprezentáciou je však obzvlášť závažný v distribuovaných systémoch. Podstata zložitosti tohto problému je znázornená na adresárovom grafe obrázku 4-6.



Obrázok 4-6. Distribuovaný adresárový graf na dvoch počítačoch.

Na tomto obrázku sa adresár *D* odkazuje na adresár *B*. Problém v prípade distribuovaných súborových systémov (na rozdiel od diskových) nastáva v prípade, keby bol zrušený odkaz z adresára *A* do adresára *B*. V stromovej hierarchii môže byť odkaz

na adresár odstránený len v prípade, ak je adresár, na ktorý daný odkaz odkazuje prázdny. V prípade grafovej reprezentácie je povolené zrušiť odkaz na adresár v prípade, že existuje aspoň jeden iný odkaz tento na adresár. Zavedením referenčného čísla znázorneného v pravom hornom rohu každého adresára – obrázok 4-3 je jednoduché určiť kedy je odstraňovaný odkaz na adresár posledný.

Po tom, ako sa zruší odkaz z *A* do *B* je referenčné číslo *B* znížené z 2 na 1, čo vyzerá ako jednoduché riešenie problému, avšak adresár *B* by bol v tomto prípade neprístupný z koreňového adresára súborového systému (*A*). Tri adresáre, *B*, *D* a *E* a všetky súbory v nich by sa stratili. Tento problém existuje aj v centralizovaných systémoch, je však kritickejší v distribuovaných. Pokiaľ je všetko na jednom počítači, je možné, aj keď potenciálne zložito, objaviť stratené adresáre, pretože sú všetky informácie o nich uložené na jednom mieste. Aktivita všetkých súborov môže byť v takýchto systémoch zastavená a graf môže byť obnovený počnúc koreňovým adresárom označením všetkých dostupných adresárov. Na konci tohto procesu sa všetky neoznačené adresáre považujú za nedostupné. V distribuovaných systémoch, kde je spojených viacero počítačov nie je možné zastaviť všetku aktivitu súborového systému, čím je získanie „snímky (snapshot)“ súborového systému veľmi zložité, ak nie nemožné.

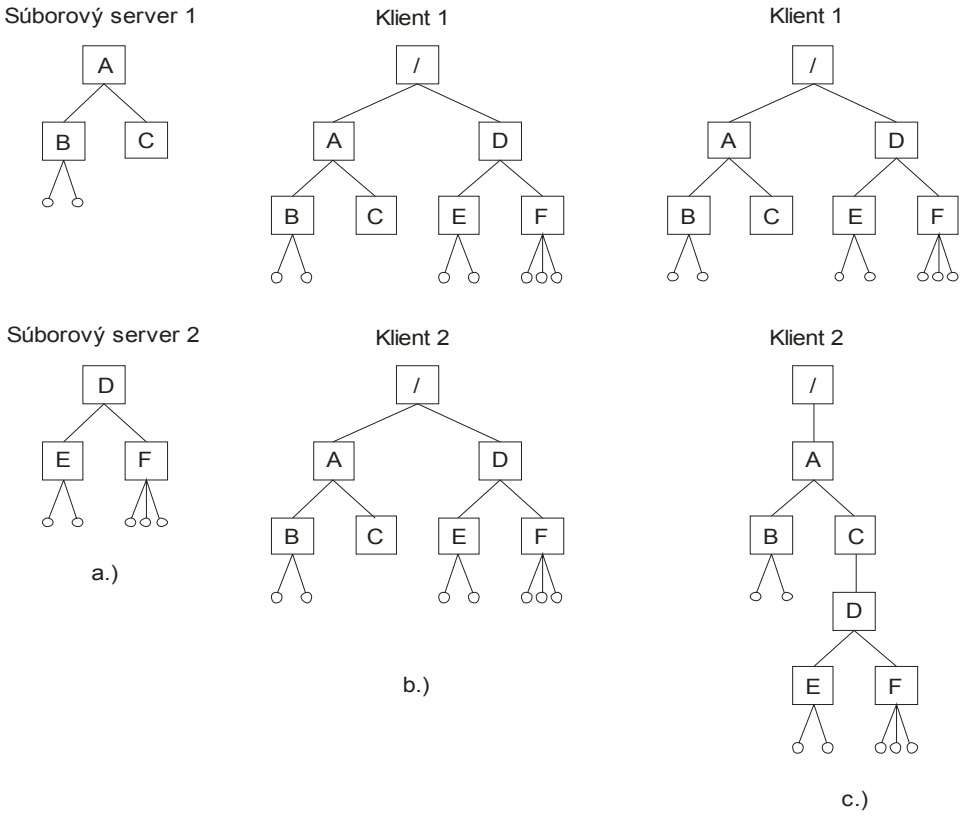
Kľúčovým problémom návrhu každého distribuovaného súborového systému je, či všetky počítače a procesy majú identický pohľad na adresárovú hierarchiu. Ako príklad na pochopenie tohto návrhového rozhodnutia poslúži obrázok 4-7. Na obrázku 4-7a sú znázornené dva súborové servery. Každý z nich obsahuje tri adresáre a niekoľko súborov. Na obrázku 4-7b je znázornený systém v ktorom má každý klient rovnaký pohľad na distribuovaný súborový systém. Pokiaľ cesta /D/E/x je platná na jednom počítači, je automaticky platná na všetkých ostatných počítačoch.

Na rozdiel od tohto pohľadu, je na obrázku 4-7c znázornená možnosť, keď má každý počítač v systéme rôzny pohľad na distribuovaný súborový systém. V porovnaní s predchádzajúcim príkladom môže byť cesta /D/E/x platná na jednom počítači, ale nemusí byť platná na ostatných. V systémoch, ktoré spravujú viacero súborových serverov pomocou vzdialeného pripájania je bežná práve situácia znázornená na obrázku 4-7c. Tento variant je síce flexibilne a priamočiaro implementovateľný, má však nevýhody, ktoré zamedzujú celému systému aby sa správal sa ako obyčajný – staromódny systém zdieľania času. V systémoch zdieľania času je pohľad na súborový systém pre všetky procesy rovnaký (ako na obrázku 4-7b). Táto vlastnosť uľahčuje jednoduchšie pochopenie systému a tiež jeho jednoduché programovanie.

Transparencia mien

V systéme zloženom z viacerých súborových serverov, kde je každý z nich samostatný (neobsahuje žiadne referencie na adresáre alebo súbory na iných súborových serveroch) môžme použiť ako binárne meno súboru číslo lokálneho i-uzla, ako je tomu napríklad aj v systémoch UNIX.

Viac všeobecná schéma pomenovania sa dosiahne v prípade, ak bude binárne meno udávať tak meno servera ako aj špecifický súbor na tomto serveri. Alternatívnym spôsobom na dosiahnutie rovnakého cieľa môže byť použitie *symbolických odkazov*. Symbolický odkaz je položka adresára ktorá sa mapuje na reťazec (server, meno súboru), ktorý môže byť použitý na vyhľadanie binárneho mena na serveri. Symbolické meno predstavuje samo o sebe cestu k súboru.



Obrázok 4-7. (a) Dva súborové serveri. Štvorce predstavujú adresáre, kruhy súbory. (b) Systém, v ktorom majú všetci klienti rovnaký pohľad na súborový systém. (c) Systém, v ktorom majú rôzny klienti rôzny pohľad na súborový systém

Ďalším spôsobom by mohlo byť použitie spôsobilostí (capability) ako binárnych mien súborov. Pri tejto metóde vyhľadanie ASCII mena vráti spôsobilosť, ktorá sa môže vyskytovať v rôznych tvaroch. Môže napríklad obsahovať číslo fyzického alebo logického počítača alebo sieťovú adresu príslušného servera, ako aj číslo ktoré identifikuje súbor, ktorý je požadovaný. Fyzická adresa môže byť použitá aj na odoslanie správy pre server bez bližšej interpretácie, logická adresa môže byť lokalizovaná napríklad broadcastingom alebo vyhľadaním na mennom serveri.

Poslednou možnosťou, ktorá sa vyskytuje v distribuovaných súborových systémoch, avšak len zriedkavo v centralizovaných je možnosť vyhľadania ASCII mena s tým, že návratovou hodnotou v tomto prípade nie je *jedno* ale *viacero* binárnych mien (i-uzlov, spôsobilostí prípadne niečoho iného). Tieto typicky reprezentujú originálny súbor a všetky jeho zálohy. Využitím viacerých binárnych mien je umožnené lokalizovať jeden z príslušných súborov aj v prípade, že požadovaný súbor nie je dostupný z rôznych dôvodov (použije sa ďalšie z binárnych mien). Táto metóda teda poskytuje určitú mieru chybovej tolerancie prostredníctvom redundancie.

Principiálnym problémom vyššie opísaných spôsobov pomenovania je, že nie je plne transparentné. V tomto kontexte sú podstatné dve formy transparencie. Prvou formou je *transparencia lokácie*, pri ktorej meno cesty nenaznačuje to, kde je súbor (alebo objekt) uložený. Cesta ako */server1/adresar1/x* napovedá explicitne každému, že sa

súbor *x* nachádza na serveri 1, ale nehovorí o tom, kde sa daný server nachádza. Server sa môže voľne pohybovať kdekoľvek po sieti bez toho, aby sa zmenila cesta k súborom, ktoré sa na ňom nachádzajú. Takýto systém je lokáciou transparentný. Príkladmi takýchto systémov môžu byť napríklad NFS, Locus a Sprite.

Transparencia lokácie však vždy nemusí postačovať. Predpokladajme, že je súbor *x* extrémne veľký a miesto na serveri 1 je už ďalej nepostačujúce. Tiež predpokladajme, že na serveri 2 je dostatočné voľné miesto. Distribuovaný systém by mohol presunúť súbor *x* na server 2 automaticky. Žiaľ pokiaľ je prvým komponentom mena každého súboru meno servera, systém nemôže súbor presunúť na iné miesto automaticky ani keby *adresar1* existoval na oboch serveroch. Problémom je, že automatický presun súborov mení cestu k nemu z */server1/adresar1/x* na */server2/adresar1/x*. Program, ktorý má pôvodnú cestu v sebe nakompilovanú alebo nastavenú prestane pracovať, keďže sa cesta zmenila. Systém v ktorom môžu byť súbory presunuté bez zmeny ich mien spĺňa podmienky *nezávislosti lokácie*. Súborový systém spĺňajúci túto požiadavku je napríklad Andrew File System (Wachsmann, 2005).

Distribuovaný systém, ktorý ukladá mená počítačov alebo serverov do cesty k súborom nie je nezávislý od lokácie. Túto vlastnosť taktiež nespĺňajú systémy založené na vzdialenom pripájaní, keďže nie je možné presunúť súbor z jednej skupiny súborov (zväzku pripojenia) do inej skupiny a súčasne dosiahnuť, aby bolo možné použiť staré meno súboru. Nezávislosť lokácie nie je jednoduché dosiahnuť, napriek tomu môže byť táto vlastnosť v distribuovaných systémoch žiaduca.

Sumarizácia vyššie opísaných možností hovorí o troch základných dostupných prístupoch pri pomenovaní súborov a adresárov v distribuovanom prostredí (Tanenbaum, 2001):

- Meno počítača + cesta k súboru, ako napríklad */pocitac1/cesta/k/suboru*
- Pripájanie vzdialených súborových systémov do lokálnej hierarchie súborového systému
- Jednotný menný priestor, ktorý je identický na všetkých počítačoch

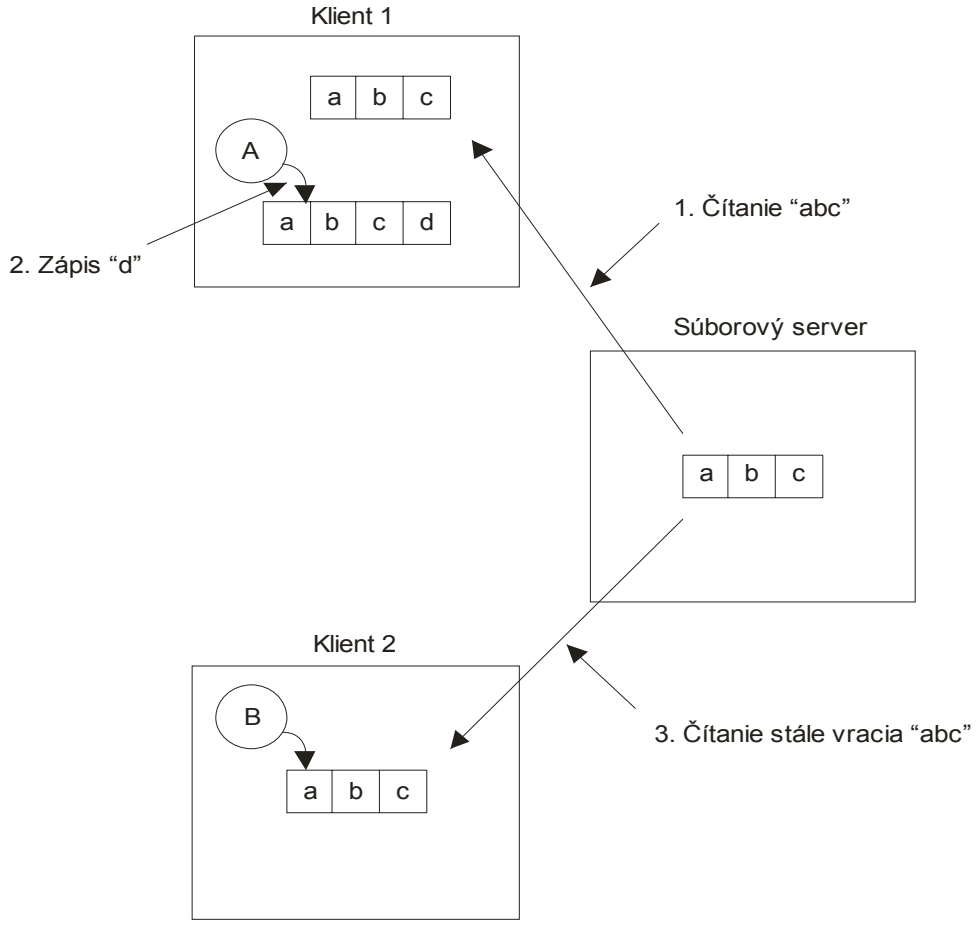
Prvé dve možnosti sú ľahko implementovateľné, špeciálne v prípadoch prepojenia existujúcich systémov, ktoré neboli navrhnuté na distribuované použitie. Posledná možnosť je najzložitejšou, vyžaduje si starostlivý návrh, je však potrebná v prípade, ak chceme dosiahnuť, aby sa distribuovaný systém správal ako jeden počítač.

Sémantika zdieľania súborov

Ako bolo opísané v kapitole 4.1.5, v jednoprocesorových systémoch sa často používa UNIX sémantika zdieľania súborov. V distribuovaných systémoch je UNIX sémantika ľahko dosiahnuteľná, pokiaľ je v systéme len jeden súborový server a klienti neukladajú súbory do lokálnej medzipamäte. V takomto prípade všetky operácie *Read* a *Write* smerujú priamo na súborový server, ktorý ich spracováva prísne sekvenčne. Práve takýto prístup umožňuje použitie UNIX sémantiky aj v distribuovaných systémoch (okrem prípadov keď operácia *Read* vráti starú hodnotu napriek tomu, že operácia *Write* mohla byť iniciovaná skôr, avšak na súborový server mohla prísť z dôvodu sieťového oneskorenia neskôr). Prakticky však výkon distribuovaných systémov, kde by všetky požiadavky na súbory smerovali na jeden server väčšinou nie je postačujúci. Tento problém je často riešený tak, že si klienti udržiavajú lokálnu kópiu naposledy používaných súborov v ich privátnych medzipamätiach. Toto riešenie

však prináša problémy, ktoré vznikajú vtedy, keď si klient lokálne modifikuje súbor vo svojej medzipamäti a krátko po tom iný klient požiada o čítanie tohto súboru zo servera – v takomto prípade dostane tento klient neplatný súbor ako je to znázornené na obrázku 4-8.

Jedným spôsobom ako prekonať tento problém je prenášanie všetkých zmien súborov z lokálnych medzipamätí okamžite späť na server. Hoci je tento prístup konceptuálne jednoduchý je neefektívny. Alternatívnym riešením je povoliť sémantiku zdieľania súborov. Namiesto toho aby sa požadovalo, aby operácia *Read* vrátila hodnoty po všetkých predchádzajúcich operáciách *Write* je možné zaviesť pravidlo: „Zmeny na otvorenom súbore sú spočiatku viditeľné len pre procesy (prípadne počítač) ktorý modifikuje súbor. Pre ostatné procesy (prípadne počítače) budú zmeny viditeľné až po zatvorení súboru“. Osvojenie tohto pravidla nemení nič na situácii z obrázku 117-8, avšak predefinuje aktuálne správanie (*B* získa pôvodnú hodnotu zo súboru) na korektné. Keď *A* zatvorí súbor, pošle jeho kópiu na server a tak ďalšie operácie *Read* vrátia novú hodnotu. Toto pravidlo je často implementované a je známe pod menom *session sémantika*.



Obrázok 4-8. Použitie lokálnej medzipamäte na klientovi v distribuovanom systéme.

Pri použití session sémantiky nastáva otázka, čo sa stane ak dvaja alebo viacerí klienti súčasne modifikujú rovnaký súbor vo svojich medzipamätiach. Jedným z možných riešení by mohlo byť nasledujúce: tak, ako sa súbory striedavo zatvárajú, je ich hodnota odoslaná späť na server a finálna hodnota závisí od toho, ktorý klient naposledy zatvoril súbor. Menej šťastná, ale trochu jednoduchšie implementovateľná alternatíva môže spočívať v tom, že konečný výsledok je výsledkom jedného z kandidátov, avšak voľba konkrétneho z nich sa nešpecifikuje.

Hlavným problémom pri použití session sémantiky je, že nedodržuje jeden z aspektov UNIX sémantiky, a to, že každá operácia *Read* vracia naposledy zapísanú hodnotu. V UNIXe je asociovaný s každým otvoreným súborom ukazovateľ, ktorý indikuje aktuálnu pozíciu v súbore. Operácia *Read* číta a operácia *Write* zapisuje údaje počnúc touto pozíciou. Tento ukazovateľ je zdieľaný medzi procesom, ktorý otvoril súbor a všetkými jeho potomkami. V prípade session sémantiky, pokiaľ bude potomok vykonávaný na inom počítači nie je možné dosiahnuť tento spôsob zdieľania.

Dôsledok tohto problému sa dá ukázať na veľmi jednoduchom príkaze shellu:

```
% vykonaj > výstup
```

kde príkaz vykonaj spúšťa dva programy *A* a *B* po sebe, jeden po druhom. Pokiaľ oba programy produkujú výstup, očakáva sa, že výstup produkovaný procesom *B* bude priamo nasledovať za výstupom z procesu *A* a to v súbore *vystup*. Spôsob akým sa tento stav dosahuje na jednoprocesorových systémoch, je založený na tom, že v čase, keď štartuje proces *B* zdedí po procese *A* ukazovateľ ktorý je medzi nimi a shellom zdieľaný. Pri takomto postupe bude prvá slabika výstupu programu *B* zapísaná hneď po poslednej slabike zapísanej programom *A*. V prípade session sémantiky a bez zdieľaných ukazovateľov súborov je nutné použiť úplne iný mechanizmus aby sa dosiahlo rovnaké správanie tohto jednoduchého shellovského príkazu.

Úplne odlišným prístupom ku sémantike zdieľania súborov v distribuovaných systémoch je používanie takzvaných immutable súborov. Pri tomto prístupe je umožnené vytvoriť nový súbor a vložiť ho do adresára pod rovnakým menom ako meno pôvodného súboru, pričom sa predchádzajúca verzia súboru stáva (pod týmto menom) nedostupnou. Takto je síce nemožné modifikovať súbor *X*, ostáva však možnosť atomicky nahradiť súbor *X* novým súborom. Inými slovami: aj keď súbor nemôže byť modifikovaný, adresár modifikovaný byť môže. Rozhodnutím, že súbory nemôžu byť modifikované sa eliminuje problém ako sa dohodnúť s dvomi procesmi – jedným z nich, ktorý zapisuje do súboru a iným, ktorý z neho číta.

Problém, ktorý ostáva je rozhodnutie čo sa stane v prípade, ak sa dva procesy snažia nahradiť rovnaký súbor v rovnakom čase. Ako aj v prípade session sémantiky, aj tu sa zdá najlepším riešením nahradenie starého súboru najnovším súborom alebo nahradiť súbor „druhým“ súborom nedeterministicky.

Trocha zložitejším problémom je, čo robiť v prípade, ak je súbor nahradený iným súborom v čase, keď z neho práve číta nejaký proces. Jedným z možných riešení je nejakým spôsobom zabezpečiť pre čítajúceho pokračovanie čítania starej verzie súboru, napriek tomu, že sa už nenachádza v žiadnom adresári. Podobne to rieši systém UNIX v ktorom proces, ktorý otvoril súbor môže bez problémov pokračovať v práci s ním napriek tomu, že mohol byť medzičasom vymazaný. Iným riešením je detekcia stavu zmeny súboru a zabránenie jeho ďalšieho čítania tým, že operácia *Read* vráti chybu.

Posledným spôsobom ako dosiahnuť možnosť zdieľania súborov v distribuovanom systéme je použitie atomických transakcií. Pre prístup k súboru alebo skupine súborov proces najprv vykoná nejaký druh primitívy *BEGIN TRANSACTION* pre signalizovanie, že ďalšie operácie musia byť vykonané nedeliteľne. Potom prichádzajú systémové volania na čítanie a zápis jedného alebo viacerých súborov. Ak je požadovaná úloha vykonaná, tak sa použije primitíva *END STRANSACTION*. Kľúčovou vlastnosťou tejto metódy je, že systém zaručuje, že všetky operácie v rámci transakcie budú vykonané v poradí v akom prídu bez interferencie z iných – súbežných transakcií. Pokiaľ sa dve alebo viac transakcií začnú vykonávať v rovnakom čase, systém zabezpečí, že výsledok bude rovnaký, ako keby sa vykonali v nejakom (nešpecifikovanom poradí).

V tabuľke 4-1 je sumarizácia štyroch spôsobov na zdieľanie súborov v distribuovaných systémoch popísaných vyššie.

Tabuľka 4-1. Spôsoby zdieľania súborov v distribuovaných systémoch.

Metóda	*Komentár*
UNIX sémantika	Každá operácia nad súborom je okamžite viditeľná pre všetky procesy
Session sémantika	Zmeny sú viditeľné pre ostatné procesy až po zatvorení súboru
Immutable súbory	Nie sú umožnená aktualizácie obsahu súboru; zjednodušené zdieľanie a replikácia
Transakcie	Zmeny, ktoré sa vykonali buď nastanú všetky, alebo nenastane žiadna

Súborový systém AFS

Andrew File System (AFS) je bezpečný globálny distribuovaný systém, ktorý poskytuje vlastnosti nezávislosti lokácie, škálovateľnosti a poskytuje možnosť transparentnej migrácie dát (Wachsmann, 2005). AFS pracuje vo veľkom počte operačných systémov a je používaný množstvom veľkých webových serverov, na ktorých „slúži“ mnoho rokov. AFS poskytuje množstvo zaujímavých vlastností nedostupných v iných distribuovaných systémoch, a to napriek tomu, že je takmer 20 rokov starý. Pôvodne komerčný AFS, bol uvoľnený firmou IBM pre Open Source komunitu v roku 2000, odkedy sa vďaka tomuto kroku teší pomerne veľkej popularite.

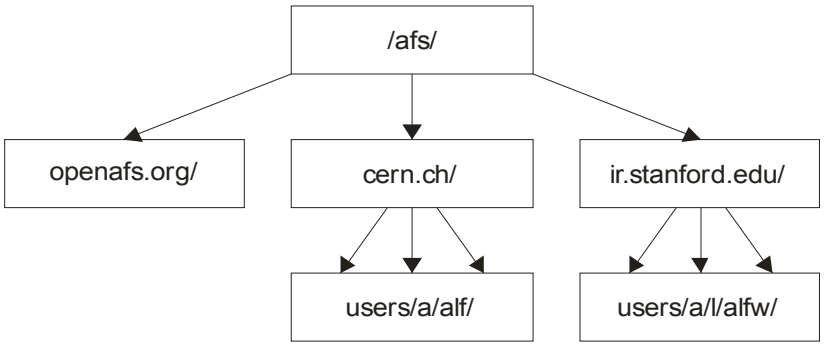
Medzi vlastnosti tohto súborového systému patria:

- Multiplatformovosť – pracuje na platformách od UNIX, Linux, Compaq, Sun, IBM, Microsoft Windows až pod Mac OS X.
- Všetky klientske počítače používajú lokálne medzipamäte. Manažér lokálnej medzipamäte sleduje používateľov na počítači a obsluhuje ich požiadavky na dáta. Zdieľanie dát prostredníctvom medzipamätí sa vykonáva v malých kúskoch súborov, ktoré sú kopírované z AFS na lokálny disk. Lokálnu medzipamäť počítača zdieľajú všetci používatelia toho istého počítača.
- AFS využíva globálne unikátny menný priestor, ktorý je znázornený na obrázku 4-9. Keďže sú tieto mená unikátne a taktiež neobsahujú mená serverov, AFS

spĺňa vlastnosti nezávislosti lokácie. Na nájdenie hľadaných dát klientmi má AFS replikované databázy na ich lokáciu, ktoré títo klienti kontaktujú. Práve touto vlastnosťou sa výrazne odlišuje od známeho Network File System (NFS), v ktorom má každý klient informáciu o hosťujúcich súborových serveroch jednotlivých častí NFS.

Jednotlivé nezávislé domény AFS sa volajú bunky (cells) a korešpondujú s Kerberos oblasťami (realms).

Možnosti škálovania AFS sú veľmi dobré; na modernom hardvéri dokáže jeden AFS server obslúžiť aj tisícky používateľov bez akýchkoľvek problémov a to napriek tomu, že bol povodne navrhovaný na pomer klientov k pomeru serverov 200:1.



Obrázok 4-9. Globálne unikátny menný priestor AFS (nezávislosť lokácie).

Na súborových serveroch AFS sú dáta ukladané na špeciálne partície – */vicep##*, kde ## môže nadobúdať hodnoty *a-zz*, čo umožňuje použitie 256 partícií v rámci jedného servera. Všetky tieto partície ukladajú dátové kontajnery zvané zväzky. Zväzky sú najmenšou entitou, ktorá môže byť presunutá, replikovaná alebo zálohovaná. Tieto zväzky obsahujú súbory a adresáre a je nutné ich pripojiť v rámci AFS, k tomu, aby boli použiteľné. Miesta pripojenia zväzkov sú reprezentované adresármi. AFS je obzvlášť vhodný na poskytovanie dát, ktoré sú len na čítanie, pretože tieto dáta môžu byť obsiahnuté v lokálnych medzipamätiach klientov. K tomu, aby zdieľanie týchto dát bolo ešte lepšie, AFS umožňuje vytváranie klonov (ktoré sú len na čítanie) na jeho rôznych súborových serveroch. Táto vlastnosť je výhodná hlavne v prípade, ak sa niektoré uzli poškodia, keďže klient môže byť presmerovaný na iný uzol, ktorý obsahuje identické dáta. Tieto replikačné techniky je vhodné použiť aj v prípade, ak sú jednotlivé súborové serveri od seba geograficky ďaleko.

AFS využíva na účely zálohovania snímkový mechanizmus. Tieto snímky vznikajú v prednastavený čas a pracujú na úrovniach zväzkov. Snímky môžu byť neskôr použité na zálohu údajov na záložné médiá.

Komunikačný protokol AFS je navrhnutý pre WAN siete. Využíva vlastnú implementáciu mechanizmu volania vzdialených procedúr zvanú *Rx*, ktorá pracuje nad protokolom UDP.

AFS využíva na autentifikáciu používateľov mechanizmus Kerberos 4, nie je však veľký problém použiť ani novšiu verziu Kerberos 5, ktorá zabezpečuje ešte vyššiu bezpečnosť. AFS poskytuje zabezpečenie adresárov prostredníctvom ACL (kapitola

4.1.1). Pridávať ACL môžu len používatelia alebo skupiny overené prostredníctvom mechanizmu Kerberos, pričom okrem tohto mechanizmu sa používa ešte ďalší zabezpečovací mechanizmus, ktorý slúži na autorizáciu používateľov a skupín (PTS).

4.1.9 Špeciálne súborové systémy

Za špeciálne súborové systémy je možné považovať všetky súborové systémy, ktoré nie sú diskové alebo sieťové. Špeciálne súborové systémy sa často používajú v operačných systémoch ako UNIX a Linux. Asi najrozšírenejším špeciálnym súborovým systémom súčasnosti je súborový systém ProcFS, ktorý sa používa na zistenie informácií o bežiacich procesoch operačného systému. Ďalšími zaujímavými špeciálnymi súborovými systémami sú napríklad sémantické súborové systémy, ktorých základnú funkcionalitu zhrnie nasledujúca podkapitola.

Sémantické súborové systémy

Sémantický súborový systém je úložiskový informačný systém, ktorý poskytuje flexibilný asociatívny prístup k obsahu úložiska tým, že extrahuje atribúty zo súborov (tieto atribúty môžu byť potenciálne pridané aj samotným používateľom) prostredníctvom transformátorov, ktoré sú špecifické pre každý typ súboru. Asociatívny prístup je zabezpečený jednoduchým rozšírením existujúcich protokolov hierarchických súborových systémov a tiež špeciálnymi protokolmi, ktoré sú založené na prístupe založenom na obsahu. Kompatibilita s protokolmi existujúcich súborových systémov môže byť zabezpečená vďaka konceptu virtuálneho adresára. Mená virtuálnych adresárov reprezentujú dopyty, ktoré poskytujú flexibilný asociatívny prístup k súborom a adresárom. Rýchly, na vlastnostiach založený prístup k obsahu súborových systémov býva implementovaný automatickou extrakciou a indexáciou kľúčových vlastností objektov súborového systému. Automatická indexácia súborov a adresárov sa nazýva *sémantika*, pretože používateľom programovateľné transformátory používajú na extrakciu vlastností pre indexovanie informáciu o sémantike objektov súborového systému.

Sémantický súborový systém poskytuje používateľské rozhranie, ako aj aplikačné programové rozhranie pre prístup k prostriedkom asociatívneho prístupu. Aplikačné programové rozhranie, ktoré umožňuje vzdialený prístup zahŕňa špecializované protokoly pre získavanie informácií (Ansi 39.50 version 2, 1991) a rozhrania založené na vzdialenom volaní procedúr (Tay & Ananda, 1990).

Veľkou výhodou väčšiny sémantických súborových systémov je, že sú kompatibilné s existujúcimi hierarchickými súborovými systémami; implementácia sémantických súborových systémov môže teda byť plne kompatibilná s existujúcimi protokolmi pre sieťové súborový systémy ako NFS a AFS.

Sémantika sémantických súborových systémov

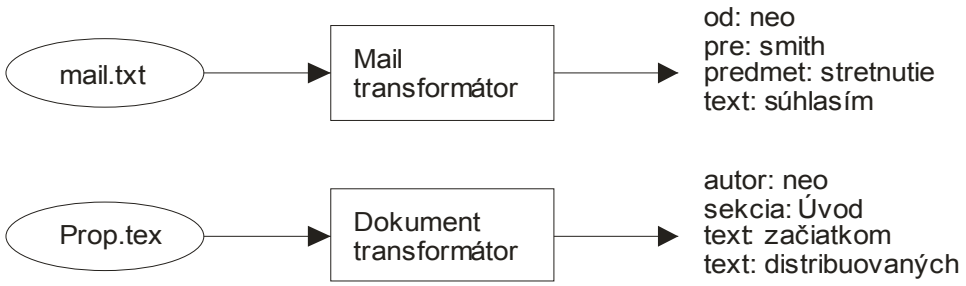
Sémantické súborové systémy môžu implementovať širokú škálu rôznych sémantík. Súbory uložené v sémantickom súborovom systéme sú pre vytvorenie opisných atribútov interpretované transformátormi (špecifickými pre typ súborov), ktoré neskôr umožnia prehľadávanie súborov (Gifford et al., 1991).

Atribút môže byť reprezentovaný dvojicou meno atribútu a hodnota, kde mená atribútov opisujú vlastnosti súborov, pričom hodnoty atribútov môžu byť rôznych typov ako napríklad reťazec alebo celé číslo. V niektorých prípadoch môže mať jeden

súbor viac atribútov s rovnakým menom atribútu. Na výber transformátora pre konkrétny typ súboru sa často používa používateľom rozšíriteľná *tabuľka transformátorov* (Gifford et al., 1991).

Iným prístupom môže byť označkovanie obsahu súborov, kde môžu byť použité značkovacie ontológie ako napríklad v súborových systémoch TagFS (Bloehdorn et al., 2006) a FlickrFS [11]. Tento prístup navyše umožňuje kompletné exportovanie metadát, ktoré sa potom môžu zdieľať medzi používateľmi.

Pre prispôsobenie rôznym typom súborov (ako napríklad súbory elektronickej pošty), ktoré zvyčajne obsahujú viaceré objekty sa v niektorých systémoch zovšeobecňuje jednotka asociatívneho prístupu a tak sa nepristupuje len k súborom ako k celku – namiesto toho sa pristupuje k jednotlivým objektom v nich. V (Gifford et al., 1991) nazvali autori tieto jednotky asociatívneho prístupu entitami. Tieto entity potom mohli pozostávať z celého súboru, objektu vo vnútri súboru alebo adresára. Výstup transformátorov môže pre jednotlivé súbory potom vyzerať tak, ako je to znázornené na obrázku 4-10.



Obrázok 4-10. Príklad výstupu transformátorov.

Rozhranie asociatívneho prístupu k sémantickému súborovému systému je založené na dopytoch, ktoré opisujú požadované atribúty entít (alebo značiek). Dopyt je teda opisom požadovaných atribútov, ktoré umožňujú vysoký stupeň selektivity pri lokalizácii požadovanej entity. Výsledkom dopytu je množina súborov a/alebo adresárov ktoré obsahujú opísané entity.

Sémantický súborový systém sa považuje za *dopytovo konzistentný*, keď zaručuje taký výsledok dopytu, ktorý korešponduje s aktuálnym obsahom súborového systému.

Ak sú ukončené aktualizácie obsahu sémantického súborového systému, tak môže spĺňať požiadavky na dopytovú konzistenciu. Táto vlastnosť je známa ako *konvergentná konzistencia*. V prípade, že sa v sémantickom súborovom systéme používajú vhodné atomické transakcie je možné dosiahnuť stálu dopytovú konzistenciu.

V (Gifford et al., 1991) tvoria položky adresára s menom atribútu virtuálne adresáre hodnôt. Virtuálny adresár hodnôt obsahuje jednu položku pre každý súbor (alebo adresár), ktorý spĺňa vlastnosti popísané menom atribútu a hodnotou. Virtuálny adresár */sfs/owner:/neo* teda bude v adresári */sfs* poskytovať položky súborov, ktoré vlastní používateľ *Neo*. Každá položka je symbolickým odkazom na skutočný súbor.

[11] manishrjain – flickrfs, http://manishrjain.googlepages.com/flickrfs/. Last accessed in January 2007

Príklad dopytu na sémantický súborový systém môže vyzerať takto:

```
% ls -F /sfs/owner:/neo
mail.txt@ paper.ex@ prop.tex@
%
```

V prípade (Bloehdorn et al., 2006) a iných značkovacích sémantických súborových systémov by mohol rovnaký dopyt vyzerať napríklad:

```
% ls /owner-neo
mail.txt@ paper.ex@ prop.tex@
%
```

Táto syntax sa môže líšiť pri rôznych implementáciách sémantických súborových systémov a môže byť rozšírená o bohatší dopytovací jazyk (použitie logických operátorov, umožnenie vyhľadávania na viacerých sieťových počítačov súčasne a podobne).

V súčasnosti existujú dva hlavné koncepty architektúry sémantický súborových systémov. Pri prvom koncepte môže sémantický súborový systém prekryť štandardný súborový systém, čím sa zabezpečí, aby všetky operácie v systéme prechádzali cez vrstvu sémantického súborového systému. Tento spôsob má pre používateľa výhodu, že umožňuje využívať všetky možnosti sémantického súborového systému vždy, bez toho aby sa odvolával na význačné adresáre pre spracovanie dopytu. Taktiež umožňuje serveru okamžitú indexáciu pri odpovedi na operácie zmeny súborového systému.

Alternatívnou možnosťou môže byť koncept, pri ktorom sémantický súborový systém vytvára symbolické (prípadne iné) odkazy do virtuálnych adresárov základného súborového systému. V tomto prípade, na rozdiel od prvého konceptu operácie obchádzajú vrstvu sémantického súborového systému.

Keďže oba koncepty majú ako svoje výhody, tak aj nevýhody, môže byť pomerne ťažké rozhodnúť sa pre jeden z nich. To, ktorý z nich získa väčšie uplatnenie je pravdepodobne otázkou blízkej budúcnosti; za predpokladu, že sa sémantické súborové systémy stanú dostupné pre väčšinu bežných používateľov.

Potencionálne rozšírenia sémantických súborových systémov

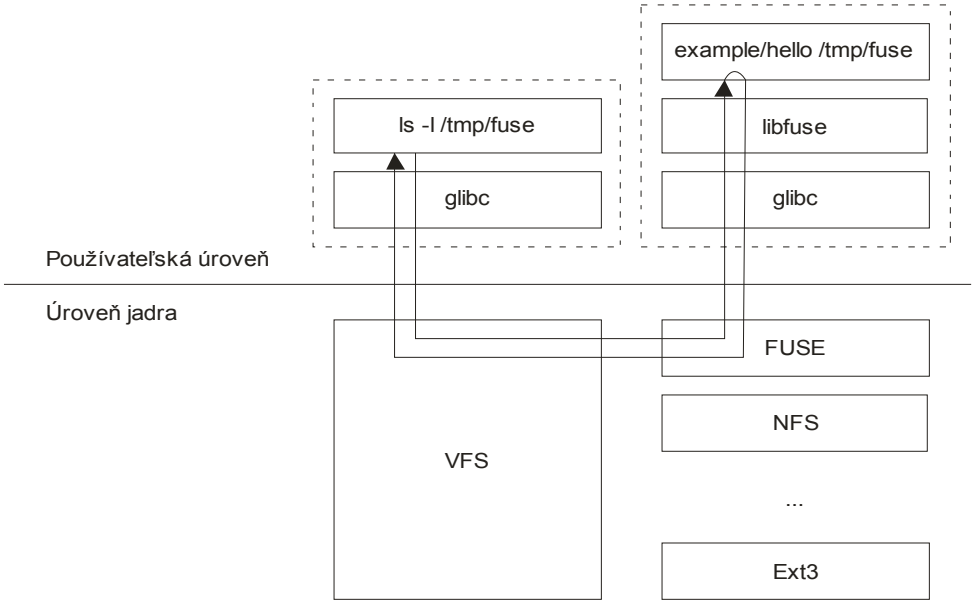
Prehľadávanie sémantiky nemusí byť pre všetky typy súborov vždy jednoduché. Pri jednoduchých formátoch (hlavne textových súboroch) je táto úloha jednoduchá, avšak pri súboroch obsahujúcich multimediálne dáta môže byť táto úloha pomerne zložitá (príkladom by mohol byť multimediálny video súbor, v ktorom by bolo umožnené vyhľadávanie v snímkach videa podľa času, prípadne vyhľadávanie podľa obsahu). Ďalšie možné vylepšenia by mohli vzniknúť rozšírením modelu dát sémantických súborových systémov. Príkladom môže byť entitno-relačný prístup (Cattell, 1983), ktorý ponúka väčšie možnosti ako obyčajné prehľadávanie jednoduchých atribútov. Keďže nároky na vyhľadávanie údajov rastú veľmi rýchlo, je nutné aby sa im prispôsobovali aj technológie. Pri sémantických súborových systémoch môže prispôsobenie týmto novým požiadavkám znamenať napríklad organizáciu sémantických súborových systémov do sémantických knižničných systémov (Ding, 2003). Dôležitou otázkou aj napriek množstvu existujúcich rozšírení je však, či výsledok dopytu zadaný používateľom zodpovedá jeho potrebám. Práve v tejto oblasti sa očakáva ďalší výskum.

Súborový systém FUSE

Súborový systém FUSE predstavuje hybridný súborový systém (Zadok et al., 2006), ktorý pozostáva z dvoch častí:

- štandardný súborový systém na úrovni jadra, ktorý presmerováva systémové volania do démona na používateľskej úrovni
- knižnica pre jednoduchý vývoj špecifických démonov pre súborový systém

Architektúra tohto súborového systému spolu s priebehom volania súborového systému z používateľskej úrovne je znázornená na obrázku 4-11.



Obrázok 4-11. Architektúra súborového systému FUSE spolu s priebehom volania súborového systému z používateľskej úrovne.

Vývoj nových súborových systémov pre FUSE je relatívne jednoduchý, keďže démon na používateľskej úrovni môže využívať bežné systémové volania pre VFS (ako je to vidieť aj na obrázku 4-11).

Súborový systém FUSE však má dve hlavné nevýhody:

1. výkonnosť je limitovaná z dôvodu použitia rozhrania medzi úrovňou jadra a používateľa
2. súborový systém môže používať len FUSE API, ktoré sa takmer zhoduje z VFS API, zatiaľ čo súborové systémy na úrovni jadra môžu pristupovať k bohatšiemu API jadra operačného systému (napríklad volania pre manažment siete alebo procesov)

Medzi najznámejšie súborové systémy založené na FUSE patria:

a. CvsFS

b. Bluetooth File System

c. Nfsmount
d. BitTorrent File System
e. FuseISO
f. Logic File System
g. FuseCompress
h. CopyFS
i. Captive NTFS
j. CryptoFS

Súborových systémov založených na FUSE v súčasnosti ustavične pribúda, a vďaka jednoduchej implementácii týchto súborových systémov je predpoklad, že ich počet bude narastať aj naďalej.

Súborový systém WinFS

Čoraz viditeľnejším problémom súčasnosti sa stáva obrovské množstvo dát v ktorých je však čím ďalej, tým ťažšie nájsť to, čo je v danom čase potrebné. V prípade, ak sa naplnia predpovede o veľkostiach diskov budúcnosti (terabajty) naberie tento problém ešte väčšie rozmery. Nájdenie špecifických častí dát závisí od identifikácie nejakej špecifickej položky v nich, to však znamená, že všetky dáta a ich časti by mali byť roztriedené. Hrubozrnný systém, ako súborový a adresárový systém nie je dostatočne flexibilný na to, aby umožňoval skladovať všetky informácie, ktoré sú potrebné pre uľahčenie respektíve umožnenie takéhoto hľadania.

WinFS je súborový systém, v ktorom čokoľvek, čo je uložené v jeho úložisku je považované za položku a každá položka má svoje metadáta popísané určitou schémou. Položky, ktoré sa riadia schémou sú uložené v úložisku WinFS ako serializované .NET objekty a sú sprístupňované prostredníctvom pohľadov T-SQL, ktoré umožňujú prístup k vlastnostiam položiek.

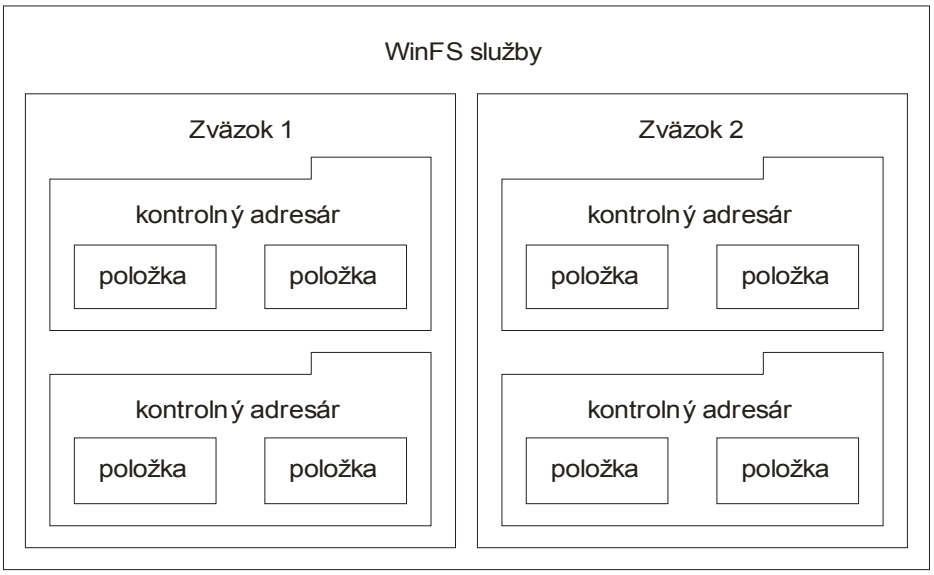
Obrázok 4-12 znázorňuje konceptuálnu architektúru položiek vo WinFS. Každá položka je v kontrolnom adresári; tieto sú použité pre kontrolu životnosti položiek. Adresáre sú tiež položky a každý adresár musí byť obsiahnutý vo vnútri iného adresára. Jedinou výnimkou z tohto pravidla je koreňový kontrolný adresár. Položky sú uložené vo zväzku, ktorý je najväčším autonómnym kontajnerom položiek. Na každom počítači pracujú služby WinFS nad týmito zväzkami a zabezpečujú ukladanie, zrušenie, modifikáciu a lokáciu položiek.

Úložisko WinFS

WinFS poskytuje pohľady T-SQL na tabuľky položiek, s jedným pohľadom pre každý typ položky. Pre dosiahnutie bezpečnosti úložiska a konzistencie dát môžu byť dáta cez pohľady len čítané a modifikované môžu byť len použitím uložených procedúr. Príklad nižšie vracia požadované informácie pre všetky skladby v knižnici médií na počítači:

```
SELECT Artist, Title, Album FROM Audio.[Track!V1]
```

WinFS poskytuje niekoľko aplikačných rozhraní (API) vyššej úrovne na jednoduché vkladanie, rušenie a dopytovanie položiek.



Obrázok 4-12. Konceptuálna architektúra položiek vo WinFS.

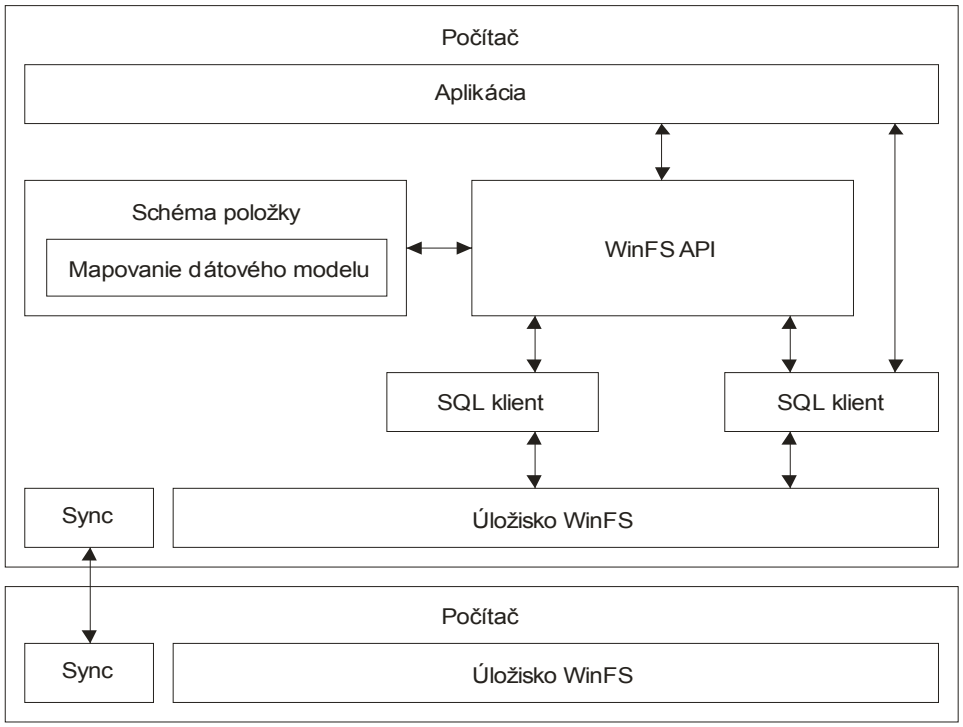
Manažované API WinFS

Obrázok 4-13 znázorňuje hlavnú časť architektúry WinFS. Úložisko WinFS môže byť sprístupnené prostredníctvom niekoľkých API, aplikácie môžu použiť napríklad ADO.NET s SQL Server Provider, OLE DB s natívnym kódom, API založené na COM objektoch a manažovateľné API. Vo všeobecnosti, vývojári uprednostňujú tieto API pred T-SQL, avšak majú stále prístup k T-SQL keďže existuje množstvo prípadov, kedy nie sú iné API postačujúce. Interne v systéme sa všetky prístupy cez rôzne API transformujú vždy do T-SQL.

Všetok kód WinFS API je vykonávaný prostredníctvom *ItemContext*, pričom *ItemContext* je pomerne široký pojem:

1. WinFS API musí vytvoriť spojenie na úložisko; toto spojenie zostane aktívne pokiaľ je aktívne *ItemContext*.
2. Hneď ako *ItemContext* reprezentuje dátové spojenie, poskytuje transakčné prostriedky; v prípade ak používateľ chce, môže vykonať množinu zmien nad úložiskom atomicky.
3. Nakoniec *ItemContext* opisuje oblasť činností, ktoré budú vykonávané po definovaní spôsobu vyhľadávania položiek.

ItemContext poskytuje prístup k dátovému pripojeniu prostredníctvom *BeginTransaction* a *EndTransaction*. *BeginTransaction* vytvára novú transakciu a vracia objekt transakcie; všetky vykonané zmeny v úložisku sa uložia až po vykonaní operácie *EndTransaction*, ktorá zabezpečí aby boli všetky vykonané zmeny atomické. Po ukončení prístupu k úložisku by sa mala zavolať operácia *Close* nad *ItemContext* pre uvoľnenie dátového pripojenia.



Obrázok 4-13. Hlavná časť architektúry WinFS.

Položky obsiahnuté v adresároch

Najrozsiahlejšia úroveň klasifikácie objektov sa dosahuje prostredníctvom adresárov. Adresáre WinFS sú použité na ukladanie položiek, pričom jedna položka môže existovať súčasne vo viacerých adresároch WinFS. V prípade požiadavky o vytvorenie adresára sa tento uloží do niektorého z existujúcich adresárov. Koreňový adresár každého zväzku WinFS obsahuje štyri adresáre: System Data, Volume, Schema a Store Information. Adresáre, ktoré vytvárajú používatelia sú v adresári SystemData. Pri vytváraní položky v adresári sa využívajú takzvané udržiavacie odkazy. Objekt adresára je len inou položkou, ktorá má udržiavací odkaz pre každú položku, ktorú obsahuje. Tento udržiavací odkaz je špeciálny, keďže určuje životnosť položky na ktorú odkazuje a všetky udržiavacie odkazy na položku predstavujú počet referencií (ako *hardlink* v OS UNIX). Pokiaľ sú zrušené všetky udržiavacie odkazy na položku, je zrušená aj položka samotná. Adresáre WinFS nie sú viazané k typom položiek, ktoré obsahujú. V praxi to znamená napríklad, že adresár Letná dovolenka 2003 môže obsahovať všetky fotky z dovolenky, emailové správy potvrdzujúce rezervácie a tiež pesničku Dovolenka od nejakého autora.

Notifikácie

Ako aj meno predpokladá WinFS notifikácie sú mechanizmus umožňujúci informovať kód programu o zmene položky. WinFS používa model udalostí .NET pre registrovanie a ošetrovanie notifikácií. Registrácia pre notifikácie sa udržiava dovtedy, kým sa neukončí aplikácia, alebo kým si táto aplikácia neodregistruje tieto notifikácie.

Na monitorovanie jednotlivých položiek môže byť vytvorený Objekt *ItemWatcher*, ktorý obsahuje aj udalosť *ItemChanged*:

```
public event ItemChangedEventHandler ItemChanged;
```

Aplikácia teda môže pristúpiť k položke v úložisku a pridať obslužný program (handler) pre túto položku. Neskôr, keď iná aplikácia vykoná zmenu na tejto položke, tak sa pošle udalosť pre prvú aplikáciu. Táto zmena môže pochádzať od toho istého procesu, od iného procesu, alebo dokonca od iného počítača. Pokiaľ zmena nastane z iného počítača potom sa prejaví na pôvodnom počítači až po ich synchronizácii.

Synchronizácia

WinFS je služba na ukladanie dát, preto je prirodzené, že podporuje replikáciu na úložiská, ktoré sú na iných počítačoch. Táto replikácia sa vo WinFS nazýva synchronizácia. Pre implementovanie synchronizácie si WinFS najprv zistí ktoré dáta boli zmenené a po tom určí ktoré z nich je nutné replikovať. Najmenšie časti schémy položiek, ktoré sú vo WinFS sledované sú elementy (položky sa skladajú z elementov). Pokiaľ je element v položke zmenený pre WinFS je zmenená celá položka. Položky sú vo WinFS najmenšie jednotky konzistencie a preto sú aj najmenšou jednotkou dát, ktoré môžu byť replikované na iný počítač – položka je teda replikovaná buď ako celok, alebo nie je replikovaná vôbec.

Pred synchronizáciu musí používateľ identifikovať počítače, ktoré majú byť jej súčasťou. Aplikácie môžu iniciovať synchronizáciu prostredníctvom WinFS API alebo môže synchronizácia nastať ako udalosť plánovanej úlohy. V oboch prípadoch aplikácia, ktorá iniciovala akciu poskytne synchronizačný profil pre WinFS. Tento profil je informáciou o adresári, ktorý sa má synchronizovať, o spôsobe synchronizácie (len odoslanie, len prijatie, odoslanie aj prijatie) a informáciou o položkách, ktoré by mali byť synchronizované. Naviac obsahuje profil aj informáciu o tom, ako riešiť konflikty. Stratégia riešenia konfliktov môže vyvolať akciu, ako napríklad jednoduché odmietnutie replikácie, ktorá môže byť spracovaná automaticky kúskami kódov – takzvanými rozkladačmi konfliktov alebo môže požiadať používateľa o zásah.

Súborový systém Active Disk File System

V súborovom systéme založenom na aktívnych diskoch (ADFS) je každý súbor považovaný za pomenovaný objekt kombinovaný s kódmi svojich operácií. Pokiaľ niektoré objekty reprezentujú rovnaké charakteristiky a zdieľajú rovnaké operácie ale ich dátové komponenty sú odlišné, tieto objekty vytvárajú určitú triedu svojho typu. Objekty sú sprístupňované prostredníctvom presne definovaných rozhraní súborového systému. Niektoré rozhrania sú použité na vykonanie kódu používateľom definovanej operácie pre niektoré objekty. Práve táto vlastnosť môže byť veľmi zaujímavá v prípade špecifických požiadaviek používateľov. Autori tohto systému (Lim et al., 2001) však uvádzajú, že práve táto časť systému nie je zatiaľ implementovaná, preto je ťažké sa vyjadrovať k potencionálnemu využitiu a obľube takejto možnosti zo strany používateľov. Hlavná výhoda tohto nového prístupu uvádzaná autormi má spočívať v redukcii záťaže centrálneho súborového servera tým, že časť funkcionality systému z centrálneho súborového manažéra sa bude vykonávať priamo na úrovni disku.

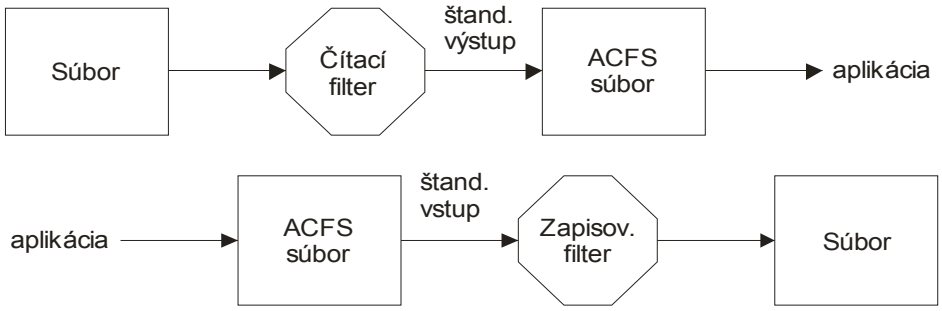
Aktívny súborový systém pre dolovanie dát a multimédiá

Dostupnosť lacnej pamäte a výpočtového výkonu otvorila cestu k novej paradigme zvanej Aktívne disky (Riedel et al., 1998). Aktívne disky sú pamäťové zariadenia s určitou schopnosťou pre vykonávanie kódu. Použitím aktívnych diskov je možné vykonať určité operácie na samotnom. Server môže použiť výsledok týchto operácií alebo vykonať ďalšie operácie nad týmto výsledkom. Táto časť počítačovo náročných operácií môže byť presunutá na pamäťové zariadenia. Zo zvyšovaním prenosovej rýchlosti siete môžu byť pamäťové zariadenia pripojené priamo na sieť namiesto zbernice počítača. Tento posun je v skutočnosti nutný pretože lokálne pamäte sú plné problémov škálovateľnosti, zdieľania a manažmentu.

Nové architektúry v tejto oblasti sú: Storage Area Networks (SAN) a Network Attached Storage (NAS). Zatiaľ čo SAN pracuje na úrovni SCSI, NAS pracuje so systémovými volaniami UNIXu. V tejto oblasti prebieha rozsiahly výskum (Benner, 2001, Hernandez et al., 2002). Tieto disky môžu byť použité aj na dolovanie dát, na čo v súčasnosti aplikovávame určité druhy filtrov (na veľké objemy dát). Tieto filtre sú zvyčajne implementované na vyššej úrovni logiky aplikácie na dolovanie dát.

V tomto aktívnom súborovom systéme (Srinivasan & Singh, 2000, Máté, 2006) autori implementovali tieto filtre, ako časti súborového systému UNIX. Nápad spočíval v súborovom systéme, ktorý podporuje filtrovanie na aplikačnej úrovni a bude kompatibilný zo súborovým systémom UNIX.

ACFS je použitý na vytvorenie väzby medzi súborom a procesom, ako je to znázornené na obrázku 4-14. ACFS i-uzol obsahuje pole pre *pid* procesu, ktorý je s ním asociovaný. Pokiaľ tento proces ukončí svoju činnosť je *pid* nastavený na nulu. Keď aplikácia požiada o otvorenie súboru *open("acfs://filterR?file","r")*, filter monitor vytvorí súbor ACFS a presmeruje štandardný výstup filtra f*ilterR* na ACFS súbor. Deskriptor súboru je vrátený aplikácii ktorá prislúcha ACFS súboru. V prípade požiadavky na zápis by bola situácia podobná: *open("acfs://filterW?file", "w")*, aplikácia zapíše do ACFS súboru to, čo načítala z filtra. V tomto prípade je výstup filtra presmerovaný do aktuálneho otvoreného ACFS súboru.



Obrázok 4-14. Hlavná časť architektúry WinFS.

Toto jednoduché rozšírenie súborového systému UNIX však v podstate len mení vyhľadávanie zo stavu: program, ktorý prehľadáva požadované súbory (všetky otvorí a prezrie) na: program, ktorý otvorí každý súbor a tento vyhľadávací program podstrčí tomuto súboru aby niečo našiel (pre každý jeden súbor zvlášť). Otázkou teda môže byť, čo tento prístup prináša, čo v súčasnosti nie je možné spraviť inými (jednoduchšími) prostriedkami.

4.2 Plne aktívny súborový systém

Táto nová koncepcia by mala umožňovať prekonanie niektorých nevýhod bežných – pasívnych súborových systémov. Každý súbor by považoval za „živú“ entitu, ktorá koná v prospech seba podľa presne definovaných pravidiel – podobne ako navrhuje (Lim et al., 2001) s tým rozdielom, že by sa výpočtový kód súborov vykonával nie len v prípade požiadavky používateľa, ale kedykoľvek; tiež by sa nemusel vykonávať v prostredí aktívnych diskov.

Pokiaľ by sa napríklad súboru „povedalo“, že jeho obsah je veľmi dôležitý, potom by mal konať tak, aby svoje údaje ochránil (jemu povolenými operáciami). Napríklad by si mohol vybrať spoľahlivejší súborový systém proti strate údajov, vytvoriť svoju záložnú kópiu a podobne. Týmto súborom by malo byť umožnené ukladanie ich dát do ľubovoľného úložiska údajov, napríklad lokálneho súborového systému, sieťového súborového systému, alebo aj databázy.

Koncept aktívneho súborového systému (Máté, 2007) má umožniť vytvoriť prostredie, ktoré vyhovuje ako dočasným – nie veľmi dôležitým údajom (ako napríklad Ramdisk), tak aj prostredie pre údaje, ktoré sú extrémne dôležité. Umožňoval byť tiež zavedenie rôznych „typov“ súborov, pričom každý súbor určitého typu by mal svoje špecifické vlastnosti a operácie, ktorými by sa riadil. Na rozdiel od pasívneho súborového systému by tento prístup tiež umožňoval zavedenie rôznych „kategórií“ súborov, ktoré by sa o seba „starali“ a spolu „interagovali“ (aj keď nie nutne v zmysle agentových systémov). Súbor ako celok by dokonca nemusel byť uložený na tom istom médiu – mohol by byť distribuovaný na viaceré médiá.

4.2.1 Výhody a nevýhody tohto prístupu

Medzi hlavné výhody tohto prístupu patrí:

- *Vyššia flexibilita súborov* – súbor, v prípade ak má schopnosť komunikácie v rámci akéhosi výpočtového prostredia, môže na základe získaných informácií učiniť jednoduché rozhodnutia, ktoré však môžu mať ďalekosiahle následky. Jednoduchým príkladom môžu byť súbory, ktoré si overujú svoju integritu (prostredníctvom rôznych hešovacích algoritmov) a v prípade zistenia svojho poškodenia túto informáciu predávajú ostatným súborom. V takomto prípade by sa ostatné (ešte nepoškodené) súbory mohli na základe týchto informácií rozhodnúť o vytvorení svojich záložných kópií, prípadne sa presunúť na iné úložisko, ktoré je bezpečnejšie.
- *Možnosť existencie súborov s výrazne odlišnými vlastnosťami* – súbory môžu disponovať rôznymi vlastnosťami a schopnosťami, ako napríklad: schopnosť komprimácie alebo šifrovania svojho obsahu, schopnosť vytvárať viaceré verzie samého seba a podobne. V prípade adresárov by to mohli byť schopnosti hľadania súborov, ktoré vyhovujú určitým jednoduchým podmienkam a v prípade ich úspešného nájdenia by ich mohol adresár zaradiť do svojho obsahu (napríklad adresár zo súbormi väčšími ako 100MB).
- *Možnosť zavedenia hierarchie pri sémantickom prehľadávaní súborového systému* – v prípade ak súborom poskytneme schopnosti prehľadávať svoj obsah (podobnými metódami ako je tomu v prípade sémantických súborových systémov) a ak adresárom poskytneme metódy na prehľadávanie nájdenej

sémantiky v súboroch a tiež metódy na zhlukovanie týchto súborov získame schopnosť dynamickej kategorizácie súborov. V prípade, ak iné adresáre budú prehľadávať sémantiku týchto adresárov a na základe nej ich budú zoskupovať, nám vznikne hierarchia pri sémantickom prehľadávaní súborového systému, čo môže výrazne urýchliť prehľadávanie existujúcich súborov.

- *Možnosť využitia tohto princípu na LBA klastroch* – v prípade, ak sa na niektorom počítači v klastri preplní súborový systém (alebo zahltí CPU), by sa súbory z tohto počítača mohli podľa svojho uváženia automaticky presťahovať na iný súborový systém alebo počítač a udržiavať tým rovnováhu medzi využitím diskového priestoru (alebo CPU) na jednotlivých súborových systémoch a počítačoch klastra.
- *Možnosť využitia tohto princípu pri Gridovom počítaní* – keďže súbor môže vykonávať rôzne operácie, predstavuje objekt (podobne ako vo WinFS), zložený z dát a zo schopností. V prípade, ak by boli tieto schopnosti výpočtovými úlohami nad jeho dátami, by bolo možné ich považovať za úlohy, ktoré by sa mohli distribuovať v rámci Gridovej infraštruktúry.
- *Decentralizácia zabezpečenia súborov* – súbor by sa mohol sám rozhodnúť (na základe schopností, ktoré má k dispozícii) či povolí, alebo nepovolí operáciu, o ktorú je žiadaný. Taktiež v prípade, ak má požadované schopnosti môže predstierať vykonanie určitých operácií, ktoré môžu odradiť potenciálnych útočníkov a minimalizovať potenciálne škody páchané týmito útočníkmi.
- *Možnosť rozširovania súborov zo zachovaním spätnej kompatibility* – schopnosti súboru by sa mohli meniť podľa požiadaviek používateľa. Zmena existujúcej vlastnosti súboru by mohla zapríčiniť, že sa súbor prestane správať tak ako je požadované. Preto by súbor mal mať možnosť uchovávať si akúsi históriu svojich vlastností a v prípade potreby by mu malo byť umožnené vrátiť sa, k ich predchádzajúcim „verziám“.
- *Potenciálna možnosť samorozvíjania súborov* – tak ako majú súbory rôzne schopnosti na vykonávanie jednoduchých operácií, by mohli mať aj schopnosti, ktoré by im umožnili výmenu existujúcich schopností. Tieto schopnosti by potom mohli využívať aj súbory, ktoré ich po svojom vytvorení nemali. V takomto prípade by sa súborový systém začal „nápadne“ podobať na multiagentové systémy, čoho výhody resp. nevýhody je v súčasnosti ťažké odhadnúť.

Medzi potenciálne nevýhody tohto prístupu patrí:

- *Vysoká výpočtová náročnosť* – keďže by mohol každý súbor vykonávať určitý kód (podľa svojich schopností) je nutné dbať na to, aby sa nezahltili systémové prostriedky. Tento problém predstavuje pravdepodobne najzraniteľnejšie miesto tohto nového konceptu a bude nutné mu venovať adekvátnu pozornosť.
- *Nové bezpečnostné hrozby* – kód, ktorý vykonávajú jednotlivé súbory je nutné nejakým spôsobom vložiť do súborového systému resp. do súborov samotných. Súborový systém by mal poskytovať mechanizmy, ktoré by zabránili vloženiu takéhoto kódu útočníkom. V prípade, ak je súborom samotným umožnené tento kód vymieňať a používať, je toto riziko ešte väčšie. K tomu, aby sa bezpečnostné hrozby znížili na minimum bude nutné nájsť rozumný kompromis medzi funkcionalitou systému a množstvom a typom použitých bezpečnostných mechanizmov v ňom.

- *Potenciálna nerovnováha v systéme* – v prípade, ak umožníme súborom migráciu na iné – lepšie úložiská a/alebo počítače, by sa mohlo stať, že by v konečnom dôsledku boli všetky súbory presunuté na najvýkonnejší uzol distribuovaného systému, prípadne by mohli všetky súbory spoločne migrovať vždy na to úložisko, ktoré sa v danom okamihu javilo pre všetkých ako najlepšie. Túto nerovnováhu by bolo možné odstrániť napríklad spoplatnením prostriedkov v súborovom systéme.

4.2.2 Základné operácie ktoré by mal výpočtový systém poskytovať

Plne aktívny súborový systém by mal poskytovať také základné operácie, ktoré sú nevyhnutné pre naplnenie cieľov tohto konceptu. Tieto operácie by mali byť čo najjednoduchšie a mali by poskytovať možnosť rozšírenia do budúcnosti. Plne aktívny súborový systém by mal:

- poskytnúť komunikačné rozhranie pre objekty súborového systému
- poskytnúť dostatočné zabezpečenie objektov v rámci tohto prostredia
- poskytnúť informácie o jednotlivých úložiskách (rozhodovanie súborov)
- umožniť presun súborov medzi jednotlivými úložiskami dát (aj sieťovými)
- určovať cenu konkrétneho úložiska (spoplatnenie prostriedkov)
- poskytnúť spätnú väzbu medzi súborom a výpočtovým prostredím
- poskytnúť prostredie pre vykonávanie súborov (výpočtové prostredie)

4.2.3 Zaujímavé vlastnosti súčasných súborových systémov

V tejto práci boli opísané základné koncepty súborových systémov od diskových až po špeciálne koncepty.

Otázkou ktorá sa nastoluje je, ktoré z týchto konceptov a/alebo vlastností by bolo vhodné využiť pri tomto súborovom systéme.

Z predchádzajúcich podkapitol tejto kapitoly je zrejmé, že aj v prípade plne aktívneho súborového systému by malo ísť o *distribuovaný súborový systém* (kapitola 4.1.8). Z pohľadu požiadavky o samomigráciu súborov (LBA klastre) je zrejme nutné zabezpečiť vlastnosť *nezávislosti lokácie*. Požiadavka na možnosť vytvárania záložných kópií predurčuje nutnosť poskytnúť aj možnosť uchovávania *verzií súborov* a z pohľadu zabezpečenia dát je v dnešnej dobe prakticky nutnosťou *použitie žurnálovania*. Zvýraznené vlastnosti sú v súčasnosti poskytované súborovým systémom AFS. Súčasne však je nutné zabezpečiť možnosť *pridávania a odoberania vlastností* súborom a adresárom, comu najbližšie má koncept súborového systému Reiser 4 s konceptom zásuvných modulov.

Napriek tomu, že existencia súborových systémov s podobnými vlastnosťami existuje, pričom najmenej jeden z týchto súborových systémov (AFS) podporuje väčšinu vlastností navrhnutého konceptu nie je jednoduché tento koncept bez ďalšieho výskumu priamočiaro implementovať a overiť si jeho vlastnosti.

4.2.4 Problémy, ktoré zostávajú otvorené

- Ako zabezpečiť simultánne vykonávanie schopností súborov pri ich veľkom počte (stotisíce na jednom počítači)?
- V prípade, ak by nebolo možné zabezpečiť vykonávanie týchto schopností súbežne, ako určiť ktorý súbor, kedy a prečo bude vykonávaný?
- Ako v prípade dostatočného výkonu (prípadne limitovania počtu súborov, ktoré môžu byť v rámci súborového systému aktívne) zabezpečiť distribuovaný systém pred možnosťou vnesenia a vykonávania škodlivého kódu?

4.3 Zhrnutie

Koncept aktívnych súborov je relatívne nový. Vďaka narastajúcemu výpočtovému výkonu je v súčasnosti možné navrhovať a skúmať také vlastnosti systémov, ktoré by pred niekoľkými rokmi boli prakticky nemožné. Keďže je pomerne veľký predpoklad na ďalšie napredovanie v oblasti dostupnosti vyššieho výpočtového výkonu, je pravdepodobné aj to, že podobných prípadne ešte odvážnejších konceptov bude čím ďalej tým viac. Práve preto sa ďalšia etapa mojho výskumu v tejto oblasti bude zaoberať možnosťami fyzickej realizácie tohto konceptu (prípadne jeho ucelenej časti) a overením a pornovnaním jeho možností v porovnaní zo súčasnými súborovými systémami.

Použitá literatúra

ANSI 39.50 VERSION 2. National Information Standards Organization, Bethesda, Maryland, Second Draft, 1991.

BENNER, A.. Fiber channel for SANs. McGraw-Hill, 2001.

BENCHMARKS OF REISERFS VERSION 4, *http://www.namesys.com/benchmarks.html*. Last accessed in November 2006.

BLOEHDORN, S. – GÖRLITZ, O. – SCHENK, S. – VÖLKEL, M. *TagFS - Tag Semantics for Hierarchical File Systems*. In Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06), Graz, Austria, September 6-8, September 2006.

BOBROW, D.G.. – BURCHFIEL, J.D. – MURPHY, D.L. – TOMLINSON, R.S. – BERANEK, B. *TENEX, a Paged Time Sharing System for the PDP-10*. Communication of the ACM; Volume 15, Number 3, 1972.

CATTELL, R.G.G. *Design and implementation of a relationship-entity-datum data model*. Xerox Corporation, Palo Alto Research Center, May 1983.

GORTER, O. Home/projects/dbfs, *http://ozy.student.utwente.nl/projects/dbfs/*. Last accessed in January 2007.

DING, H.. *Challenges in Building Semantic Interoperable Digital Library System*, Department of Computer and Information Science, Norwegian University of Science and Technology, 7491, Trondheim, Norway, 2003.

GIFFORD, D. – JOUVELOT, P. – SHELDON, M. – O'TOOLE, J. *Semantic File systems*. ACM Operating Systems Review, pages 16-25, Oct. 1991.

GORTER, O. *Database File System, An Alternative to Hierarchy Based File Systems*. Graduation project, University of Twente, Computer Sciences, Enschede, the Netherlands, 2004.

GRÜNBACHER, A.. SuSE Linux AG Nuremberg, *POSIX Acess Control Lists on Linux*. SuSE Labs, 2003.

HERNANDEZ, R. – CHAL, C.K. – COLE, G. – CARMICHAEL, K. NAS and iSCSI Solutions. IBM Redbook, Feb 2002.

KROAH-HARTMAN, G. Linux technology center, *udev – A Userspace Implementation of devfs*. Proceedings of the Linux Symposium, Ottawa, Ontario Canada, July 23th–26th, 2003.

LEVY, E. – SILBERSCHATZ, A. *Distributed File Systems: concepts and examples*. Department of computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188.

LIM, H. Trust Services Engineering Mountain View; Chirag Wighe, Server Products Group Wind Rivers Systems Alameda, *Active Disk File System : A Distributed, Scalable File System*. MSS, Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies - Volume 00, 2001.

MÁTÉ, J. Computing Environment for Active File System. In Mária Bieliková, editor, IIT.SRC 2006: Student Research Conference, pages 235-242. Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, April 2006.

MÁTÉ, J. Fully Active File System. SOFSEM 2007, Current Trends in Theory and Practice of Computer Science.

MURPHY, D. TENEX and TOPS-20 Papers, *http://tenex.opost.com/*. Last accessed in December 2006.

PETERSON, A. – BURNS, R. *Ext3cow: a time-shifting file system for regulatory compliance*. ACM Transaction on Storage (TOS), Pages 190-212, May 2005.

PORTS, D.R.K. – CLEMENTS, A.T. – DEMAINE, E.D. *PersiFS: a versioned file system with an efficient representation*. ACM Symposium on Operating Systems Principles, Proceeding of the twentieth ACM symposium on Operating Systems Principles, 2005.

PRABHAKAR, S. – AGRAWAL, D. – EL ABBADI, A. – SINGH, A. *A brief survey of tertiary storage systems and research*, Pages 155-157. Symposium on Applied Computing, Proc. of the 1997 ACM symposium on Applied computing, 1997.

RIEDEL, E. – GIBSON, G.A. – FALOUTSOS, C. Active storage for largescale data mining and multimedia. In Proc. 24th Int. Conf. Very Large Data Bases, (1998) 62–73.

SCHOLERMAN, S. – MILLER, L. International Business Machines Corporation Rochester, MN, *Relational Database Integration in the IBM AS/400*. Sigmod record, Vol. 22, No. 4, December 1993.

SILBERSCHATZ, A. – Galvin, P.B. *Operating System Concepts (5th Edition)*, pages 337-369. Addison-Wesley, 1998, ISBN: 0-201-59113-8.

SRINIVASAN, S.H. – SINGH, P. Active File Systems for Data Mining and Multimedia, Satyam Computer Services Ltd Applied Research Group, 14 Langford Avenue, Bangalore, India.

SWEENEY, A. – DOUCETTE, D. – WEI HU – ANDERSON, C. – NISHIMOTO, M. – PECK, G. *Scalability in the XFS File System*. USENIX conference in San Diego, California, 1996.

TANENBAUM, A.S. *Modern Operating Systems (2nd Edition)*, pages 379-452. Prentice Hall; 2 edition, 2001, ISBN: 7-111-09156-6.

TAY, B.H. – ANANDA, A.L. *A survey of remote procedure calls*. ACM SIGOPS Operating Systems Review, Pages: 68-79, 1990.

TWEEDIE, S.C. *Journalling the ext2fs Filesystem*. LinuxExpo, 1998.

WACHSMANN, A. *Part III: AFS—A Secure Distributed Filesystem*. Linux Journal, Volume 2005 , Issue 132, April 2005.

ZADOK, E. – IYER, R. – JOUKOV, N. – SIVATHANU, G. *On Incremental File System Development*, Pages 161-196. ACM Transactions on Storage, Vol. 2, No. 2, May 2006.

5 VYUŽITIE ZNALOSTÍ Z WIKIPÉDIE PRI HĽADANÍ

Vyhľadávanie informácií na webe sa stalo bežnou činnosťou väčšiny Internetových používateľov. Takýto používatelia majú na výber viaceré vyhľadávacie systémy, ale v súčasnosti je veľmi dominantný a úspešný systém Google, ktorý používa algoritmus PageRank. Ale vývoj sa nezastavil a objavili sa nové algoritmy ako napríklad ExpertRank systému Teoma, alebo HITS algoritmus vyvíjaný v rámci projektu CLEVER.

Táto kapitola mapuje prostredie webu, systém kolaboratívnej encyklopédie Wikipédia, ďalej poskytuje prehľad súčasných vyhľadávacích technológií. Zameriavame sa na niektoré projekty, ktoré získavajú a používajú znalosti obsiahnuté v encyklopédii Wikipédia. Cieľom výskumu je možnosť použiť znalosti obsiahnuté v systéme Wikipédia pri hľadaní na webe a jeho realizácia pomocou rozšírenia algoritmu QD-PageRank.

5.1 Problémové prostredie

Táto kapitola predstavuje prostredie, ktoré pokrýva problematiku minimovej práce. Uvedené sú základné charakteristiky systému World Wide Web (ďalej len web) a kolaboratívneho prostredia Wikipédia.

5.1.1 Web

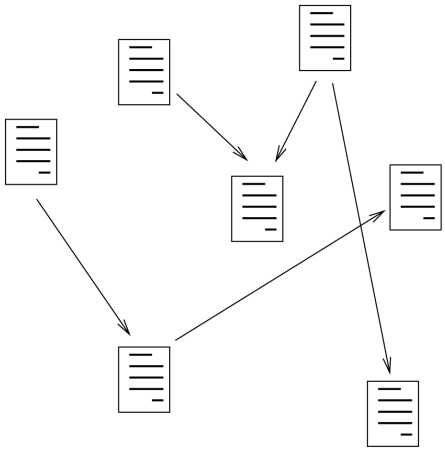
World Wide Web je systém prepojených hypertextových dokumentov prenášaných cez celosvetovú počítačovú sieť Internet. Je založený na kombinácii štyroch hlavných myšlienok:

- *Hypertext* – jedná sa o formát zápisu informácií, ktorý umožňuje používateľovi pohybovať sa medzi časťami dokumentov alebo celými dokumentmi pomocou interných spojení medzi týmito dokumentmi. Tieto spojenia sa nazývajú hyperodkazy (skrátene odkazy).
- *Jedinečné identifikácie zdrojov* (angl. *Uniform Resource Identifiers* - URIs). URI (Berners-Lee, 2005) sú používané pri lokalizovaní a identifikácii zdrojov (súborov, dokumentov alebo iných zdrojov) prístupných cez sieť.

- *Klient-server model* – V tomto komunikačnom modeli klientský softvér alebo klientské počítače posielajú dopyty serverovému softvéru alebo serverovým počítačom, ktoré poskytujú klientom zdroje alebo služby.
- *Značkový jazyk* – Značkovacie jazyky sa používajú znaky alebo kódy vložené v texte na označenie štruktúry, sémantického významu alebo prídavných informácií pre reprezentáciu.

Hypertex

Vzťah hypertextu a lineárneho textu je rozoberaný v článku (Jelínek, 1998). Hypertextom je označovaný systém pre prezentovanie informácií, ktorý umožňuje prepájať textové stránky nesekvečným, úplne všeobecným spôsobom. Štruktúra hypertextu (obrázok 5-1) je tvorená uzlami (textové stránky – objekty informácie) a orientovanými odkazmi (vzťahy medzi objektmi).



Obrázok 5-1. Štruktúra hypertextu – orientovaný graf.

Koncept hypertextu vôbec nie je nový. Usporiadanie informácií v encyklopédiách alebo slovníkoch predpočítačovej éry je svojim charakterom hypertextové. Veď aj napríklad Talmud svojimi početnými odkazmi a vetvenými komentármi sa dá označiť za prahypertextový dokument.

Hypertext, ktorý svojim charakterom umožňuje ukladanie informácií nelineárnym spôsobom, sa javí ako vhodnejšie médium pre ukladanie našich myšlienok ako klasické médiá lineárneho charakteru. My sami premýšľame v nelineárne usporiadaných zhlukoch, ktoré sa snažíme v mozgu prepojiť a vybudovať si sieť pojmov a znalostí. Keď čítame odbornú knihu, často sa pri čítaní vraciame, pozeráme dopredu a občas si pripomenie už prečítané pasáže. Niekedy si urobíme poznámku o pochopených súvislostiach, preskočíme inú časť nájdenú pomocou obsahu alebo registra. V konečnom dôsledku z lineárneho textu vytvárame obecnú štruktúru informácií, ktorá je blízka aj nášmu chápaniu a ukladaniu informácií v pamäti.

Naopak, keď pripravujeme nejaký tlačený dokument, máme na začiatku našej práce množinu viac-menej neusporiadaných myšlienok, logicky alebo sémanticky vzájomne prepojených. Hlavná činnosť pri vytváraní tlačeného dokumentu je tieto informácie usporiadať – snažíme sa informácie usporiadať do jednotného lineárneho prúdu

myšlienok, linearizovať s minimálnym počtom nelineárnych prvkov, ako sú napríklad poznámky pod čiarou, odkazy na iné miesta alebo využitie registra a obsahu.

Usporiadanie poznatkov v ľudskej pamäti je reprezentované sémantickou sieťou, v ktorej sú pojmy navzájom asociované väzbami. Hypetext sa z tohto pohľadu javí ako ideálne médium, ktoré môže prirodzeným spôsobom kopírovať usporiadanie faktov a relácií v ľudskej pamäti. Problémom je, ako transformovať sémantickú pamäťovú sieť tak, aby sa človek pri prechádzaní hypertextu nestratil.

Prevod už hotových textových dokumentov do hypertextu predstavuje nasledujúce kroky:

1. Oblasť, ktorú je možné do určitej miery automatizovať:
    - Rozdelenie textu do častí, ktoré sa stanú informačným obsahom jednotlivých uzlov. Napríklad to môže byť malá kapitola, odstavec, poznámka, príklad.
    - Základné odkazy sa dajú väčšinou odvodiť od štruktúry textového dokumentu. Tvorí ich napríklad prirodzená postupnosť kapitol, register, obsah (názvy kapitol a odstavcov)
    - Objektívne väzby, ktoré predstavujú odkazy v text na stránky alebo odstavce, citácie, odkazy na definície.
2. Oblasť, ktorú predstavujú väzby a tieto môže pridať iba autor na základe sémantiky dokumentu – ide o previazanie častí textu, ktoré spolu časovo, tematicky, protikladovo alebo vysvetľujúco súvisia.

Priame vytváranie hypertextového dokumentu, ktorý je svojim charakterom nelineárny, býva ťažšie, ako previesť iný už hotový textový dokument na hypertext. Lineárny text totiž už implicitne obsahuje nejakú štruktúru (najčastejšie hierarchickú) a je iba na autorovi hypertextu, do akej miery ju využije. Druhým problémom pri tvorbe hypertextového dokumentu je to, že autor by okrem logickej štruktúry dokumentu mal zvážiť aj navigačnú štruktúru dokumentu a jej vplyv na orientáciu čitateľa v dokumente.

Niektoré praktické zásady pre tvorbu hypertextových dokumentov:

- Informácie treba rozdeliť do malých celkov, ktoré majú spoločnú myšlienku alebo tému. Tieto celky potom predstavujú jednotlivé uzly hypertextového grafu. Odporúčaná veľkosť textu je jedna až dve obrazovky.
- Často krát autor potrebuje vytvoriť odkazy na dokumenty, ktoré ešte neexistujú. Stáva sa to najmä vtedy, keď sa postupuje pri vytváraní hypertextovej štruktúry zhora nahor (najprv sa vytvára navigačná štruktúra a až neskôr cieľové uzly). V takýchto prípadoch je dôležité vhodne voliť označenie uzlov, aby nedochádzalo k zámenám uzlov alebo ich duplicite.
- Pri vytváraní a formátovaní hypertextového dokumentu si treba uvedomiť aj základné pravidlá pre prípravu dokumentov v elektronickej podobe (UT Austin TeamWeb, 2007), lebo ich vplyv na použiteľnosť dokumentov môže byť dosť výrazný (Ivory, 2001). Nutná je dostatočná veľkosť písma, prehľadnosť dokumentu bez zbytočného kombinovania farebných a zvýrazňovacích prostriedkov. Pre rýchlejšie čítanie dokumentu je vhodné formátovať text do úzkych stĺpcov.

- Základným prehreškom voči príjemnému vnímaniu hypertextového dokumentu je závislosť informačného obsahu jedného uzla od obsahu druhého uzla. To núti čitateľa zbytočne si pamätať ďalšie informácie pri prechode medzi uzlami. Toto sa môže stať, keď v jednom uzle je kladená otázka a v druhom uzle je uvedená odpoveď bez opätovného zopakovania otázky.

Značkovacie jazyky

Jeden z prvých značkovacích jazykov bol GML (Generalized Markup Language) a používal sa pri uchovávaní právnických textov pre IBM. Princíp GML sa osvedčil a v 80. rokoch začala na jeho základe vyvíjať štandardizačná komisia ANSI jazyk, ktorý umožňoval vytvoriť si vlastnú sadu značiek, vhodnú pre daný druh dokumentu. Podobnú snahu malo aj združenie GCA a spojením týchto dvoch aktivít vznikol jazyk SGML (Standard Generalized Markup Language) (ISO, 1986).

Jazyk SGML je skutočne všeobecný – umožňuje definíciu vlastného značkovacie jazyka (sady značiek, parametrov a ich vzájomných kombinácií) pomocou definície typu dokumentu (angl. *Document Type Definition*). Samotný jazyk obsahuje veľké množstvo voliteľných parametrov ako napr. maximálna dĺžka názvu značky, znaky používané pre oddelenie textu od značiek, atď. Táto komplexnosť brzdila rozšírenie SGML. Najznámejšou podmnožinou SGML je HTML (Hypertext Markup Language), ktorý sa používa pre tvorbu webových stránok. Značky, ktoré sa na stránkach môžu používať, určuje príslušné DTD, ktoré je pre každú verziu HTML doplnené.

Jazyk HTML si získal veľkú obľubu kvôli svojej jednoduchosti, ktorá bola v kontraste s komplexnosťou SGML. Ukázalo sa však, že pevne daná sada značiek, ktoré HTML používa, už nestačí. Pre účely vyhľadávania a efektívnejšej výmeny dát by bolo lepšie mať možnosť použiť vlastné značky. Riešením by bolo znova použiť SGML, ale počas používania SGML sa ukázalo, že v praxi sa používa iba časť jeho možností. Táto najdôležitejšia podmnožina SGML bola vybraná ako nový jazyk XML (eXtensible Markup Language) (Bray, 2000). Ide o podmnožinu, ktorá si zachováva možnosť definovať vlastné DTD pre jednotlivé dokumenty. Na rozdiel od SGML je veľa parametrov pevne definovaných – maximálna dĺžka názvu značiek, použité oddeľovače, špeciálne znaky atď. Na rozdiel od predchádzajúcich počítačových štandardov nie je XML pevne zviazaný s angličtinou a priamo podporuje znakovú sadu Unicode (Unicode Consortium, 2006). Navyše je syntax zápisu dokumentov XML pomerne prísna oproti SGML, čo umožňuje lacnejší a rýchlejší vývoj aplikácií, ktoré s týmto jazykom pracujú.

Princípy značkovacieho jazyka XML sa dajú vysvetliť na nasledujúcom príklade:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE človek SYSTEM "popis.dtd">
<človek>
  <meno>Miroslav</meno>
  <adresa>
    <ulica>Vajnorská</ulica><číslo>53</číslo>
    <mesto>Bratislava</mesto>
  </adresa>
</človek>
```

Prvý riadok je XML deklaracia, ktorá určuje, akú verziu štandardu XML dokument používa, môže obsahovať aj informácie o použitom znakovom kódovaní a externých

závislostiach. Každý XML dokument sa skladá z elementov, ktoré môžu byť do seba navzájom vnorené. Napríklad element `človek` obsahuje elementy `meno`, *adresa* a tento obsahuje elementy `ulica` a `mesto`. Elementy sa v texte vyznačujú pomocou značiek. Väčšina elementov má dve značky, začiatočnú a ukončovaciu[12]. Napríklad element `<meno>Miroslav</meno>` má začiatočnú značku `<meno>` a ukončovaciu značku `</meno>`. Každá začiatočná značka musí byť spárovaná s príslušnou ukončovacou značkou, ukončovacia značka nesmie chýbať, nesmie byť navyše a elementy sa nesmú navzájom prekrižovať:

```
<h1>Prekrižovanie <i>nie</h1> je povolené.</i>
```

Elementu môžu byť priradené atribúty, tie sa zaznačujú v začiatočnej značke v tvare `meno_elementu="hodnota elementu"`. Napríklad

```
<a href="#mailinglist">Projects</a>
```

Atribúty sa obvykle používajú na spresnenie významu elementu. Pokiaľ spĺňa dokument všetky uvedené syntaktické pravidlá a všetky elementy dokumenty sú obsiahnuté v jednom koreňovom elemente (v našom príklade element `človek`), označuje sa tento dokument ako správne štruktúrovaný (angl. *well-formed*). V prípade, že potrebujeme prísnejšie kontrolovať štruktúru XML dokumentu, môžeme popísať požadovanú štruktúru dokumentu pomocou DTD:

```
<!ELEMENT človek (meno, adresa*)>
<!ELEMENT meno  (#PCDATA)>
<!ELEMENT adresa (ulica?, číslo?, mesto)>
<!ELEMENT ulica  (#PCDATA)>
<!ELEMENT číslo  (#PCDATA)>
<!ELEMENT mesto  (#PCDATA)>
```

Tento DTD dokument definuje triedu XML dokumentov, ktoré majú koreňový element `človek` obsahujúcu práve jeden element `meno` a žiaden alebo viaceré elementy `adresa`. Element `adresa` môže, ale nemusí, obsahovať nepovinný element `ulica`, nepovinný element `číslo` a musí obsahovať povinný element `mesto`. Elementy `meno`, `ulica`, `číslo`, `mesto` môžu obsahovať iba textové reťazce, žiadne ďalšie elementy.

Kontrola dokumentu podľa DTD má viaceré obmedzenia, lebo DTD nepodporuje XML menné priestory a nemá dostatočnú vyjadrovaciu silu. Existujú novšie jazyky silnejšie ako DTD, ktoré postupne nahradzujú DTD: XML Schema (Fallside, 2001), RELAX NG (Clark, 2001), Document Structure Description (Klarlund, 2000).

5.1.2 Wikipédia

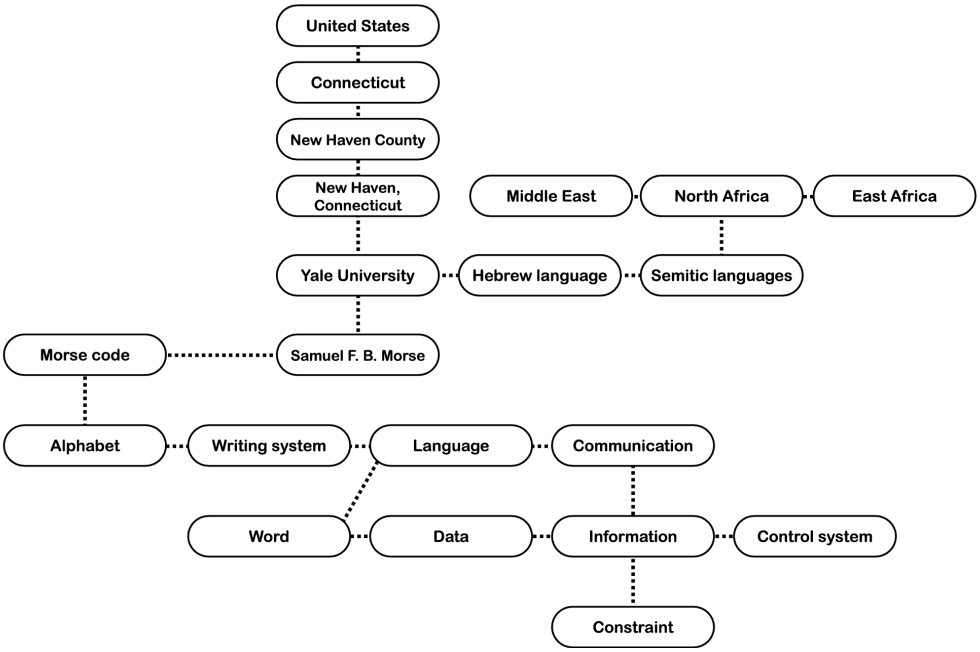
Wikipédia (Wikipedia, 2007) je viacjazyčná voľne prístupná encyklopédia prístupná na webe. Je koncipovaná ako masívne kolaboratívne prostredie spoločne tvorené dobrovoľníkmi a až na malé výnimky, môže byť upravované hocikým, kto má prístup

[12] Element, ktorý neobsahuje nič okrem začiatočnej a ukončovacej značky, môže byť prepísaný do jednej značky ukončenej znakom /. Napríklad je ekvivalentné .

k web stránke projektu. Wikipédia je spravovaná neziskovou organizáciou Wikipedia Foundation založenou Jimmom Walesom (Wales, 2005).

Svojím rozsahom predstihla všetky ostatné encyklopédie a k februáru 2007 obsahovala viac ako 6 miliónov článkov vo viac ako 250 jazykoch. Najväčšia jazyková edícia bola anglická s vyše 1,6 miliónom článkov. Wikipédia nie je tematicky zameraná alebo ohraničená, jej spektrum siaha napríklad od umenia, kultúry, histórie, geografie, cez rôzne prírodné vedy až po aktuálne správy a populárne a žurnalistické témy.

Medzi príspevkami existuje veľké množstvo vzájomných odkazov, čo umožňuje používateľovi sledovať vzťahy medzi príspevkami alebo si rýchlo prečítať podkladové informácie k príspevku. Na (obrázok 5-2) prebranom z (Wikipedia, 2007) napríklad vidieť, aké prepojenie existuje medzi príspevkami s heslom *Constraint* a heslo *United States*.



Obrázok 5-2. Prepojenia medzi príspevkami.

Wikipédia používa wiki softvér (Leuf, 2001) pre súbežné upravovanie encyklopédie viacerými autormi naraz a pre ukladanie ich zmien do databázy. Články obsiahnuté v databáze sú poprepájané pomocou krížových odkazov podobne, ako je tomu v klasických encyklopédiách. Pre organizovanie článkov podľa ich obsahu používa Wikipédia vlastný systém kategórií, ktorý je podobný systému Propædia z Encyclopædia Britannica, ale nie je úplne hierarchický. Pre poskytnutie prehľadu o užšej tematickej časti obsahuje Wikipédia portály.

Pre ochranu duševného vlastníctva prispievateľov používa Wikipédia licenciu GNU Free Documentation License (GFDL) (Free Software Foundation, 2002). Jedná sa o copyleft licenciu, ktorá ponecháva autorské práva jednotlivým prispievateľom, ale ostatným povoľuje kopírovanie, vytváranie odvodených diel a aj komerčné využitie príspevkov, pokiaľ je uvedený aj pôvodný autor a odvodené diela sú tiež pod GFDL.

V systéme Wikipédia môže ktokoľvek vytvárať a pozmeňovať príspevky – až na niektoré osobitné stránky. Pri každej zmene sú predchádzajúce verzie príspevku zachované v histórií zmien v chronologickom poradí, čo umožňuje kedykoľvek si pozrieť staršiu verziu. Filozofiou systému Wikipédia (Riehle, 2006) je, že neobmedzovaná spolupráca veľkého počtu informovaných editorov postupne vylepší encyklopédiu aj v obsahovej šírke aj v podrobnosti jednotlivých príspevkov a preto sú zmeny okamžite viditeľné na web stránke projektu, bez predchádzajúcej odbornej recenzie. Argumenty proti tejto filozofii sú viaceré, napríklad (Denning, 2005) uvádza:

- *Presnosť* – Nemôžete si byť istý, ktoré informácie sú presné a ktoré nie. Zavádzajúce informácie môžu spôsobiť finančnú stratu, aj keď boli získané zadarmo.
- *Motivácia* – Nepoznáte motivácie prispievateľov. Možno sú nesebeckí, ale môžu byť aj politicky alebo finančne motivovaní, alebo môžu byť dokonca vandali.
- *Nejasná odbornosť* – Niektorí prispievatelia môžu presiahnuť svoju odbornosť a uvádzať dohady, špekulatívne informácie alebo nesprávne informácie. Keďže prispievatelia sú v histórii zmien v príspevku uvádzaní pod svojimi pseudonymami, je ťažké overiť ich dôveryhodnosť.
- *Premenlivosť* – Príspevky alebo opravy môžu byť v budúcnosti vymazané následnými editormi. Premenlivosť komplikuje situáciu pri citovaní informácií – napríklad mala by sa citovať prečítaná verzia príspevku (čím tí, ktorí budú sledovať túto citáciu, prídu o opravy a iné vylepšenia príspevku) alebo najnovšia verzia (ktorá sa už môže výrazne líšiť).
- *Pokrytie* – Tématika príspevkov vo väčšine prípadov reprezentuje záujmy a znalosti prispievateľov. Nejde o presne zvolenú skupinu reprezentujúcu všetky ľudské znalosti a preto sú mladé alebo Internetovo-orientované témy dobre popísané a témy, ktoré boli v „pred-Internetovom" období, slabšie popísané.
- *Zdroje* – Veľa príspevkov neuvádza žiadne nezávislé zdroje. Málo príspevkov (hlavne kratšie príspevky) obsahuje citácie na zdroje, ktoré nie sú digitalizované a voľne dostupné na Internete.

Ukazuje sa (Rosenzweig, 2006), že presnosť Wikipédie je približne rovnaká, ako v prípade klasických encyklopédií a následky vandalizmu sú prevažne krátkodobé (Viegas, 2004). Ako sa Wikipédii podarí vyrovnať sa s ostatnými problémami, ukáže čas, jednoznačne Wikipédia rastie veľmi rýchlo. Motivácia prispievať je u každého iná, ale článok (Kuznetsov, 2006) uvádza ako najdôležitejšie dôvody: altruizmus – charitatívnosť a ochota venovať čas, reciprocita – ten, kto prispieva, očakáva príspevky aj od ostatných, komunita – väčšina príspevkov vzniká pri vzájomnej komunikácii a veľa ľudí vyhľadáva práve túto možnosť diskutovať a spolupodielať sa, reputácia – registrovaní autori majú možnosť dosiahnuť rešpekt, dôveru a ocenenie za ich nasadenie a námahu, anonymita – sloboda nezávislého rozhodovania, možnosť úniku z prísne regulovaného, nariadeniami, záväzkami a zákonmi kontrolovaného sveta.

Projekt Wikipédia

Používateľským rozhraním k projektu Wikipédia je web stránka projektu (obrázok 5-3). Na tejto stránke sú v časti *Navigation* uvedené skratky na hlavnú prístupovú stránku, stránku portálu a ďalšie dôležité stránky. Časť Toolbox obsahuje hlavné spojovacie informácie – informácie o tom, ktoré iné príspevky sa odkazujú na aktuálny príspevok,

informácie o tom, ako citovať aktuálny príspevok atď. S každým príspevkom je spojená osobitná stránka vyhradená pre diskusiu o príspevku (obrázok 5-4), editori tu môžu vyjadrovať svoj názor na jednotlivé časti príspevku, riešiť sporné body a konfliktné situácie. Diskusia má formu rozvetveného stromu, ktorý vzniká odpoveďami editorov na predchádzajúce komentáre.

História zmien príspevku (obrázok 1-5) je lineárna postupnosť časových záznamov, každý obsahuje danú verziu príspevku, dátum zmeny príspevku, autora zmeny a krátky komentár ku zmene. Pomocou histórie je možné si pozrieť ľubovoľné staršie verzie[13] a ľahko odstraňovať následky vandalizmu.

MediaWiki

MediaWikie je program napísaný v skriptovacom jazyku PHP, ktorý spolu s MySQL alebo PostgreSQL relačnou databázou vytvára prostredie pre projekt Wikipédia.

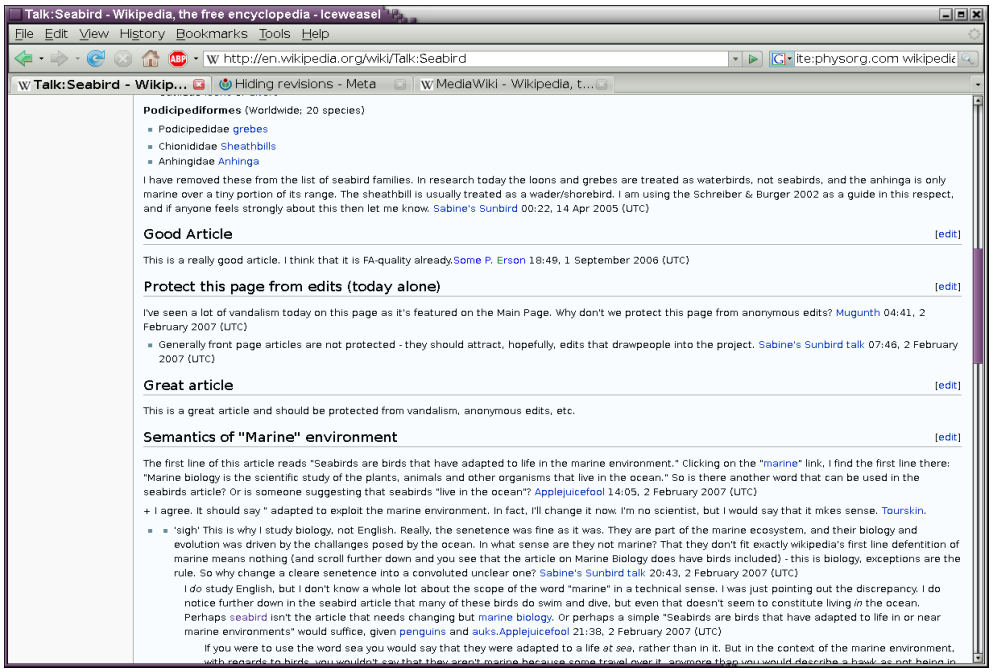
Podobne, ako v iných wiki systémoch, MediaWiki neposkytuje WYSIWYG prostredie, formátovanie stránok sa robí pomocou jednoduchého značkovacieho jazyka.

Najdôležitejšie prvky, ktoré tento jazyk obsahuje, sú:

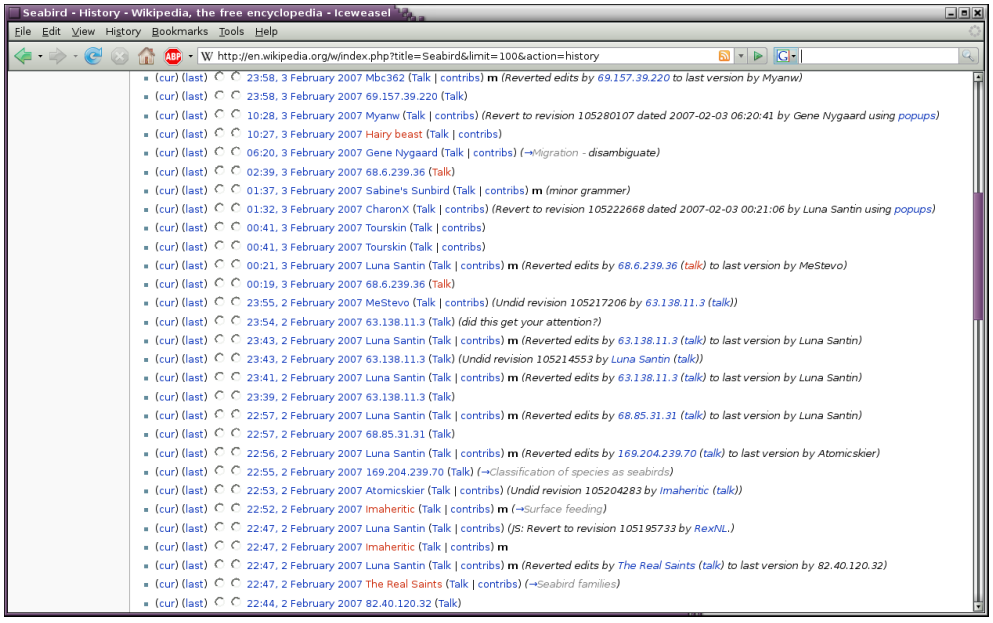
- Zvýrazňovanie textu pomocou skloneného písma a tučného písma.
- Rozdeľovanie stránky na kapitoly, odseky, pododseky spolu s automatickým generovaním obsahu podľa tohto delenia.

Obrázok 5-3. Stránka anglického príspevku o morských vtákoch v projekte Wikipedia.

[13] Verzie, ktoré obsahovali osobné údaje alebo autorsky chránené materiály, sa dajú skryť tak, aby neboli voľne prístupné v histórii zmien.



Obrázok 5-4. Diskusia o príspevku.



Obrázok 5-5. História zmien príspevku.

- Neusporiadané a očíslované zoznamy.
- Tabuľky. Tieto tabuľky sú analogické HTML tabuľkám a na rozdiel od tabuliek v relačných databázach, ktoré podporujú dátové typy, všetky položky sú textové reťazce.

- Podpora pre obrázky a iné druhy multimediálnych súborov. Tieto súbory sa ďalej dajú organizovať do galérií.
- Šablóny. Tieto umožňujú oddeliť často sa opakujúci text do osobitných stránok. Šablóny sa dajú parametrizovať.
- Odkazovanie medzi stránkami, v rámci jednej stránky, odkazy na stránky mimo MediaWiki, poznámky pod čiarou.
- Menné priestory – používajú sa na rozdelenie stránok do skupín. Napríklad šablóny sa nachádzajú v mennom priestore Template, stránky s diskusiou o príspevkoch v priestore Talk, stránky s návodom na používanie MediaWiki v priestore Help.

5.2 Pojmy – modely – metódy

V tejto kapitole sú rozoberané vyhľadávacie systémy a teoretické modely, na ktorých sú založené.

5.2.1 Vyhľadávacie systémy

Vyhľadávanie a získavanie informácií na webe má svoje problémy (Huang, 2000), z ktorých najdôležitejšie sú:

- *Veľkosť webu* – veľkosť webu sa odhadovala vo februári 2005 na približne 11,5 miliárd stránok (Gulli, 2005).
- *Dynamickosť webu* – web sa mení každý deň, zatiaľ čo väčšina vyhľadávacích systémov pracuje so stránkami, ktoré sú staré niekoľko dní až týždňov.
- *Heterogénnosť* – web obsahuje široké spektrum typov dokumentov: text, hypertext, obrázky, zvukové súbory, animovaný obraz, atď.
- *Variabilita v jazykoch* – za pomoci štandardu Unicode sú vytvárané stránky v takmer všetkých používaných a niektorých mŕtvych jazykoch.
- *Duplicita* – kopírovanie je ďalšou dôležitou stránkou webu, odhaduje sa, že približne 30% všetkých stránok sú duplikáty.
- *Veľa prepojení* – každý dokument má v priemere viac ako 8 odkazov na iné stránky.
- *Nesprávne sformované dopyty* – vyhľadávacie systémy musia vedieť obslúžiť krátke a nepresné odkazy používateľov.
- *Veľká variabilita používateľov* – každý používateľ má svoje vlastné potreby, predpoklady a znalosti.
- *Špecifické správanie* – odhaduje sa, že vyše 85% používateľov si pozerá iba prvú stránku s výsledkami, ktoré sú vrátené vyhľadávacím systémom. 78% všetkých používateľov nemení svoj prvý dopyt.

O architektúre a fungovaní veľkých webových vyhľadávacích systémov nebolo publikované veľa informácií, lebo väčšina týchto služieb funguje na komerčnej báze a tieto informácie predstavujú prísne strážené tajomstvo. Výnimkou je vyhľadávací

systém Google (Brin, 1998), ktorý bol zo začiatku koncipovaný ako akademický webový vyhľadávací systém.

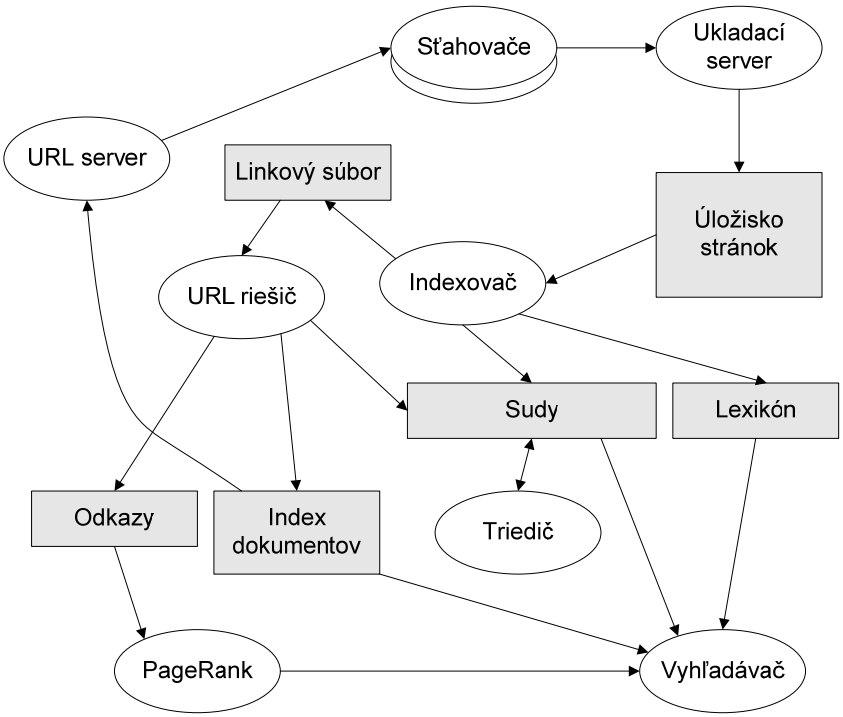
Vyhľadávací systém sa skladá z troch častí: indexovača, sťahovača a dopytovacieho servera. Sťahovač zbiera stránky z webu. Indexovač spracúva stiahnuté stránky a reprezentuje ich ako dátové štruktúry vhodné pre efektívne vyhľadávanie. Dopytovací server prijíma požiadavky od používateľa a vracia stránky identifikované pomocou vyhľadávacích dátových štruktúr.

V systéme Google sú tieto pri časti rozdelené a vysoko paralizované medzi sadu serverov tak, ako je znázornené na (obrázok 5-6). V systéme je sťahovanie rozdelené medzi viaceré distribuované sťahovače. Zoznam adries, ktoré majú byť stiahnuté, sa do sťahovačov dostane z URLServera. Každá stránka je po stiahnutí poslaná do ukladacieho servera, tento danú stránku skomprimuje pomocou DEFLATE kompresného algoritmu (Deutsch, 1996) a takto ju uloží do úložiska. Každá web stránka má priradené identifikačné číslo označované ako docID, toto je vygenerované zakaždým, ako je získaná nová URL z webstránky. Indexovacia funkcia je realizovaná pomocou indexovača a triediča. Indexovač vykonáva tieto kroky: prechádza cez dokumenty uložené v úložisku, dekomprimuje ich a vykonáva ich syntaktickú analýzu. Každý dokument je prevedený na množinu výskytov slov nazývaných zásahy. Zásah obsahuje slovo, jeho polohu v dokumente, približnú veľkosť písma a informáciu o tom, či sa skladá z veľkých písmen alebo nie. Indexovač rozdeľuje tieto zásahy medzi sadu „sudov``, čím vytvára čiastočne usporiadaný dopredný index. Okrem toho indexovač identifikuje odkazy vo všetkých web stránkach a ukladá informácie o nich do linkového súboru. Tento obsahuje všetky potrebné informácie pre určenie toho, odkiaľ a kam daný odkaz ukazuje a aký je text odkazu.

URLriešič (angl. *URLresolver*) číta linkový súbor a prevádza relatívne URL adresy na absolútne URL adresy a tie následne na docID. Ukladá text odkazu spolu docID na, ktoré odkaz ukazuje, do dopredného indexu. Taktiež generuje databázu odkazov reprezentovaných ako páry docID. Táto databáza je používaná pri výpočte PageRank všetkých dokumentov.

Triedič zoberie obsah sudov, ktorý je zoradený podľa docID, znovu ho zoradí podľa wordID, či vytvorí inverzný index. Toto triedenie sa robí na mieste tak, aby pre túto operáciu bolo nutné iba malé množstvo dodatočnej pamäte. Následne program DumpLexicon zoberie tento zoznam a spolu s lexikónom vygenerovaným indexovačom vytvorí nový lexikón, ktorý používa vyhľadávač. Vyhľadávač je spúšťaný webovým serverom a používa lexikón vytvorený programom DumpLexicon spolu s inverzným indexom a hodnotením vytvoreným pomocou PageRank algoritmu pre zodpovedanie dopytov kladených používateľmi.

Opis kompletného vyhľadávacieho systému WebCrawler je uvedený v dizertačnej práci (Pinkerton, 2000). Tento systém bol prvý plno-textový vyhľadávací systém a pôvodne bol navrhovaný ako samostatný vyhľadávací stroj s vlastnou databázou. Neskôr bol upravený na meta-vyhľadávací stroj, ktorý poskytuje kombináciu oddelene identifikovaných platených a neplatených výsledkov hľadania z najobľúbenejších vyhľadávacích systémov.



Obrázok 5-6. Architektúra vyhľadávacieho systému Google.

5.2.2 Osobné a zamerané webové pavúky

Webový pavúk je program, ktorý prechádza informačným priestorom webu, pričom sleduje hypertextové odkazy a sťahuje webové dokumenty pomocou HTTP protokolu. Do tejto kategórie spadajú aj meta-vyhľadávacie pavúky (pavúky, ktoré sa pripájajú k iným vyhľadávacím službám a kombinujú ich výsledky). Iné webové roboty ako napr. shopboty, chatboty a chatterboty sa vo všeobecnosti nepovažujú za pavúky. V súčasnosti sa dá výskum webových pavúkov klasifikovať do týchto kategórií:

- *Rýchlosť a efektívnosť.*

  Táto kategória výskumu sa zameriava na zvyšovanie rýchlosti sťahovanie pavúkov. Projekty tejto kategórie sú zamerané na stavbu rýchlych pavúkov, ktoré sa dajú rozširovať aj na veľké zbierky dokumentov pomocou optimalizácií operácií ako napr. vstupno-výstupné procedúry, preklad IP adries.

- *Pravidlá prehľadávanie.*

  Výskum v tejto kategórii sa zaoberá správaním pavúkov a ich vplyvom na ostatné systémy a WWW ako celok. Správne navrhnutý pavúk by nemal preťažovať webové servere. V súčasnosti existujú dva štandardy pre ovplyvňovanie správania pavúkov autormi stránok: určovanie stránok, ktoré nemá pavúk sťahovať, pomocou súboru *robots.txt* a určovanie toho, či sa má stránka používať v pavúkovi pre indexáciu alebo získavanie odkazov pomocou META značky. Napriek tomu, že tieto štandardy nie sú povinné, väčšina pavúkov ich dodržiava a riadi podľa nich svoje správanie.

- *Získavanie informácií.*

  Väčšia časť výskumu ohľadom pavúkov sa sústreďuje na túto oblasť. V rámci nej sa skúmajú rôzne algoritmy a heuristiky tak, aby pavúk získal požadované informácie z webu čo najefektívnejšie.

Štyri hlavné použitia pre webové pavúky sú:

- *Osobné vyhľadávanie*

  Osobné pavúky sa snažia hľadať webové stránky zaujímavé pre používateľa. Keďže tieto pavúky sa väčšinou vykonávajú na klientskom počítači majú k dispozícii väčší výpočtový výkon a poskytujú viac možnosti ako pavúky umiestnené na servery.

- *Vytváranie zbierok*

  Webové pavúky sú používané pre budovanie veľkých zbierok webových stránok, ktoré sa používajú pre výpočet indexov každej vyhľadávacej služby. Ďalej sa tieto zbierky dajú použiť pri tvorbe lexikónov slov alebo zbieraní emailových adries.

- *Archivácia*

  Niektoré projekty sa snažia robiť archiváciu určitých stránok, prípadne celého webu[14].

- *Webové štatistiky*

  Z veľkého množstva webových stránok zozbieraných pomocou pavúkov sa dajú získať užitočné štatistiky o webe, ako napríklad priemerná dĺžka HTML dokumentu, priemerný počet odkazov, priemerný počet odkazov na neexistujúce stránky a ďalšie.

Analýza obsahu a štruktúry webu

Rôzne techniky analýz používaných v pavúkoch sa dajú rozdeliť do dvoch hlavných kategórií: analýzy založené na obsahu a analýzy založené na odkazoch.

Pri analýzach založených na obsahu sa pre odvodenie informácií o stránke analyzuje samotný HTML kód stránky. Napríklad z textového tela stránky sa dá zistiť, či je stránka relevantná pre hľadanú doménu. Pre získanie kľúčových konceptov popísaných na stránke sa dajú použiť indexovacie techniky. Ďalšie zlepšenie analýzy sa dá dosiahnuť použitím doménových znalostí.

Dôležitým zdrojom informácií o stránke je aj jej URL adresa, ktorá identifikuje adresu servera, doménové zaradenie web servera (stránky v doméne gov môžu byť považované za dôležitejšie ako stránky v doméne *com*), hĺbku umiestnenia v adresárovej štruktúre web servera (dôležité stránky nebývajú umiestnené hlboko v adresárovej štruktúre).

Druhá skupina analýz využíva pre získanie informácií odkazy medzi stránkami. Základný predpoklad použitý pri týchto analýzach je, že ak autor web stránky A umiestnil odkaz na web stránku B, tak si myslí, že stránka B je relevantná alebo podobná stránke A a je kvalitná. Pre odkaz, ktorý ukazuje na danú stránku, sa používa termín *vstupný odkaz*. Vo všeobecnosti platí, že o čo je stránka lepšia, o to viac má vstupných odkazov. Naviac je vhodné priraďovať stránkam, na ktoré sa odkazujú iné

[14] The Internet Archive, http://www.archive.org

kvalitné stránky, väčšiu prioritu. Takto sa správajú najznámejšie dva algoritmy pre analýzu odkazov PageRank (Brin, 1998) a HITS (Kleinberg, 1998).

Grafové prehľadávacie algoritmy

Tradičné grafové algoritmy sa skúmajú aj v odbore informatiky. Keďže na web sa dá pozerať ako na orientovaný graf s množinou uzlov (stránok) prepojených orientovaných hranami (odkazmi), dajú sa niektoré grafové algoritmy použiť vo webových aplikáciách.

Prvá kategória grafových prehľadávacích algoritmov obsahuje jednoduché algoritmy, ako napríklad hľadanie do šírky a hľadanie do hĺbky. Tieto algoritmy sú označované ako neinformované hľadanie, lebo nepoužívajú pri hľadaní žiadne dodatočné informácie. Prehľadávanie do šírky, ktoré najprv načíta stránky na jednej úrovni a potom ide na ďalšiu úroveň, je vo webových pavúkoch najpoužívanejšou technikou.

Druhá kategória grafových algoritmov je informované hľadanie. O každom prehľadávanom uzle existuje dodatočná informácia, ktorá sa môže použiť ako heuristika pri riadení prehľadávania grafu. Často používaným príkladom informovaného hľadania je algoritmus best-first, v tomto algoritme existuje heuristická funkcia *f(n)*, ktorá vo všeobecnosti môže závisieť od uzla *n*, cieľa hľadania, informácií, ktoré sa zozbierali po tento bod hľadania a ľubovolných dodatočných znalostí o probléme. Best-first funguje ako rozšírenie prehľadávania do šírky, kde sa vždy expanduje ten uzol, ktorý má najvyššie ohodnotenie pomocou heuristickej funkcie.

Typická implementácia algoritmu používa prioritnú frontu. Vo webových pavúkoch sa pre heuristiky informovaného hľadania používajú rôzne metriky, napríklad počet vstupných odkazov, PageRank skóre, početnosť kľúčových slov, podobnosť s hľadaným dopytom.

Ďalšou kategóriou je paralelné hľadanie. Tieto algoritmy sa snažia spracovať rôzne časti vyhľadávacieho priestoru paralelne. Autori článku (Chen, 1995) uvádzajú ako príklad pre paralelné hľadanie paralelnú aktivitu v umelých neurónových sieťach a genetické algoritmy, ale tieto dva príklady sú dosť vzdialené od klasických grafových algoritmov. Napriek veľkým možnostiam týchto algoritmov a použitiu v tradičných aplikáciách pre získavanie informácií, sa tieto algoritmy nepresadili vo webových aplikáciách.

Webové pavúky pre osobné hľadanie

Veľa webových pavúkov bolo vyvinutých pre pomoc jednotlivým používateľom pri vyhľadávaní užitočných informácií na webe. Keďže väčšinou tieto pavúky bežia na klientských počítačoch, majú k dispozícii pre vyhľadávací proces väčší výpočtový výkon a pamäť. Taktiež tieto nástroje poskytujú používateľovi väčšie možnosti pri riadení a personalizácii vyhľadávacieho procesu.

Najznámejším skorým príkladom osobného webového pavúka je program tueMosaic (De Bra, 1994). V programe tueMosaic používateľ zadal kľúčové slová, šírku a hĺbku hľadania a spustil pavúka, aby začal sťahovať stránky zo zadanej štartovacej stránky. Program používa „fish search`` algoritmus, čo je pozmenený „best-first`` vyhľadávací algoritmus.

Neskôr vývoj pavúkov prebiehal rôznymi smermi, napríklad TkWWW robot bol vyhľadávač integrovaný do prehliadača TkWWW, pavúk SPHINX (Miller, 1998) robil

prehľadávanie do šírky a zobrazoval výsledok ako dvojrozmerný graf, CI Spider vykonával jazykovú analýzu a zhlukovanie nájdených výsledkov.

V iných štúdiách používajú pavúky v procese hľadania pokročilejšie algoritmy, napríklad Itsy Bitsy Spider (Chen, 1998) hľadá pomocou best-first prehľadávania a genetického algoritmu. Každé URL je modelované ako jedinec počiatočnej populácie, kríženie je definované ako výber URL, na ktoré ukazujú začiatky viacerých URL. Mutácia je modelovaná pomocou načítania náhodného URL z adresára Yahoo. Pretože genetický algoritmus je optimalizačný proces, je vhodný pre hľadanie stránok na webe podľa určitých kritérií. V ďalších pavúkoch sa používalo simulované žíhanie, naivný bayesovský klasifikátor a ďalšie algoritmy.

Osobitnou kategóriou pavúkov sú meta-pavúky, programy, ktoré sa pripájajú na iné vyhľadávacie stroje a sťahujú z nich výsledky. Kombinovaný výsledok poskytuje veľakrát pokrytie danej oblasti. MetaCrawler bol prvým meta-pavúkom. Poskytuje jednotné rozhranie k výsledkom z vyhľadávacích strojov Google, Yahoo! Search, MSN Search, Ask Jeeves, About, MIVA, LookSmart a ďalších. Meta-pavúk Dogpile poskytuje vyhľadávacie služby pre služby web, Usenet, žlté stránky, obrázky.

Nedávno boli vyvinuté vyhľadávacie pavúky pre peer-to-peer technológie. Napríklad JXTA Search (Waterhouse, 2002) používa ako svoj základ sieť Gnutella.

Prípadová štúdia

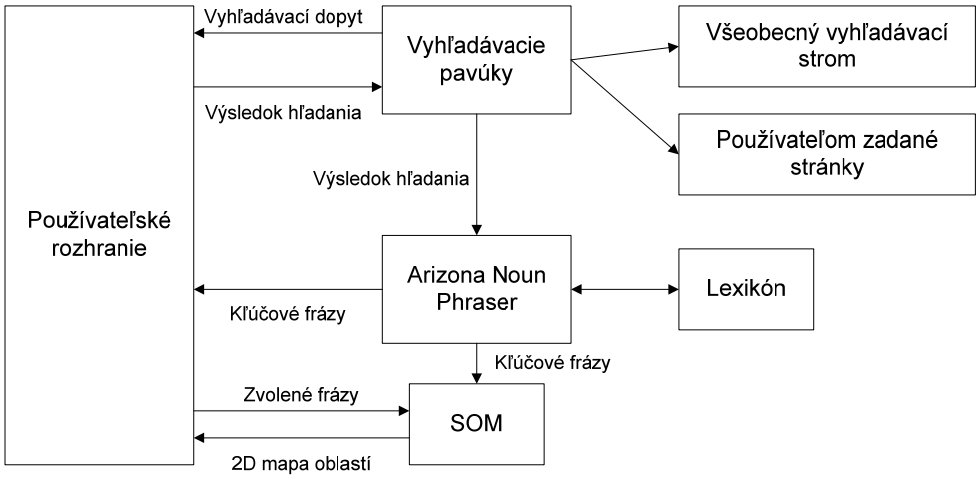
V tejto časti bude popísaná architektúra dvoch vyhľadávacích agentov rozšírených o dodatočnú analýzu stiahnutých stránok. Prvý agent Competitive Intelligence Spider (Chen, 2002) (skrátene CI Spider) je vyhľadávací agent, ktorý zbiera stránky v reálnom čase z používateľom špecifikovaných web serverov a vykonáva nad nimi indexovanie a kategorizáciu tak, aby poskytol celkový prehľad daného web servera. Druhý nástroj je Meta Spider (Chen, 2001). Ten funguje podobne ako CI Spider, ale namiesto prehľadávania konkrétnej stránky do šírky sa pripája na vyhľadávací stroj na internete a integruje takto získané výsledky.

Na obrázku 5-7 je zobrazená spoločná architektúra pre CI Spider a Meta Spider. Štyri hlavné časti sú používateľské rozhranie, internetový pavúk, *Arizona noun phaser*, samorganizujúca mapa (SOM). Tieto časti spolupracujú pri hľadaní na webe a na analýze. *Arizona noun phaser* (Tolle, 2000) je indexovací nástroj pre indexovanie kľúčových fráz, ktoré sa nachádzajú v každom dokumente zozbieranom z webu pomocou internetových pavúkov.

Nástroj vyberá z dokumentov na základe jazykového značkovania a jazykových pravidiel frázy obsahujúce podstatné mená. SOM používa umelú neurónovú sieť pre automatické zhlukovanie webových stránok do rôznych oblastí dvojrozmernej mapy. Každý dokument je reprezentovaný pomocou vstupného vektora kľúčových slov a následne je vytvorená výstupná dvojrozmerná mriežka. Po natrénovaní siete sú do nej posielané dokumenty a tieto sú rozdelené do zhlukov. Ku každej oblasti je pridelené pomenovanie, čo je výraz, ktorý najpresnejšie reprezentuje zhluk dokumentov v danej oblasti. Dôležitejšie koncepty pokrývajú väčšiu oblasť a podobné koncepty sú zoskupené blízko seba. Výsledná mapa je zobrazená v používateľskom rozhraní, pričom si používateľ môže pozrieť dokumenty v jednotlivých oblastiach.

Pre porovnanie vlastností agentov CI Spider a Meta Spider boli vykonané dva experimenty, ktorých sa zúčastnilo 30 osôb. V prvom experimente bol porovnávaný CI Spider so službou Lycos a manuálnym vyhľadávaním na serveri. Výsledky

experimentu ukázali, že aj presnosť aj množstvo vracaných stránok pre CI Spider boli podstatne lepšie ako v prípade služby Lycos pri 5% úrovni dôležitosti.



Obrázok 5-7. Architektúra agentov CI Spider a Meta Spider.

V druhom experimente bol porovnávaný agent Meta Spider so službami MetaCrawler a NorthernLight. Čo sa týka presnosti, Meta Spider dosahoval podstatne lepšie výsledky ako každý z nich a hlavne v porovnaní so službou NorthernLight. Množstvo stránok vracaných z agenta Meta Spider bolo porovnateľné so službou MetaCrawler a bolo lepšie, ako v prípade NorthernLight.

Hlavný dôvod pre lepšiu presnosť agentov bola ich schopnosť stiahnuť a skontrolovať obsah každej stránky v reálnom čase. To zabezpečuje, že každá zobrazená stránka obsahuje hľadané slová. Služby ako Lycos a NorthernLight pracujú pomocou indexov a tie môžu byť zastaralé. Vysoké množstvo vracaných výsledkov pre CI Spider sa dá zdôvodniť jeho hromadným sťahovaním stránok.

5.2.3 Algoritmy

PageRank

Algoritmus PageRank bol navrhnutý L. Pagom a S. Brinom v (Page, 1998). Podobný prístup navrhol aj Yanhong Li v (Li, 1998). Využíva sa tu podobný princíp, ako sa používa na hodnotenie akademických článkov podľa počtu citácií. Tu určuje počet citácií približnú kvalitu alebo zaujímavosť článku. PageRank rozširuje túto myšlienku o to, že pri spočítavaní nie sú všetky odkazy započítavané s rovnakou váhou a počet odkazov na stránku je normalizovaný.

Samotný výpočet je definovaný nasledovne: Keď máme stránku A, na ktorú ukazujú stránky T_1, T_2, ..., T_n a $C(T)$ je počet odkazov zo stránky T. Potom PageRank pre stránku A je:

$$PR(A) = (1-d) + d\sum_{i=1}^{n}\left(\frac{PR(T_i)}{C(T_i)}\right)$$

Tlmiaci faktor *d* je väčšinou nastavený na hodnotu 0,85. PageRank predstavuje pravdepodobné rozdelenie cez všetky počítané stránky a súčet PageRank pre všetky stránky je rovný 1. Fungovanie algoritmu sa dá pochopiť ako model správanie používateľa. Tento náhodný používateľ dostane na začiatku náhodnú stránku, následné kliká na odkazy, čím sa presúva na ďalšie stránky, nikdy nejde späť. Keď ho toto klikanie prestane baviť, vyberie si ďalšiu náhodnú stránku a znovu začne klikať. Tlmiaci faktor *d* predstavuje pravdepodobnosť, s ktorou prestane náhodný používateľ klikať na odkazy a vyberie si náhodnú stránku.

PageRank *PR(A)* pre konkrétnu stránku *A* je pravdepodobnosť, s akou sa náhodný používateľ dostane na stránku *A*. PageRank je ekvivalentný výpočtu vlastného vektora prechodovej matice *Z*.

$$Z = (1-d)\left[\frac{1}{N}\right]_{N\times N} + dM$$

kde $M_{ji} = \frac{1}{C}(T_i)$ pokiaľ existuje prepojenie z *i* do *j*, inak $M_{ji} = 0$

N je celkový počet odkazov v databáze. Jeden výpočet predchádzajúceho vzorca pre *PR(A)* je ekvivalentný výpočtu $x^{t+1} = Zx^t$, kde $x_j^t = PR(j)$ v iterácií *t*. Po dosiahnutí konvergencie (podľa (Page, 1998) stačí pre databázu s 322 miliónmi odkazov 52 iterácií) platí $x^{t+1} = x^t$ alebo $x^{t+1} = Zx^t$, teda x^t je vlastný vektor matice *Z*. Navyše keďže *Z* je normalizovaná matica, má *x* vlastnú hodnotu 1.

HITS

Algoritmus HITS (Kleinberg, 1998) navrhol J. Kleineberger a bol použitý v vyhľadávacom stroji Clever od IBM. Po zadaní dopytu nájde HITS dobré zdroje informačného obsahu (pomenované ako autoritatívne stránky) a dobré zdroje odkazov (pomenované ako uzlové stránky). Autoritatívne stránky majú veľký počet vstupujúcich odkazov. Uzlové stránky sú stránky, ktoré spájajú dohromady autority na danú tému a umožňujú vyradiť nerelevantné stránky s veľkým počtom vstupujúcich odkazov (napríklad stránky z adresáru Yahoo!). Autoritatívne a uzlové stránky majú vzájomne sa posilňujúci vzťah, lebo dobrá uzlová stránka odkazuje na veľa dobrých autoritatívnych stránok a dobrá autoritatívna stránka je odkazované z veľa uzlových stránok.

Vo všeobecnosti je cieľom vyhľadávacieho algoritmu nájsť množinu stránok S_σ pre dopyt σ, ktorá by mala nasledovné vlastnosti:

- S_σ je relatívne malá množina
- S_σ je bohatá na stránky relevantné pre danú tému
- S_σ je obsahuje väčšinu alebo veľa najdôležitejších autoritatívnych stránok

Tým, že množina S_σ je malá, dá sa na ňu aplikovať aj výpočtovo náročný algoritmus. Druhá podmienka zabezpečuje, že autoritatívne stránky budú odkazované z množiny S_σ s veľkou pravdepodobnosťou.

Pre hľadanie množiny S_σ Kleineberger navrhuje nasledujúce riešenie. Pre parameter *t* (zvyčajne 200), HITS zozbiera *t* najvyššie hodnotených stránok pre dopyt σ z textového vyhľadávacieho stroja, ako napríklad Google alebo AltaVista. Týchto *t*

stránok je označovaných ako koreňová množina R_σ. HITS rozširuje počet dôležitých autoritatívnych stránok v podgrafe pomocou expandovania množiny R_σ o odkazy, ktoré do nej ukazujú alebo ju opúšťajú.

Algoritmus je takýto:

Podgraf(σ, *e*, *t*, *d*)

σ: dopyt
e: textový vyhladávací stroj
t, *d*: prirodzené čísla

Nech R_σ označuje vrchných *t* výsledkov z *e* pre dopyt σ

Nastav $S_\sigma := R_\sigma$

Pre každú stránku $p \in R_q$

Nech $\Gamma^+(p)$ označuje množinu stránok, na ktoré odkazuje stránka *p*
Nech $\Gamma^-(p)$ označuje množinu stránok, na ktoré odkazuje stránka *p*

Pridaj všetky stránky v $\Gamma^+(p)$ do S_σ.

Ak $\left|\Gamma^-(p)\right| \leq d$ tak

Pridaj všetky stránky v $\Gamma^-(p)$ do S_σ.

Inak

Pridaj ľubovoľných *d* stránok zo $\Gamma^-(p)$ do S_σ.

Koniec cyklu
Vráť S_σ

Autori HITS ďalej zavádzajú delenie odkazov na dva druhy: na priesečné a na vnútorné. Priesečné odkazy sú medzi stránkami s rôznymi doménovými menami a vnútorné odkazy sú medzi stránkami s rovnakými doménovými menami. Všetky vnútorné odkazy sú vymazané z grafu $G[S_\sigma]$, výsledný graf G_σ obsahuje iba priesečné odkazy. Pomocou nasledujúceho algoritmu sa dajú určiť autoritatívne a uzlové stránky z G_σ.

Iteratívny algoritmus. HITS priraďuje nezápornú *autoritatívnu váhu* $x^{<p>}$ a nezápornú *uzlovú váhu* $y^{<p>}$. Váhy oboch typov sú normalizované tak, aby súčet ich druhých mocnín bol 1. Vzájomne sa posilňujúci vzťah sa dá vyjadriť nasledovne: ak *p* ukazuje na veľa stránok s veľkými *x* hodnotami, malo by dostať aj veľkú *y* hodnotu, ak *p* je odkazované z veľa stránok s veľkými *y* hodnotami, malo by dostať veľkú *x* hodnotu. Keď sú dané váhy $x^{<p>}$, $y^{<p>}$, operácia *I* upraví *x* váhy nasledovne:

$$\sum_{q:(q,p)\in E} y^{\langle q\rangle} \rightarrow x^{\langle p\rangle}$$

Operácia *O* upraví *z* váhy takto:

$$\sum_{q:(p,q)\in E} x^{\langle q\rangle} \rightarrow y^{\langle p\rangle}$$

Samotný iteratívny algoritmus:

Iteruj(G,k)

G: množina n poprepájaných stránok
k: prirodzené čísla
Nech z označuje vektor $(1,1,1,\ldots,1) \in R^n$
Nastav $x_0 := z$
Nastav $y_0 := z$
Pre i = 1, 2, ..., k
  Použi I operáciu na (x_{i-1}, y_{i-1}), čím vzniknú nové váhy x_i'
  Použi O operáciu na (x_i', y_{i-1}), čím vzniknú nové váhy y_i'
  Normalizáciou x_i' vypočítaj x_i
  Normalizáciou y_i' vypočítaj y_i
Koniec cyklu
Vráť (x_k, y_k)

Najlepšie autoritatívne stránky a najlepšie uzlové stránky sú stránky, ktoré majú najlepšie autoritatívne a uzlové váhy po skončení tohto algoritmu.

Vylepšenie algoritmu PageRank

V článku (Richardson, 2002) bol skúmaný model inteligentného používateľa, ktorý prechádza z jednej stránky na druhú, podľa obsahu stránky a podľa dopytu, ktorý používateľ hľadá. Výsledné pravdepodobnostné rozdelenie stránok je:

$$P_q(j) = (1-\beta)P'_q(j) + \beta \sum_{i \in B_j} P_q(i) P_q(i \to j)$$

kde $P_q(i \to j)$ je pravdepodobnosť, že používateľ prejde na stránku j, keď sa na stránke j a hľadá podľa dopytu q. $P'_q(j)$ určuje stránky, na ktoré používateľ skočí, keď nenasleduje žiaden odkaz. β je tlmiaci faktor. B_j je množina stránok, ktoré odkazujú na j. $P_q(j)$ je potom výsledné pravdepodobnostné rozdelenie stránok a toto zodpovedá dopytovo závislému PageRank skóre (skrátene QD-PageRank).

Podobne, ako v prípade PageRank, sa QD-PageRank počíta pomocou iteratívneho počítania predchádzajúceho vzorca a je ekvivalentný vlastnému vektoru prechodovej matice Z_q, kde:

$$Z_{q_{ji}} = (1-\beta)P'_q(j) + \beta \sum_{i \in B_j} P_q(i) P_q(i \to j)$$

$P'_q(j)$ a $P_q(i \to j)$ môžu byť ľubovoľné pravdepodobnostné rozdelenia, ale autori článku (Richardson, 2002) skúmajú pravdepodobnostné rozdelenia, ktoré odvodené od $R_q(j)$, miery relevantnosti stránky j pre dopyt q:

$$P'_q(j) = \frac{R_q(j)}{\sum_{k \in W} R_q(k)} \quad P_q(i \to j) = \frac{R_q(j)}{\sum_{k \in F_i} R_q(k)}$$

W je množina všetkých stránok. F_i je množina stránok, na ktoré stránka i odkazuje. Pri výbere z množiny možných výstupných odkazov zo stránky si inteligentný používateľ

s väčšou pravdepodobnosťou vyberá stránky, o ktorých si myslí, že sú relevantnejšie pre daný dopyt (úmerne R_q). Podobne, ako v prípade PageRank algoritmu, pokiaľ všetky odkazy na stránke majú nulovú relevantnosť, alebo stránka neobsahuje žiadne odkazy, nasleduje automatický skok na ľubovoľnú stránku v sieti s pravdepodobnosťou danou $P'_q(j)$.

Pri viaczložkových dopytoch $Q = q_1, q_2, ...$ je model inteligentného používateľa nasledovný: používateľ si zvolí jeden dopyt q podľa nejakého pravdepodobnostného rozdelenia $P(q)$ a používa tento dopyt pre riadenie svojho správania pre veľký počet krokov (kým sa nedosiahne konvergencia). Následne si zvolí iný dopyt tiež podľa $P(q)$ pre veľký počet krokov a dokola. Výsledné pravdepodobnostné rozdelenie pre navštívené stránky je:

$$P_Q(j) = \sum_{q \in Q} P(q) P_q(j)$$

Funkcia relevantnosti $R_{q(j)}$ môže byť ľubovoľná. V najjednoduchšom prípade kde $R_q(j) = R$ je nezávislá od dopytu a dokumentu sa QD-PageRank redukuje na PageRank. Iný typ jednoduchej funkcie relevantnosti môže byť $R_q(j) = 1$ pre stránky, v ktorých sa nachádza výraz q a 0 pre všetky ostatné. Dajú sa použiť aj heuristické prístupy založené na meraní veľkosti textu, jeho polohy, alebo TFIDF (Salton, 1987) metrika.

5.3 Wikipédia a znalosti

Viacero výskumných skupín si všimlo potenciál Wikipédie ako zdroja znalostí pre rôzne aplikácie. V tejto kapitole budú niektoré tieto projekty rozobraté.

Wikipedia XML Corpus

V rámci projektu Wikipedia XML Corpus (Denoyer, 2006) bola pripravená zbierka XML dokumentov prevedených z textového obsahu Wikipédie a zbierka multimediálnych materiálov z obrázkov uvedených v článkoch anglickej Wikipédie. Hlavná textová zbierka používa UTF-8 znakové kódovanie a pokrýva osem jazykov: angličtinu, francúzštinu, nemčinu, holandčinu, čínštinu, arabčinu a japončinu. Zamýšľané použitie zbierky je pre rôzne úlohy z XML Information Retrieval a strojového učenia, ako napríklad katalogizačné úlohy, klastrovacie úlohy, mapovanie štruktúr.

Wikipedia[3]

Projekt Wikipedia[3] (System One, 2007) je snahou o prevod obsahu anglickej Wikipédie do RDF. Výsledná dátová množina kombinuje štrukturálne informácie ako odkazy a zaradenie článkov do kategórií so základnými metainformáciami o jednotlivých článkoch. Súčasná verzia vychádzajúca z anglickej Wikipédie ku dňu 26.3.2006 obsahuje približne 47 miliónov trojíc dostupných vo formátoch RDF/XML, Turtle a N-Triples pod GFDL licenciou na web stránke projektu. Dátová množina používa ontológiu http://www.systemone.at/2006/03/wikipedia, ktorá je modelovaná podľa WikiOnt (Harth, 2005), pričom kdekoľvek to bolo možné, boli použité elementy z Dublin Core (Beckett, 2002) a Simple Knowledge Organisation System (Miles, 2005).

Platypus Wiki

Platypus Wiki (Tazzoli, 2004) bol jeden z prvých sémantických wiki systémov. Umožňuje používateľovi pridávať sémantické informácie do osobitných vstupných polí,

ktoré sú oddelené od štandardného wiki textu. Používateľ vkladá sémantické informácie pomocou RDF v N3 notácií (Berners-Lee, 2005). Tento prístup je veľmi orientovaný na RDF a pravdepodobne dosť zložitý pre obyčajného používateľa Wikipédie, lebo požaduje dobré znalosti RDF a N3.

WikSAR

WikSAR (Semantic Authoring and Retrieval within a Wiki) (Aumüeller, 2005) používa na rozdiel od Platypus Wiki inú, lepšie integrovanú syntax. Používateľ tu môže pridávať sémantické tvrdenia pomocou riadkov v tvare `predikát:objekt` priamo v zdrojom texte. Napríklad pridaním `FigureBy:WilliamShakespeare` na stránke s názvom `PrinceHamlet` sa do systému pridá RDF trojica, kde objekt je `WilliamShakespeare`, predikát je `FigureBy` a subjekt je `PrinceHamlet`.

Semantic Wikipedia

Cieľom projektu (Volkel, 2006) je sprístupniť ručne editované informácie Wikipédie pre automatizované spracovanie pomocou W3C štandardov RDF (Manola, 2004), XSD (Fallside, 2001), RDFS (Brickley, 2004) a OWL (Smith, 2004). Autori chcú tento cieľ dosiahnuť pomocou rozšírenia programu MediaWiki a pri návrhu identifikovali týchto päť základných požiadaviek:

- *Použiteľnosť* – V prvom rade musí rozšírenie MediaWiki spĺňať požiadavky na použiteľnosť, keďže MediaWiki používa veľmi veľká komunita používateľov. Títo používatelia musia dokázať používať toto rozšírenie bez dodatočných technických znalostí alebo školenia.
- *Expresívnosť* – Je výhodné mať čo najväčšie množstvo informácií dostupných v strojovo spracovateľnom formáte, ale toto môže komplikovať použiteľnosť a výkonnosť. Taktiež pri veľkej expresivite je možné zapisovať logicky nekonzistentné informácie.
- *Flexibilita* – Wiki systémy sa dajú použiť pre veľmi rozmanité úlohy a používateľ dokáže meniť formu a obsah zozbieraných informácií takmer neohraničeným spôsobom.
- *Škálovateľnosť* – Rozsah Wikipédie a fakt, že neustále narastá, je výzvou pre súčasné sémantické technológie.
- *Výmena údajov a kompatibilita* – Pre automaticky spracovateľnú Wikipédiu je nutné navrhnúť konkrétne rozhrania a exportovacie funkcie. Taktiež je dôležitá kompatibilita so súčasnými nástrojmi.

Autori ďalej identifikovali štrukturálne črty, ktoré majú potenciál pre automatizované spracovanie. Sú to tieto:

- *Kategórie*, ktoré klasifikujú články podľa ich obsahu. Kategórie už existujú vo Wikipédii a sú použité hlavne pre podporu orientácie vo veľkom množstve článkov.
- *Typované odkazy*, ktoré klasifikujú odkazy medzi článkami podľa ich významu. V súčasnej verzii programu MediaWiki existujú iba beztypové odkazy podobne ako WWW. Odkaz z článku London do článku England sa zapíše v zdrojovom kóde článku England ako:

```
[[London|Londýn]]
```

V oboch prípadoch sme nedefinovali, o aký odkaz ide, v akej relácii sú články England a London. Ak by sme chceli definovať, že Londýn je hlavným mestom Veľkej Británie, pomocou typových odkazov zapíšeme:

```
[[capital::London]]
```

Táto syntax bola rozšírená aj pre definíciu viacerých typov pre jeden odkaz, jednoducho sú jednotlivé typy uvedené za sebou: `[[typ1, typ2, ..., typn::cieľ odkazu|voliteľný text odkazu]]`

- *Atribúty*, ktoré definujú jednoduché dátové hodnoty relevantné pre obsah článku. Napríklad, keby sme chceli definovať počet obyvateľov Londýna pomocou typových odkazov, museli by sme pridať odkaz na článok s názvom „7421328", toto jednoznačne nie je správny spôsob, ako definovať takýto druh údajov, vytvára veľké množstvo stránok s číselnými názvami, v ktorých navyše názov nezachytáva číselný význam stránky (napríklad lexikografické usporiadanie názvov je iné, ako číselné usporiadanie).

  Ako teda zapísať populáciu? Riešením je zápis populácie pomocou atribútu s číselnou hodnotou

```
[[population:=7421328]]
```

Použitie `:=` namiesto `::` umožňuje rozlíšiť tieto dva koncepty a súčasne poukazuje na ich príbuznosť. Taktiež je možné pridať voliteľný text, ktorý sa zobrazuje namiesto atribútu v článku.

```
[[population:=7421328|približne 7,5 milióna]]
```

Rozšírenie Semantic Wikipédia podporuje aj číselné veličiny s priradenou fyzikálnou jednotkou. Napríklad plocha Londýna sa dá zadefinovať ako:

```
[[area:=609 square miles]]
```

Množina podporovaných jednotiek je v súčasnosti dosť obmedzená, pretoby bolo zaujímavé preskúmať možnosť zaintegrovať program GNU Units (Mariano, 2004), ktorý obsahuje bohatú databázu fyzikálnych veličín.

Semantic Wikipedia používa dátový model RDF. Typové odkazy popisujú RDF vlastnosti[15] medzi RDF prostriedkami, ktoré označujú články Wikipédie. Atribúty zodpovedajú RDF vlastnostiam, ktoré sú medzi článkami a RDF dátovými literálmi. Súčasný model Wikipédia kategórií sa dá modelovať pomocou RDFS tried, lebo Wikipédia obmedzuje použitie kategórií iba na klasifikáciu článkov, neexistujú kategórie kategórií a tým pádom použitá časť RDFS je kompatibilná so sémantikou OWL DL.

[15] URI pre RDF vlastnosti sa generuje z typu odkazu, napríklad odkaz capital bude mať URI http://ontoworld.org/index.php/_Relation-3A#capital.

Autori Semantic Wikipedie navrhujú použiť pre prístup k údajom RDF dátového modelu dopytovacie jazyky ako napr. SPARQL (Seaborne, 2006), ktorý by sa dá použiť aj pre implementáciu vyhľadávacieho rozhrania.

5.4 Zhrnutie

Zaujímavým problémom je možnosť použiť znalosti z Wikipédie v algoritme QD-PageRank a preskúmať vlastnosti tohto spojenia – implementačné a používateľské.

Funkcia relevantnosti $R_q(j)$ v pravdepodobnostiach

$$P'_q(j) = \frac{R_q(j)}{\sum_{k \in W} R_q(k)} \quad P_q(i \to j) = \frac{R_q(j)}{\sum_{k \in F_i} R_q(k)}$$

závisí od stránky j a dopytu q. Zaujímavé funkcie $R_q(j)$ by mohli byť:

Priame odkazy z Wikipédie

- Funkcia $R_q(j)$ definovaná ako počet príspevkov z Wikipédie, ktoré obsahujú dopyt q a obsahujú odkaz na stránku j.

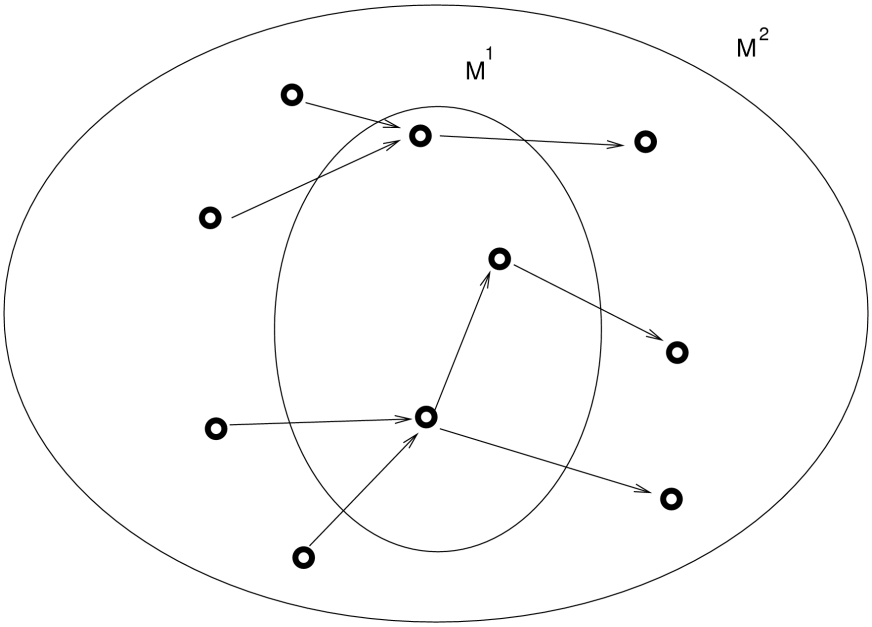
  Stránka, ktorá je takto vybraná, má veľkú relevantnosť, lebo bola vybraná a pridaná manuálne editormi do príspevku. V tomto prípade sa Wikipédia používa ako ručne editovaná zbierka zaujímavých odkazov podobná Yahoo! adresáru, Open Directory. Problémom by mohlo byť, že články Wikipédie obsahujú odkazy iba na veľmi malú časť celého webu a preto by takto definovaná funkcia $R_q(j)$ mala pre veľké množstvo stránok hodnotu 0. Riešením by bolo oslabiť podmienku, aby j bolo zhodné s URL adresou uvedenou v príspevku a do úvahy by sa brala aj podobnosť URL adries.

Susedné príspevky

- Pre daný dopyt Q sa pomocou PageRank alebo QD-PageRank nájde n najrelevantejších príspevkov vo Wikipédii. Označme túto množinu M^1. Množina príspevkov, ktoré sú odkazované z množiny M^1 alebo odkazujú do množiny M^1 označme ako M^2. Vo väčšine prípadov je M^2 väčšia ako M^1 (obrázok 1-8).

  Funkcia $R_q(j)$ je potom definovaná ako súčet zhody stránky j so stránkami množiny M^1 a s menšou váhou súčet zhody stránky j so stránkami množiny M^2. Pre porovnávanie zhody stránky s príspevkom z Wikipédie sa dajú použiť rôzne algoritmy, od Bayesovských klasifikátorov až po neurónové siete, pre vyhľadávanie je ale dôležitá nízka pamäťová a výpočtová náročnosť.

  Takto definovaná funkcia napodobňuje správanie používateľa, ktorý si najprv nájde na Wikipédii relevantné príspevky, tie si prečíta a podľa získaných informácií vyhľadáva ďalej stránky na webe. Problémom tohto prístupu je náchylnosť tejto funkcie R vysoko hodnotiť stránky, ktoré skopírovali informácie priamo z príspevkov Wikipédie.



Obrázok 5-8. Vzťah množín M^1 a M^2.

Údaje pre spracovanie článkov Wikipédie sa dajú získať z projektu Wikipedia XML Corpus a Wikipedia[3].

Dôležité bude aj preskúmať časovú a pamäťovú náročnosť uvedených funkcií. V článku (Richardson, 2002) je odhadovaná časová a pamäťová náročnosť pre QD-PageRank na 100-200 násobok náročnosti jednoduchého PageRank, ale autori predpokladali v analýze funkciu $R_q(j)$, ktorá vždy vracia 0, keď sa d q nenachádza na stránke j.

Použitá literatúra

AUMÜLLER, D. – AUER, S. (2005). Towards a semantic wiki experience - desktop integration and interactivity in WikSAR. In *Proc. of 1st Workshop on The Semantic Desktop - Next Generation Personal Information Management and Collaboration Infrastructure, Galway, Ireland,*Nov. 6th.

BECKETT, D. – MILLER, E. – BRICKLEY, D. (2002). Expressing Simple Dublin Core in RDF/XML. Technical report, Dublin Core Metadata Initiative.

BERNERS-LEE, T. (2005). *Primer: Getting into RDF & Semantic Web using N3*. http://www.w3.org/2000/10/swap/Primer.html.

BERNERS-LEE, T. – FIELDING, R. – MASINTER, L. (2005). Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard).

BRAY, T. – PAOLI, J. – SPERBERG-MCQUEEN, C. M. (2000). Extensible markup language (XML) 1.0 (second edition). W3C Recommendation REC-xml-20001006, World Wide Web Consortium (W3C). Available at http://www.w3.org/XML/.

BRICKLEY, D. – GUHA, R. V. (2004). RDF Vocabulary Description Language 1.0: RDF Schema. W3C recommendation, World Wide Web Consortium.

BRIN, S. – PAGE, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107-117.

CHEN, H. (1995). Machine learning for information retrieval: Neural networks. *Journal of the American Society for Information Science*, 46(3):194-216.

CHEN, H. – CHAU, M. – ZENG, D. (2002). CI Spider: A Tool for Competitive Intelligence on the Web. *Decision Support Systems*, 34(1):1 - 17.

CHEN, H. – CHUNG, Y.-M. – RAMSEY, M. – YANG, C. C. (1998). A Smart Itsy Bitsy Spider for the Web. *Journal of the American Society of Information Science*, 49(7):604-618.

CHEN, H. – FAN, H. – CHAU, M. – ZENG, D. (2001). MetaSpider: Meta-searching and Categorization on the Web. *Journal of the American Society for Information Science and Technology*, 52(13):1134 - 1147.

CLARK, J. – MAKOTO, M. (2001). *RELAX NG Specification*. OASIS, 1 edition.

DE BRA, P. M. E. – POST, R. D. (1994). Information retrieval in the World-Wide Web: Making client-based searching feasible. *Computer Networks and ISDN Systems*, 27(2):183-192.

DENNING, P. – HORNING, J. – PARNAS, D. – WEINSTEIN, L. (2005). Wikipedia risks. *Commun. ACM*, 48(12):152-152.

DENOYER, L. – GALLINARI, P. (2006). The Wikipedia XML corpus. *SIGIR Forum*, 40(1):64-69.

DEUTSCH, P. (1996). DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational).

FALLSIDE, D. C. (2001). XML Schema Part 0: Primer. W3C recommendation, World Wide Web Consortium.

FREE SOFTWARE FOUNDATION (2002). The GNU Free Documentation License. http://www.fsf.org/copyleft/fdl.html.

GULLI, A. – SIGNORINI, A. (2005). The indexable web is more than 11.5 billion pages. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 902-903, New York, NY, USA. ACM Press.

HARTH, A. – GASSERT, H. – O'MURCHU, I. – BRESLIN, J. G. – DECKER, S. (2005). WikiOnt: An Ontology for Describing and Exchanging Wikipedia Articles. In *Proceedings of Wikimania 2005 - The First International Wikimedia Conference*.

HUANG, L. (2000). A survey on web information retrieval technologies. Technical report, ECSL.

ISO (1986). *ISO 8879:1986: Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland.

IVORY, M. Y. – SINHA, R. R. – HEARST, M. A. (2001). Empirically validated web page design metrics. In *CHI*, pages 53-60.

JELÍNEK, I. (1998). Od textu k dobrému hypertextu. *Softwarové novin*, 9(6):102-107. ISBN: 1210-8472.

KLARLUND, N. – MØLLER, A. – SCHWARTZBACH, M. I. (2000). Document structure description 1.0. Notes Series NS-00-7, BRICS, Department of Computer Science, University of Aarhus. 40 pp.

KLEINBERG, J. M. (1998). Authoritative sources in a hyperlinked environment. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 668-677, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics. IBM Technical Report RJ 10076.

KUZNETSOV, S. (2006). Motivations of contributors to wikipedia. *SIGCAS Comput. Soc.*, 36(2):1.

LEUF, B. – CUNNINGHAM, W. (2001). *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley, Reading, Massachusetts. ISBN-13: 978-0201714999.

LI, Y. (1998). Toward a qualitative search engine. *IEEE Internet Computing*, 2(4):24-29.

MANOLA, F. – MILLER, E. (2004). RDF Primer. W3C recommendation, World Wide Web Consortium.

MARIANO, A. (2004). GNU Units. http://www.gnu.org/software/units/units.html.

MILES, A. – BRICKLEY, D. (2005). SKOS Core Vocabulary Specification. W3C working draft, World Wide Web Consortium.

MILLER, R. – BHARAT, K. (1998). SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers. In *Proceedings of the Seventh International WWW Conference*, pages 161-172.

PAGE, L. – BRIN, S. – MOTWANI, R. – WINOGRAD, T. (1998). The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project.

PINKERTON, B. (2000). *WebCrawler: Finding what people want*. Phd thesis, University of Washington.

RICHARDSON, M. – DOMINGOS, P. (2002). The Intelligent Surfer: Probabilistic Combination of Link and Content Information in PageRank. In *Advances in Neural Information Processing Systems 14*. MIT Press.

RIEHLE, D. (2006). How and why Wikipedia works: an interview with Angela Beesley, Elisabeth Bauer, and Kizu Naoko. In *WikiSym '06: Proceedings of the 2006 international symposium on Wikis*, pages 3-8, New York, NY, USA. ACM Press.

ROSENZWEIG, R. (2006). Can History be Open Source? Wikipedia and the Future of the Past. *The Journal of American History*, 93(1):117–146. http://chnm.gmu.edu/resources/essays/d/42.

SALTON, G. – BUCKLEY, C. (1987). Term weighting approaches in automatic text retrieval. Technical report, Cornell University, Ithaca, NY, USA.

SEABORNE, A. – PRUD'HOMMEAUX, E. (2006). SPARQL Query Language for RDF. Technical Report http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/, W3C.

SMITH, M. K. – WELTY, C. MCGUINNESS, D. L. (2004). OWL Web Ontology Language Guide. W3C recommendation, World Wide Web Consortium.

SYSTEM ONE (2007). Wikipedia[3]. [Online; accessed 12-February-2007] http://labs.systemone.at/wikipedia3.

TAZZOLI, R. – CASTAGNA, P. CAMPANINI, S. E. (2004). Towards a Semantic Wiki Wiki Web. In *ISWC*.

TOLLE, K. M. – CHEN, H. (2000). Comparing noun phrasing techniques for use with medical digital library tools. *Journal of the American Society for Information Science*, 51(4):352 - 370.

UNICODE CONSORTIUM (2006). *The Unicode Standard, Version 5.0*. Addison-Wesley, Reading, Massachusetts. ISBN-10: 0321480910.

UT AUSTIN TEAMWEB (2007). University of Texas - Austin Web Publishing Guidelines. [Online; accessed 20-February-2007] http://www.utexas.edu/web/guidelines/.

VIÉGAS, F. B. – WATTENBERG, M. DAVE, K. (2004). Studying cooperation and conflict between authors with history flow visualizations. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 575-582, New York, NY, USA. ACM Press.

VÖLKEL, M. – KRÖTZSCH, M. – VRANDECIC, D. – HALLER, H. STUDER, R. (2006). Semantic Wikipedia. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 585-594, New York, NY, USA. ACM Press.

WALES, J. (2005). Wikipedia in the free culture revolution. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* pages 5-5, New York, NY, USA. ACM Press.

WATERHOUSE, S. – DOOLIN, D. – KAN, G. FAYBISHENKO, A. (2002). Distributed search in P2P networks. *IEEE Internet Computing*, 6:68-72.

WIKIPEDIA (2007). Wikipedia – Wikipedia, The Free Encyclopedia. [Online; accessed 10-February-2007]. http://en.wikipedia.org/w/index.php?title=Wikipedia&oldid=106967020

GAMMA, E. ET AL. (1995). *Design Patterns*. 1st edition. Massachusetts: Addison-Wesley, pp. 395.

DEEPAK, A. ET AL. (2001). *Core J2EE Patterns: Best Practices and Design Strategies*. 1st edition. Prentice Hall/Sun Microsystems Press, pp. 496.

DURFEE, E.H. – LESSER, V.R. (1991) Partial Global Planning: A Coordination Framework for Distributed Hypothesis Formation. In: *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21, No. 5, 1167-1183.

HAKALA, M. – HAUTAMÄKI, J. – KOSKIMIES, K. ET AL. (2001). Generating application development environments for Java frameworks. In: *Proc. of the 3rd Int. Conf. on Generative and Component-Based Software Engineering GCSE'01 Vol*. 2186, Springer Verlag, pp. 163–176.

ZHONG, N. – LIU, J. – YAO, Y. (Eds.) (2003) *Web Intelligence*. 1st edition. Springer, pp. 440.

6 VYHĽADÁVANIE INFORMÁCIÍ

Oblasť vyhľadávania informácií (*information retrieval, IR*) sa zaoberá reprezentáciou, uchovávaním, organizáciou a dostupnosťou informácií. Na rozdiel od získavania dát *(data retrieval*) pri získavaní informácií nie je potrebné iba získať presne definované informácie, napríklad podľa obsahu kľúčových slov, ale cieľom je získať čo najrelevantnejšie informácie o nejakom subjekte.

Oblasť vyhľadávania informácií sa nezačala rozvíjať len s nástupom počítačov ale je stará ako ľudstvo samo aspoň od nástupu písma ako symbolickej formy zaznamenávania informácií. Už najmenej 4000 rokov sa ľudia snažia organizovať informácie tak aby v nich bolo možné vyhľadávať. Typickým príkladom je obsah knihy alebo rôzne indexy v knižniciach, kde sa oblasť vyhľadávania informácii začala najskôr rozvíjať.

S nástupom internetu prišli nové výzvy pre túto oblasť:

- Informácie sú dostupné za oveľa nižšiu cenu ako predtým
- Rozvoj internetu napomohol tomu, že informačné zdroje sú distribuované a dostupné veľmi rýchlo a za pár sekúnd
- Takisto sloboda zverejniť akúkoľvek informáciu kýmkoľvek je dostupná prvý krát v histórii

Tieto výzvy sa nedajú naplniť riešeniami, ktoré sa používali v knižniciach ako hierarchické zoznamy, indexy a podobne.

S rozvojom internetu je čoraz ťažšie nájsť tu správnu informáciu na webe, ktorý je dynamickým neustále sa zväčšujúcim a meniacim priestorom.

Najdôležitejším médiom na internete je stále text, ktorým sa zaoberáme aj v tejto kapitole, avšak v súčasnosti je čoraz dôležitejšie vyhľadávanie informácií z iných médií na webe ako sú obrázky, zvuk alebo hudba, video a v budúcnosti určite pribudne aj IP televízia a ďalšie.

6.1 Základné pojmy a architektúra vyhľadávacieho systému

Architektúru vyhľadávacieho systému môžeme rozdeliť na dva základné procesy ktoré sa vykonávajú nad informačným priestorom a jeho základnými jednotkami ktorými sú dokumenty. Prvým je proces získavania informácií ktorý sa skladá z týchto častí:

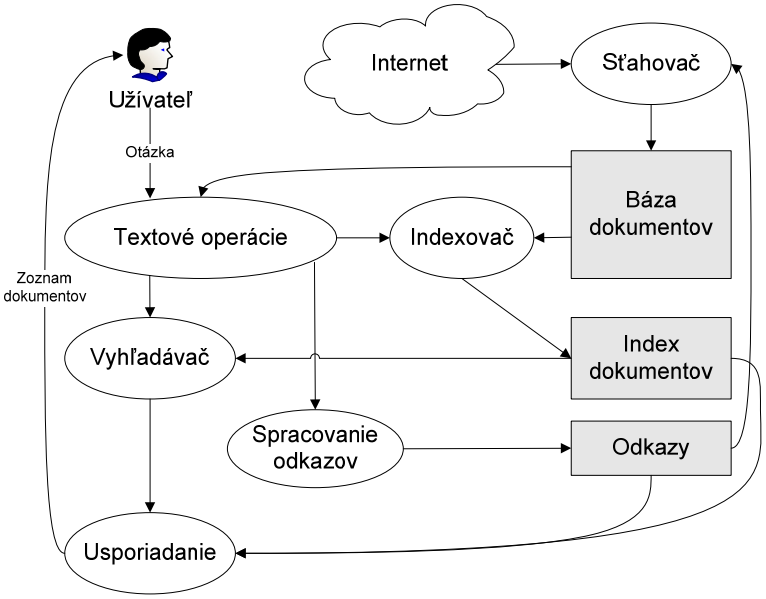
- stiahnutie dokumentov

- textové operácie
- indexovanie
- spracovanie odkazov

Druhým procesom je proces vyhľadávania skladajúci sa z

- formulácia dopytu a operácie na dopyte
- spracovanie dopytu
- vrátenie výsledku na používateľské rozhranie
- spätná väzba od používateľa

Na obrázku 6-1 je zobrazená architektúra vyhľadávacieho systému, ktorá zahŕňa tieto procesy. V ďalšom texte v krátkosti opisujeme jednotlivé časti týchto procesov, pričom niektorým väčšiu pozornosť v rámci kapitoly.



Obrázok 6-1. Architektúra vyhľadávacieho systému.

Dokumenty sú základnou jednotkou pre vyhľadávanie. Dokumenty sa sústredia v báze dokumentov, ktorá je dopredu pripravená, alebo v prípade systémov ktoré pracujú s prostredím internetu sa báza napĺňa pomocou web sťahovačov (*Web Crawler* alebo *Web Spider*), ktoré prechádzajú odkazy nájdené v dokumente a sťahujú nové dokumenty do bázy dokumentov na základe nadefinovaných pravidiel (napr. obmedzenia na web sídla, hĺbku alebo relevanciu dokumentov).

Textové operácie slúžia na zistenie „*Index terms*“ termov z dokumentu ale aj na prvotné spracovanie dopytu, kde je potrebné aplikovať tie isté operácie ako na dokument. Tu sa vyhadzujú nepodstatné slová „stop slová“ (*stop words*) alebo hľadajú základné tvary slov. Pod textové operácie patrí aj prvotná tokenizácia textu (časť 6.5) a môžeme tu zaradiť aj konvertovanie rôznych formátov na text ako aj identifikáciu objektov z dokumentu (napr. odkazy URL).

Dokument spracovaný textovými operáciami je následne indexovaný, čím sa vytvorí index dokumentov na základe zvoleného modelu, ktorý slúži na rýchle vyhľadanie relevantných dokumentov.

Vo vyhľadávacom systéme zameranom na prostredie internetu je dôležité spracovať odkazy extrahované z dokumentov. Toto môže slúžiť rôznym účelom ako napríklad zisťovanie informácií z grafovej hypertextovej štruktúry, zisťovanie dodatočných informácií o dokumentoch na základe textu odkazov na iných stránkach alebo aj navigácii v dokumentoch. Primárnym cieľom vo vyhľadávacích systémoch je však podpora zoraďovania výsledkov vyhľadávania napríklad pomocou PageRank[16] algoritmu.

Niektoré vyhľadávacie systémy umožňujú spätnú väzbu na výsledky vyhľadávania, čím sa môže doplniť alebo spresniť dopyt používateľa napríklad označením správnych a nesprávnych vstupov alebo hľadaním podobných dokumentov. Takisto je možné podporovať proces prehliadania výsledkov (browsing) napríklad pomocou fazetového prehliadača.

6.2 Modely a indexovanie

V IR sú 3 základné modely:

- *Booleovský* model je reprezentovaný množinou termov a teda je založený na *teórii množín*
- *Vektorový model*. Vo vektorovom modeli sú dokumenty a dopyty reprezentované ako vektor v n-dimenzionálnom priestore, a teda ho môžeme nazvať aj *algebrický* model
- *Pravdepodobnostný model* je založený na teórii pravdepodobností. Napríklad využívajúci Bayesovú teóriu pre podobnosť dokumentov, čo sa využíva pri SPAM filtroch.

V ďalšom sa budeme zaoberať hlavne booleovským a vektorovým modelom ktoré sa najviac používajú v tejto oblasti hlavne pri vyhľadávaní na webe.

Keď definujeme k_i ako term v dokumente D_j kde má váhu $w_{ji} \geq 0$.

Booleovský model

Potom booleovský model, v ktorom sa namiesto dôležitosti slova v dokumente zisťuje iba jeho prítomnosť resp. neprítomnosť, váham w_{ji} prideľuje hodnoty 0 alebo 1. Výhodou modelu je jeho jednoduchosť a jasná formálna reprezentácia dokumentu. Nevýhodou je že dokument je buď úplne relevantný alebo úplne nerelevantný čo môže viesť k veľmi malým alebo veľmi veľkým kolekciám. Pri veľkých kolekciách potom nie je možné zoradiť dokumenty podľa frekvencie výskytu. Problém je zreteľný najmä pri rozsiahlych dokumentoch, kde počet slov v reprezentácii výrazne rastie. Mnohé zo slov sa pritom vôbec netýkajú obsahu dokumentu, sú však („vďaka“ modelu) brané ako rovnako významné slová (Koštiaľ 2002, Furdík, 2003).

[16] http://www.google.com/technology/

Vektorový model

Vektorový model (Luhn 1953). Každý dokument D_j je reprezentovaný vektorom $d_j = \left(w_{j1}, w_{j2}, ..., w_{jn}\right)$, kde w_{ji} je váha, resp. dôležitosť termu k_i (v tomto prípade frekvencia výskytu) v dokumente d_j a n je veľkosť množiny termov popisujúcej dokument (Salton-Buckley 1988). Váhu môžeme vypočítavať rôznymi spôsobmi. Najznámejší je výpočet váhy na základe počtu výskytov termov. Z analýz je známe (Salton, 1975) že váha termu založená na počte výskytov je vhodná najmä v systémoch kde chceme získať čo najviac relevantných výsledkov – vysoké pokrytie (recall), kým termy s nízkym počtom výskytov nám zasa môžu zvyšovať presnosť (precision).

Vzájomná podobnosť dokumentov alebo dokumentu D_j a dopytu $\vec{q} = \left(q_1, q_2, ..., q_n\right)$ sa zistí metódou kosínusovej korelácie vektorov $\vec{d}_j$ a $\vec{q}$:

$$sim\left(d_j, q\right) = \frac{\vec{d}_j \bullet \vec{q}}{\left|\vec{d}_j\right|\left|\vec{q}\right|} = \frac{\sum_{i=1}^{n} w_{ij} w_{iq}}{\sqrt{\sum_{i=1}^{n} w_{ij}^2} \sqrt{\sum_{i=1}^{n} w_{iq}^2}}.$$

Výhodou modelu je že sa dá jednak zrýchliť vyhľadávanie a výkonnosť IR systému ako aj zoraďovanie relevantných dokumentov podľa miery podobnosti. V súčasnosti je vektorový model najpopulárnejším modelom pre IR systémy a modely z neho odvodené ako napríklad latentný sémantický model.

Indexovanie

Cieľom indexovania je optimalizovať rýchlosť a výkonnosť vyhľadávania dokumentov pre zadaný dopyt. Bez vytvorenia indexov by musel systém prechádzať každým dokumentom a hľadať kľúčové slová alebo aplikovať vzory z dopytu na každý dokument čo by bolo náročné na čas aj výpočtovú silu.

Pri vytváraní indexu je nutné zohľadniť nasledovné faktory:

- Rýchlosť vyhľadávania: ako rýchlo môžu byť termy nájdené v indexoch ale tiež s ohľadom na rýchlosť aktualizácie indexu pri pridaní, zmene alebo odobratí dokumentov
- Veľkosť indexov: koľko miesta na disku zaberú, ako sa zväčšujú s počtom dokumentov
- A tiež faktory ako distribúcia indexovacieho mechanizmu na viac výpočtových zdrojov a následné spájanie indexov, odolnosť voči chybám v dátach a ďalšie.

Dátová štruktúra indexov súvisí s vyššie uvedenými modelmi. V oblasti systémov pracujúcich s internetom je indexovanie založené hlavne na vektorovom a niekedy aj na booleovskom modeli.

Hlavné typy indexov sú nasledovné:

- Invertované indexy: ukladajú výskyt každého slova alebo termu v dokumente vo forme množín alebo hash tabuliek. Takýto typ indexu sa používa pri booleovskom modeli.

- Matica termov dokumentu: používa sa najmä v latentnom sémantickom modeli ktorý je rozšírením vektorového modelu, pričom sa ukladá výskyt slov v riedkej dvojrozmernej matici.
- Stromy: sú využívané na ukladanie indexov pomocou asociatívnych polí kde kľúčmi sú jednotlivé reťazce alebo termy. Takáto štruktúra je rovnako rýchla ako použitie hash tabuliek ale vyžaduje väčšiu pamäť pre indexy.
- Sufixové stromy: majú lineárnu časovú náročnosť a ukladajú slová alebo termy pomocou prípon slov. Problémom je najmä veľkosť výsledných indexov.
- Taktiež je možné použiť techniky ako *n*-gram, citačné alebo hyperlinkové indexy a ďalšie.

6.3 Vyhľadávanie a prehliadanie

Vyhľadávanie informácií je v IR systémoch inicializované používateľom, a to tak že definuje dopyt, ktorý nejakým spôsobom reprezentuje to čo chce používateľ nájsť. Najčastejšie sú takýmto dopytom kľúčové slová. Dopyt však môže obsahovať ďalšie spresňujúce značky ako logické operátory, obmedzenia na doménu dokumentov a podobne. V súčasnosti je tiež snaha definovať dopyt v prirodzenom jazyku. Spôsob dopytu je definovaný takzvaným dopytovacím jazykom (*query language*), kde poznáme nasledovné základné typy takýchto jazykov (Baeza-Yates, 1999):

- Kľúčové slová: patria medzi najjednoduchší a najbežnejší typ dopytu. V prípade kľúčových slov je možné hľadať dokumenty ktoré obsahujú dané slovo alebo slová. Pričom existujú ďalšie rozšírenia takýchto dotazov a to tak že sa hľadá v dokumente presná fráza, teda postupnosť slov alebo časť textu kde tieto slová spolu súvisia v kontexte, teda napríklad pomocou definície maximálnej vzdialenosti týchto slov. Vzdialenosť sa môže definovať pomocou počtu slov alebo napríklad výskyt slov v rámci rovnakého odseku.
- Logické dopyty: (*boolean queries*) sú založené na využívaní logických operátorov medzi atómami dopytu. Atómy sú väčšinou kľúčové slová. Potom vrátené dokumenty spĺňajú logický dopyt. Tento typ dopytu sa dosť často využíva aj v súčasných IR systémoch. Medzi základné operácie patria OR, AND a BUT. OR a AND sú intuitívne operácie „alebo“ a „a zároveň“, BUT môže byť použité na získanie dokumentov (a BUT b) ktoré spĺňajú podmienku „a“ a nespĺňajú podmienku „b“.
- Prirodzený jazyk: V súčasnosti je snaha definovať otázky v prirodzenom jazyku a tie previesť na nejakú formalizovanú formu ktorej rozumie počítač. Podrobnejšie informácie v tejto oblasti sú napríklad v (Baeza-Yates, 1999).
- Dopyt založený na vzoroch (pattern matching): Tieto techniky sú vhodné najmä v špecializovaných vyhľadávacích systémoch. Pri vyhľadávaní na internete by vznikal problém s výkonnosťou systému. Dopyty založené na vzoroch môžu byť nasledujúcich typov: hľadanie časti slova, reťazca, hľadanie na základe predpony a prípony slov alebo hľadanie pomocou regulárnych výrazov.
- Špeciálne typy dopytov: Tu môžeme zaradiť dopyty špecifické pre jednotlivé vyhľadávacie systémy, ktoré obmedzujú kolekciu dokumentov prípadne využívajú špeciálne vlastnosti vyhľadávačov alebo informačného priestoru.

Príkladom je obmedzenie dopytu vo vyhľadávači Google[17] na typ súboru, webové sídlo alebo jazyk.

Typ dopytovacieho jazyka súvisí so zvoleným modelom (kapitola 6.2) vyhľadávacieho systému. Dopyt je spracovávaný pomocou operácií na dopyte ktoré môžu zahŕňať rôzne operácie na základe spätnej väzby od používateľa, ktoré zmenia pôvodný dopyt (Baeza-Yates, 1999). Dopyt musí byť tiež spracovaný pomocou textových operácií aby kľúčové slová z dopytu boli zmenené na rovnaké termy ako obsahujú indexy dokumentov. Takisto je nutné zanedbať stop slová, tokenizovať dopyt a podobne.

Výsledkom vyhľadávania je zoznam dokumentov. Tento zoznam dokumentov je väčšinou usporiadaný (pozri ďalšiu kapitolu 6.4). Dôležitou súčasťou vyhľadávania informácií je aj prehliadanie (browsing). V rámci internetu je podpora prehliadania dôležitou súčasťou vzhľadom na hypertextový priestor. Existujú teda prístupy ktoré podporujú navigáciu používateľa v informačnom priestore po prvotnom vyhľadaní informácií a to aj napríklad tak že počas prehliadania modifikujú dopyt vyhľadávania čoho výsledkom je spresnenie dopytu. Príkladom je fazetová navigácia ktorá obmedzuje informačný priestor definovaním podmienok pri prehliadaní (Tvarožek, 2007).

6.4 Usporiadanie

Usporiadanie dokumentov podľa relevantnosti je dôležitou funkciou vo všetkých vyhľadávacích systémoch. Väčšina vyhľadávačov sa snaží vrátiť čo najviac relevantných dokumentov, teda zvyšuje pokrytie (*recall*) na úkor presnosti (*precesion*), čo zvyšuje potrebu usporiadať dokumenty podľa relevantnosti. Tu sa využívajú rôzne algoritmy väčšinou založené na mierach podobnosti dokumentov k dopytu. Tieto podobnosti sa odvíjajú od počtu výskytov kľúčových slov (termov), ich výskytu v nadpisoch dokumentov a podobne. Ďalšou metrikou na usporiadanie sú aj algoritmy založené na vyhodnocovaní odkazov medzi dokumentmi ako napríklad PageRank algoritmus. Dokumenty je možné zoraďovať aj na základe daľších sémantických prístupov ako napríklad Top-K (Gurský, 2005).

Ak máme viacero spôsobov na tvorbu utriedených zoznamov, tieto je možné navzájom kombinovať normalizáciou kritérií ich váhovaním a následným násobením jednotlivých kritérií.

Usporiadanie vo vyhľadávači Google

Google používa na zoradenie webových stránok podľa „dôležitosti“ algoritmus PageRank, ktorý berie do úvahy faktory ako názov dokumentu, kľúčové slová a ďalšie faktory. Základné kroky, ktoré Google vykonáva pri hodnotení stránok vo svojom vyhľadávacom algoritme, sú nasledovné:

1. Nájde všetky webové stránky obsahujúce dané kľúčové slovo (slová).
2. Vyhodnotí stránky na základe faktorov nachádzajúcich sa na stránke, ako napríklad, či sa dané slovo nachádza v názve stránky, akým štýlom je písané, ako často sa slovo (slová) nachádzajú v hlavnej časti dokumentu, a pod.

[17] http://www.google.sk/advanced_search?hl=en

3. Vyhodnotí text vo vnútri odkazu (<a href>), samotný odkaz a vyhodnotí váhu textu. (napríklad <a href www.nazovdomeny.com>Nazovdomeny má väčšiu váhu ako <a href www.nazovdomeny.com>Návrat)
4. Pridá výsledok algoritmu PageRank, ktorého hodnotu budeme nazývať „page rank".

V skutočnosti je tento výpočet o čosi komplikovanejší a budeme sa mu venovať nižšie. Je dôležité si uvedomiť, že krok 4 v predchádzajúcom postupe (pridanie výsledku algoritmu PageRank) nie je pripočítanie, ale násobenie s výsledkom z predchádzajúcich troch krokov. To znamená, že aj keď sa na stránke nachádza kľúčové slovo (slová) viackrát, ale stránka má page rank 0, vo vrátenom zozname sa táto stránka zobrazí medzi poslednými.

Podobnosť na základe metrík a korelácií

Predpokladáme, že máme dva dokumenty D_i a D_j reprezentované vektormi $\vec{d}_i = (w_1, w_2, ..., w_n)$ a $\vec{d}_j = (v_1, v_2, ..., v_n)$, potom je možné funkciu vzájomnej podobnosti dokumentov vypočítať napríklad pomocou týchto metrík (Kostelník, 2000):

- Euklidovská vzdialenosť: $sim(\vec{d}_i, \vec{d}_j) = \sum_{k=1}^{n} \sqrt{(w_k - v_k)^2}$
- L-metrika (Manhattan): $sim(\vec{d}_i, \vec{d}_j) = \sum_{k=1}^{n} |w_k - v_k|$
- Snp metrika: $sim(\vec{d}_i, \vec{d}_j) = \max_k |w_k - v_k|$
- Sokalova metrika: $sim(\vec{d}_i, \vec{d}_j) = \frac{\sum_{k=1}^{n} \sqrt{(w_k - v_k)^2}}{n}$

Najčastejšie používaný koeficient korelácie je tzv. kosínusová korelácia spomenutá v časti 6.2. Existuje viacero modelov kosínusovej korelácie: klasický, latentný sémantický a pravdepodobnostný.

Algoritmus PageRank

PageRank je metóda spoločnosti Google[18] na meranie „dôležitosti" webových stránok. Hodnotu ktorú vráti algoritmus PageRank budeme nazývať „page rank". Teória za touto metódou je, že ak stránka A ukazuje na stránku B, tak stránka A hovorí, že stránka B je „asi" dôležitá. Ak na stránku ukazujú dôležité stránky, tak aj odkazy tejto stránky na iné stránky sa stávajú dôležitými.

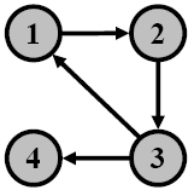
Page rank aktuálnej webovej stránky je možné zobraziť ako toolbar[19] vo webovom prehliadači. Hodnotenie stránok v tomto toolbare je len od 0 do 10. Keďže page rank môže nadobúdať ľubovoľné kladné hodnoty, nie je celkom zrejmé, akú metódu Google na určenie tejto hodnoty používa. Odhady ukazujú (Wills, 2006), že hodnoty, ktoré

[18] www.google.com

[19] www.toolbar.google.com

zobrazuje Googe toolbar sú približne logaritmované hodnoty PageRank algoritmu logaritmom pri základe 10.

Modelovanie aktivity náhodného používateľa webu sa dá reprezentovať ako orientovaný graf prepojení vychádzajúcich a vchádzajúcich na webové stránky. Obrázok 6-2 modeluje zjednodušenú situáciu webu so 4 vzájomne prepojenými webovými stránkami.



Obrázok 6-2. Prepojenia medzi 4 webovými stránkami.

Definícia algoritmu PageRank ako ho prvýkrát definovali Page a Brine v (Page, 1998a) je nasledovná.

Definícia 6-1: Page rank stránky A je $PR(A) = (1-d) + d\sum_{i=1}^{n} \frac{PR(T_i)}{C(T_i)}$ *kde,* $0 \le d < 1$ *je tzv. „damping factor" najčastejšie nastavený na 0,85 viď (Page, 1998b),* T_i *je stránka ktorá ukazuje na stránku A,* $PR(T_i)$ *je PageRank tejto stránky a* $C(T_i)$ *je počet prepojení odchádzajúcich zo stránky* T_i.

Tento výpočet opakujeme (iterujeme) dovtedy, kým hodnota $PR(A)$ nezačne konvergovať k limitnej hodnote. Je potrebné si uvedomiť, že page rank webových stránok vlastne tvoria rozdelenie pravdepodobnosti a teda suma všetkých page rank hodnôt musí byť 1. Na to aby sme dosiahli rozdelenie pravdepodobnosti musíme ešte získané hodnoty vydeliť (normalizovať) sumou všetkých page rank hodnôt. Inicializačné hodnoty $PR(T_i)$ je možné nastaviť na ľubovoľné hodnoty z dôvodu konvergencie viď (Golub, 1996) alebo (Page, 1998ab) sa vždy dopracujeme k tomu istému riešeniu (mení sa len počet iterácií).

Príklad 6-1: Vypočítajme page rank jednotlivých stánok z obrázku 6-2 ak damping factor $d = 0,85$ *a prvotné page ranky stránok sú nasledovné* $PR_0(1) = PR_0(2) = PR_0(3) = PR_0(4) = 1$.

Riešenie:

$$PR_1(1) = 0,15 + 0,85\frac{1}{2} = 0,575 \quad PR_1(2) = 0,15 + 0,85\frac{1}{1} = 1$$

$$PR_1(3) = 0,15 + 0,85\frac{1}{1} = 1 \qquad PR_1(4) = 0,15 + 0,85\frac{1}{2} = 0,575$$

a následne druhá iterácia

$$PR_2(1) = 0{,}15 + 0{,}85\frac{1}{2} = 0{,}575 \quad PR_2(2) = 0{,}15 + 0{,}85\frac{0{,}575}{1} = 0{,}63875$$
$$PR_2(3) = 0{,}15 + 0{,}85\frac{1}{1} = 1 \qquad PR_2(4) = 0{,}15 + 0{,}85\frac{1}{2} = 0{,}575$$

Ak by sme takýmto spôsobom pokračovali ďalej po 88 iteráciách by sme zistili, že page rank hodnoty jednotlivých webových stránok skonvergovali k nasledujúcemu riešeniu zaokrúhlenému na tri desatinné miesta

$$PR(1) = 0{,}387 \quad PR(2) = 0{,}479$$
$$PR(3) = 0{,}557 \quad PR(4) = 0{,}387$$

Keďže page rank je vlastne pravdepodobnosť s akou sa používateľ rozhodne ísť na tú ktorú webovú stránku vydelíme každú hodnotu page rank sumou všetkých hodnôt (normalizujeme/spriemerujeme) a dostaneme nasledovné pravdepodobnostné hodnoty page rank

$$PR(1) = 0{,}21 \quad PR(2) = 0{,}26$$
$$PR(3) = 0{,}31 \quad PR(4) = 0{,}21$$

Problém výpočtu page rank pomocou sumy cez všetky prepojenia ukazujúce na stránku *A* sa dá riešiť jednoduchšie pomocou maticového počtu čomu sa budeme venovať v nasledujúcej časti.

Vzájomné prepojenie webových stránok na obrázku 6-2 sa dá reprezentovať tzv. Hyperlinkovou maticou, o ktorej hovorí nasledujúca definícia.

Definícia 6-2: Hyperlinkovou maticou H rozumieme štvorcovú maticu $n \times n$ *kde n je počet webových stránok a jednotlivé riadky a stĺpce predstavujú odkazy, ktoré vystupujú resp. vstupujú do webových stránok. Prvky matice sú definované ako* $h_{ij} = \frac{1}{l_i}$, *ak existuje prepojenie so stránky i na stránku j a celkový počet liniek vychádzajúcich so stránky i je* l_i. *Ak takéto prepojenie neexistuje potom* $h_{ij} = 0$. h_{ij} *predstavuje pravdepodobnosť toho že, náhodný používateľ sa vyberie so stránky i na stránku j .*

Hyperlinková matica pre obrázok 6-2 by vyzerala nasledovne

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Následne by sme výpočet page rank pomocou maticového počtu mohli zapísať tak, ako hovorí nasledujúca definícia.

Definícia 6-3: $PR_{i+1}(T) = (1-d)\vec{1} + dH^T PR_i(T)$ *kde,* $PR_{i+1}(T)$ *je stĺpcový vektor i+1 iterácie page rank webových stránok,* $PR_i(T)$ *je i-ta iterácia page rank hodnôt webových stránok, d je damping faktor,* $\vec{1}$ *je stĺpcový vektor pozostávajúci zo samých 1 a* H^T *je transponovaná matica k matici H.*

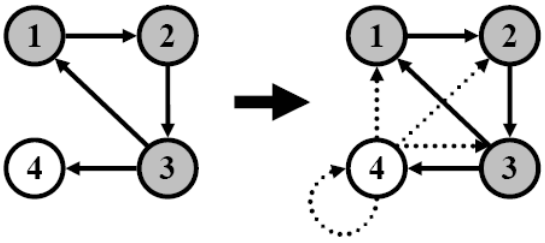
Pre zadanie z príkladu 6-1 by výpočet pomocou maticového počtu vyzeral nasledovne (ukážeme si len prvú iteráciu ostatné by vyzerali analogicky):

$$\begin{pmatrix} PR_1(1) \\ PR_1(2) \\ PR_1(3) \\ PR_1(4) \end{pmatrix} = 0{,}15 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} + 0{,}85 \begin{pmatrix} 0 & 0 & \frac{1}{2} & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0{,}575 \\ 1 \\ 1 \\ 0{,}575 \end{pmatrix}.$$

Page a Brin v nasledujúcej práci (Page, 1998b) rozobrali ďalšie faktory, ktoré vplývajú na výpočet PagerRank algoritmu (dangling links, perzonalizačný vektor). V nasledujúcom odstavci si tieto faktory predstavíme, pričom budeme vychádzať z definície hypertextovej matice .

Ak sa pozrieme na vrchol 4 v grafe na obrázku 6-2 vidíme že, z tohto vrcholu sa používateľ nemá kam dostať (žiadne prepojenie z neho nevychádza) a takéto vrcholy (web stránky) sa nazývajú „dangling nodes“. Takýmito vrcholmi sú napríklad prepojenia na pdf, doc a iné dokumenty, ktoré vlastne najviac vplývajú (obsahovo) na hodnotenie webových stránok. Je veľmi problematické určiť váhu takýchto webových stránok, je ich veľmi veľa a do značnej miery ovplyvňujú predstavenú hyperlinkovú maticu (definícia 6-2). Existuje niekoľko spôsobov riešenia tohto problému pričom my si tu predstavíme riešenie, ktoré navrhli vo svojej práci (Page, 1998b). Toto riešenie je podrobnejšie prezentované v (Wills, 2006).

„Dangling node fix“ je možné urobiť nasledujúcim spôsobom: Predpokladajme že používateľ sa rozhodne z vrcholu ktorý už nikam neukazuje ísť do všetkých *n* vrcholov s určitou pravdepodobnosťou. Toto dokumentuje obrázok 6-3.



Obrázok 6-3. Upravené prepojenia medzi 4 webovými stránkami.

Takýmto spôsobom sa nám pôvodná hyperlinková matica upraví spôsobom, o ktorom hovorí nasledujúca definícia.

Definícia 6-4: Definujeme novú hyperlinkovú maticu $S = H + rw$. *H je pôvodná hyperlinková matica a* $r = (r_1, r_2, ..., r_n)$ *je stĺpcoví vektor s prvkami* $r_i = 1$ *ak* $l_i = 0$ *ináč* $r_i = 0$ *a* $w = (w_1, w_2, ..., w_n)$ *je riadkový vektor kde platí že* $\sum_{i=1}^{n} w_i = 1$.

Hodnoty vektora w určujú pravdepodobnosť navštívenia ľubovoľného vrcholu v grafe z dangling nodu pričom, najčastejšie sa používa rovnomerné rozdelenie pravdepodobnosti $w = \left(\frac{1}{n}, \frac{1}{n}, ..., \frac{1}{n}\right)$.

Upravená matica S pre obrázok 6-3:

$$S = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix}.$$

Ďalším problémom s ktorým sa musíme vysporiadať pri výpočte page rank je pravdepodobnosť že používateľ prestane sledovať sieť prepojení a odskočí na úplne inú webovú stránku. Naša matica *S* sa niečím takýmto nezaoberá a preto, ju musíme upraviť nasledovným spôsobom.

Definícia 6-5: Pre Google maticu $G = dS + (1-d)\vec{1}\vec{v}$ *, kde* $0 \le d < 1$ *je vyššie spomínaný damping faktor,* $\vec{1} = (1, 1, ..., 1)$ *je stĺpcový vektor a* $\vec{v} = (v_1, v_2, ..., v_n)$ *je tzv. perzonalizačný riadkový vektor pre ktorý platí* $\sum_{i=1}^{n} v_i = 1$.

Brin a Page vo svojich prácach (Page, 1998ab) počas vyvíjania PageRank algoritmu ukázali že, damping faktor je vhodné nastaviť na $d = 0,85$ ($1-d$ predstavuje pravdepodobnosť že používateľ prestane sledovať sieť prepojení a odskočí na úplne inú stránku) a perzonalizačný vektor, ktorý hovorí o prioritách používateľa rozdeliť rovnomerne $\vec{v} = \left(\frac{1}{n}, \frac{1}{n}, ..., \frac{1}{n}\right)$. V praxi sa najčastejšie používa damping faktor $d \in \langle 0,85; 0,99 \rangle$. Prezradenie perzonalizačného vektora však môže viesť k tzv. *link spammingu*[20] a z toho dôvodu Google v súčasnosti neprezrádza aký používa perzonalizačný vektor a damping faktor. Link spaming sú napr. tzv. „link farms“, ktoré silným spôsobom (prakticky dookola) ukazujú jedna na druhú čím si umelo zvyšujú page rank ktorý následne predávajú zákazníkom. V roku 2004 však vymysleli Gyöngyi, Garcia-Molina, a Pederson tzv. TrustRank algoritmus (Gyöngyi, 2004) na vytváranie perzonalizačného vektora, ktorý znižuje až eliminuje nebezpečenstvo *link spammingu*. V roku 2006 si spoločnosť Google dala tento algoritmus patentovať.

[20] http://en.wikipedia.org/wiki/Spamdexing#Link_spam

Výpočet page ranku pomocou Google matice *G* prebieha podobne ako sme definovali výpočet page ranku v definícii 6-3.

Definícia 6-6: Page rank webových stránok $PR_{i+1}(T) = PR_i(T)G$, *kde G je Google matica,* $PR_{i+1}(T)$ *je stĺpcový vektor i+1 iterácie page ranku webových stránok,* $PR_i(T)$ *je i-ta iterácia page ranku webových stránok.*

Po následných úpravách dostaneme pre vektor page rank nasledujúci vzťah:

$$G = PR_i(T)\left(dS + (1-d)\vec{1}v\right) = PR_i(T)\left(d\left(H + rw\right) + (1-d)\vec{1}v\right) =$$
$$= PR_i(T)\left(d\left(H + rw\right)\right) + PR_i(T)(1-d)\vec{1}v =$$
$$= dPR_i(T)H + d\left(PR_i(T)r\right)w + (1-d)\left(PR_i(T)\vec{1}\right)v =$$
$$= dPR_i(T)H + d\left(PR_i(T)r\right)w + (1-d)v, \text{ pretože } \left(PR_i(T)\vec{1}\right) = 1$$

pričom za počiatočné hodnoty page rank sa dosádza perzonalizačný vektor

$$PR_0(T) = \vec{v}.$$

Vypočítame teraz príklad 6-1 pomocou Google matice.

Riešenie:

Hodnoty perzonalizačného faktora rozdelíme rovnomerne a 0 iteráciu nastavíme na perzonalizačný vektor $PR_0(T) = \vec{v} = \left(\frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4}\right)$ potom, dostaneme pre maticu G:

$$G = 0,85\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix} + 0,25\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}\left(\frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4} \quad \frac{1}{4}\right) = \begin{pmatrix} \frac{3}{80} & \frac{71}{80} & \frac{3}{80} & \frac{3}{80} \\ \frac{3}{80} & \frac{3}{80} & \frac{71}{80} & \frac{3}{80} \\ \frac{37}{80} & \frac{3}{80} & \frac{3}{80} & \frac{37}{80} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix},$$

teda po prvej iterácií:

$$\left(PR_1(1) \quad PR_1(2) \quad PR_1(3) \quad PR_1(4)\right) = \left(\frac{63}{320} \quad \frac{97}{320} \quad \frac{97}{320} \quad \frac{63}{320}\right).$$

Opäť po 88 iteráciách konverguje výsledný page rank vektor k svojej limitnej hodnote:

$$PR(1) = 0{,}21 \quad PR(2) = 0{,}26$$
$$PR(3) = 0{,}31 \quad PR(4) = 0{,}21 .$$

Na rozdiel od definície 6-1 resp., definície 6-3 page rank výsledné hodnoty podľa definície 6-6 priamo predstavujú rozdelenie pravdepodobnosti a ich suma je 1. Ak by sme chceli dostať hodnoty page rank ako v prvej definícii (od 0 do ∞) potrebujeme vynásobiť jednotlivé pravdepodobnosti počtom všetkých webových stránok (v našom prípade 4) pre ktoré počítame page rank. Keď to však urobíme

$$PR(1) = 0{,}855 \quad PR(2) = 1{,}058$$
$$PR(3) = 1{,}231 \quad PR(4) = 0{,}855$$

a následne porovnáme hodnoty z príkladu 6-1 zistíme, že tieto hodnoty sa vôbec nerovnajú. Na to aby boli hodnoty rovnaké musí byť urobený v definícii 6-1 resp., definícii 6-3 „dangle node fix“.

6.5 Textové operácie

Predpríprava textu pred uložením obsahu dokumentu do indexovej štruktúry je dôležitou súčasťou spracovania pre systémy vyhľadávania informácií. Účelom predprípravy textu je jeho rozdelenie na základné značky a prevedenie hodnôt jednotlivých značiek na základný tvar. Výsledkom procesu predspracovania textu je postupnosť reťazcov normalizovaných na slovné základy, ktoré sú následne ukladané do indexovej štruktúry. Prínosom je redukcia slovníka v indexoch a umožnenie vyhľadávania rôznych morfologických tvarov slov z dopytu na vyhľadávací systém.

Lexikálne analyzátory

Proces rozdelenie textu na základné značky je tiež nazývaný lexikálna analýza textu. Lexery, systémy vykonávajúce lexikálnu analýzu, majú v rámci informatiky tradičné využite pri tvorbe kompilátorov programovacích jazykov, kde sú zodpovedné za predspracovanie a označkovanie textu (kódu) pre generátor kompilátoru. Lexery si svoje využitie našli aj v systémoch na získavanie informácií.

Generované značky (*tokens*) priradzujú jednotlivým častiam textu dodatočnú informáciu o ich lexikálnom význame. Pre ilustráciu uvádzame niekoľko príkladov dvojíc - reťazca znakov a jeho lexikálnej značky: (mrož, WORD); (17.3, FLOAT); (; , SEMICOLON); (16.05.1993, DATE); (http://google.com/, LINK).

Lexikálne analyzátory sú zväčša zložené z dvoch častí, a to zo snímača a značkovača.

Snímač, zväčša realizovaný ako konečný automat, obsahuje pravidlá o možných postupnostiach znakov na základe, ktorých delí text na atomické reťazce spracovávané značkovačom. Značkovač klasifikuje podreťazce textu a prideľuje im značky z definovanej množiny prípustných značiek. Napríklad veta: „Mrzutý mrož monotónne mručal o 10:00“, by mohla byť rozdelená na nasledujúcu množinu hodnota-značka: (mrzutý, WORD); (mrož, WORD); (monotónne, WORD); (mručal, WORD); (o, WORD); (10:00, TIME).

Pri tvorbe lexeru pre IR systémy je výhodné adresovať aj nasledujúce funkcie zľahčujúce spracovanie ďalším modulom: redukcia dokumentu (odstránenie

neužitočných reťazcov), konverzia kódovania dokumentu, zmena znakov z/na kapitály, spracovanie diakritiky, spracovanie rozdelených slov.

Príkladom populárneho lexeru je flex[21].

Identifikácia jazyka

Získanie informácie o jazyku a kódovaní dokumentu je dôležitým krokom pri spracovaní IR systémami.

Dôležité kroky predspracovania dokumentu, ako odstraňovanie stop slov a získavanie základného tvaru slov, sú závislé od použitého prirodzeného jazyka.

Identifikáciu jazyka možno definovať ako klasifikáciu reťazca znakov na základe príslušnosti do prirodzeného jazyka. Výpočtová identifikácia jazyka je založená na štatistických prístupoch, využívajúc techniku *n*-gramov (Dunning, 1994), monte carlo prístup (Poutsma, 2001).alebo kombináciu n-gram a markovovských modelov (Vojtek, 2006).

Stop slová

Výraz stop slová je pomenovaním slov, ktoré sú filtrované pred, resp. po spracovaní textov prirodzeného jazyka. Jedná sa o bežné slová s častým výskytom ktoré nenesú žiadnu významovú informáciu a ich účel v texte je prevažne syntaktický. Príkladom stop slov pre Slovenský jazyk sú napríklad slová: a, o, alebo, ale, sú, táto a podobne. Pri IR systémoch ako aj pri metódach text miningu sú stop slová zväčša úplne ignorované. Tým sa redukuje priestor a zrýchľujú výpočty nad textovými dokumentmi.

Niektoré systémy pre spracovanie textov v prirodzenom jazyku zachádzajú až tak ďaleko, že ignorujú prvých *N* najfrekventovanejších slov dokumentov.

Základný tvar slov

Slová prirodzeného jazyka sa môžu vyskytovať v textoch v rôznych morfologických formách, pričom sú morfologické formy často syntakticky rozdielne. Ukladanie všetkých tvarov všetkých slov v slovníku IR systému je nepraktické z priestorových dôvodov a taktiež komplikuje vyhodnocovanie dopytov, kde používateľ zväčša špecifikuje iba jeden morfologický tvar kľúčového slova. Preto je výhodné reprezentovať rôzne morfologické tvary rovnakého slova práve jedným zastupujúcim termom v slovníku. Takýto zastupujúci term budeme označovať aj ako základný tvar slova. Rozšírenými metódami pre získavanie základného tvaru slova sú lematizácia a *stemming*.

Lematizácia

Účelom lematizácie je nájdenie lemy slova. Výraz lema možno interpretovať viacerými spôsobmi, vzhľadom na kontext. V rámci morfológie je za lemu považovaná kanonická forma lexém slova. Lexéma je abstraktná jednotka reprezentujúca rôzne formy rovnakého slova. Lema je teda reprezentantom skupiny rôznych foriem slova s rovnakým sémantickým významom. Lematizácie je zvyčajne realizovaná pomocou morfologického slovníka, (naplneného ľudskými expertmi) obsahujúceho morfologické varianty slov a k nim priradených liem. Výhodou použitia lematizácie je presnosť

[21] http://flex.sourceforge.net/

kategorizácie slov obsiahnutých v slovníku, teda k slovu získame iba lemy ktorých je morfologickým variantom.

Nevýhod je viacero, a to:

- nemožnosť získať lemu pre slová, ktoré nie sú v slovníku
- jeden reťazec znakov môže byť morfologickým variantom viacerých rozličných liem (narp. Slovenské slovo „mier“ môže byť morfologickým variantom významovo rozličných slov: „mier“, „miera“ „mieriť“). Pre zachovanie presnosti je potom potrebné uchovávať aj viac ako jednu lemu pre slová dokumentov v indexových štruktúrach.

WordNet[22] (Miller, 1990) je príkladom populárneho výkladového slovníka anglického jazyka, ktorý obsahuje aj lemy morfologických variant slov.

Stemming

Stemming je proces redukcie slov na ich koreň, základný tvar. Koreňom budeme nazývať časť slova, ktorá je rovnaká pre všetky morfologické varianty slova. Koreň získaný stemmingom nemusí byť nutne zhodný s morfologickým koreňom slova. Zvyčajne stačí ak sú morfologické varianty projektované na rovnaký koreň.

Stemming je populárnou metódou pre získavanie základných tvarov slov, pretože je algoritmizovateľný. Boli vyvinuté postupy, ktoré algoritmicky redukujú slová prirodzeného jazyka na korene. Stemmovacie algoritmy sú závislé od konkrétneho prirodzeného jazyka (stemmer pre anglický jazyk nebude produkovať dobré výsledky pre slovenský). Naviac, stemovacie algoritmy vykazujú dva druhy nepresností a to určenie rovnakého koreňa pre sémanticky rôzne slová (pre-stemmovanie), určenie viacerých rôznych koreňov pre morfologické varianty jedného slova (pod-stemmovanie). Pre-stemmovanie a pod-stemmovanie sú zároveň chybovými mierami pre vyhodnotenie presnosti stemmovacieho algoritmu. Snahy znížiť jednu z týchto mier často vedie k zvýšeniu druhej.

V súčasnosti existujú rôzne prístupy k stemmovacím algoritmom, od brute-force algoritmov využívajúcich vyhľadávacie tabuľky, cez algoritmy orezávajúce sufixy a afixy, až po stochastické algoritmy vytvárajúce pravdepodobnostným model na základe známych relácií medzi koreňmi a morfologickými tvarmi slov.

Zložitosť stemmovacích algoritmov závisí aj od prirodzeného jazyka, pre ktorý sú navrhnuté. Jazyky s bohatou morfológiou (napr. slovanské) sú vo všeobecnosti ťažšie pre tvorbu stemmerov.

Populárnym algoritmom pre stemming anglického jazyka je Porterov algoritmus (Porter, 1990, Rijsbergen, 1980).

Lematizácia a stemming v slovenčine

Slovenčina patrí medzi jazyky ktoré majú bohatú morfológiu, teda tvar slova sa mení podľa významu. Toto môže byť zjavnou nevýhodou pri počítačovom spracovaní. Je teda potrebné hľadať základný tvar slova (*lemma*) prípadne koreň slova (*stem*). V angličtine sa tiež používa lematizácia alebo steming aby sa odstránilo množné číslo, minulý čas slovies a ďalšie. V angličtine je najbežnejší Porterov algoritmus ktorý však

[22] http://wordnet.princeton.edu/

na slovenčinu nefunguje. Keď si predstavíme napríklad koreň slova „rada“ koreň je „rad“ pričom tento koreň zahŕňa pri uvažovaní bez diakritiky nasledovné slová: rada – podstatné meno, orgán; rád – podstatné meno, vyznamenanie; rád – sloveso; rad – podstatné meno, zoradenie; rada – podstatné meno, ponaučenie.

Problémom je aj rôzne kódovania slovenčiny v ktorých sú zapísané informačne zdroje (napr. web stránky). Základné sú win-1250, ISO-8859-2 alebo UTF ale tiež špeciálne HTML značky začínajúce „&#“. Veľa informačných zdrojov je písaných aj bez diakritiky napríklad ak chceme spracovávať emaily ktoré sa bežne píšu bez diakritiky je potrebne riešiť lematizáciu aj bez diakritiky.

Lematizácia a stemming sa bežne používa najme pri indexácii a následnom fulltextovom vyhľadávaní. Treba však povedať že na slovenčinu zatiaľ niečo podobné neexistuje v uspokojivom rozsahu.

Firma Forma s.r.o. [23] deklaruje že má vyriešené indexovanie a vyhľadávanie v slovenskom jazyku a teda aj stemming. Pri vyhľadávaní na stránke www.zbierka.sk však toto funguje iba v obmedzenom rozsahu. Napríklad pri dopytoch „Národná rada“, „Národnej rady“ a bez diakritiky „Narodna rada“ je výsledkom rôzny počet dotazov ktorý nezahŕňa všetky formy slova.

Podobne vyhladávač morfeo.sk[24] deklaruje vyhľadávanie v slovenčine ale napríklad pre dopyty „Štefan Luby“ a „Štefanovi Lubymu“ vráti rôzny počet stránok. Samozrejme tvary slov nie sú vyriešené ani vo vyhľadávačoch Zoznam a Google.

Na lematizátore a stemmeri pracujú aj na niektorých pracoviskách v ČR, nepodarilo sa nám však zistiť ako sú tieto výsledky úspešné na slovenčine. Zaujímavým pre tvorbu slovenského stemmeru sa javí práca Lea Galamboša[25] ktorý vyvinul stemer vhodný pre slovanské jazyky (Galambos, 2001, 2004ab), na základe ktorého bol vyvinutý stemmer pre poľský jazyk[26]. Na lematizátore (Garabík, 2007) pre slovenčinu sa pracuje na JULS SAV, ktorý je založený na slovníkovom princípe. Beta verzia je dostupná na webe[27].

V rámci projektu NAZOU je na UPJŠ vytváraný nástroj Tvaroslovník (Krajči, 2007), ktorý bude slúžiť na lematizáciu slovenského jazyka. Tento nástroj vie na základe vzorov vygenerovať lemu, ktorá sa následne kvôli presnosti overuje v slovníku.

Tvorba algoritmického stemmeru pre slovenčinu je potrebná, pretože stemmery založené na slovníkoch nevedia zistiť lemy alebo korene slov pre nové slová, priezviská, mená miest alebo mená firiem v inom ako základnom tvare. Lematizácia pritom nie je potrebná len pre správnu indexáciu a fulltextové vyhľadávanie, ale aj pre sémantickú anotáciu, identifikáciu informačných zdrojov v rámci domény a ďalšie. Tento problém by mohol čiastočne vyriešiť nástroj Tvaroslovník (Krajči, 2007), prípadne bude potrebné vyvinúť nový nástroj na stemming slovenčiny na základe existujúcich prístupov.

---

[23] http://www.forma.sk/

[24] http://www.morfeo.sk/

[25] http://kocour.ms.mff.cuni.cz/~galambos/

[26] http://getopt.org/stempel/

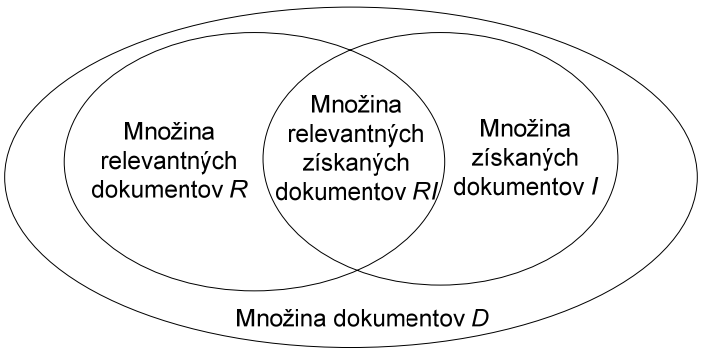
[27] http://korpus.juls.savba.sk/~garabik/junk/mlv/

6.6 Hodnotenie úspešnosti

Softvérové systémy sa najčastejšie hodnotia podľa systémových charakteristík ako čas za ktorý daný systém vykoná všetky potrebné operácie a priestor na disku alebo inom zariadení ktorý sa pri vykonávaní týchto operácií spotrebuje. Z týchto charakteristík jasne vyplýva že, čím menej času a priestoru systém potrebuje tým je úspešnejší. Takéto hodnotenie úspešnosti systémov sa označuje *performance evaluation*. V prípade systémov ktoré získavajú dáta sú systémové charakteristiky veľmi dôležité a asi aj najpodstatnejšie. Ak však bol systém navrhnutý ako IR systém okrem času a priestoru vplývajú na úspešnosť systému aj iné charakteristiky ktorým sa budeme venovať v tejto podkapitole. Keďže používateľova požiadavka pre získanie určitých dát pomocou systému je obyčajne dosť vágna výsledok ktorý, vráti vyhľadávací systém (pole dokumentov) určite nebude celkom presný (teda nie celkom to čo ten používateľ požaduje). Teda je nutné hodnotiť nakoľko je zoznam dokumentov vrátený systém presný. Takémuto vyhodnocovaniu úspešnosti sa hovorí *retrieval performance evaluation*. Veľmi podrobne je rozobraný v (Baeza-Yates, 1999)

Retrieval performance evaluation IR systémov funguje na nasledovnom princípe. Máme množinu dokumentov a jedného alebo viacerých používateľov ktorí, na základe vstupnej požiadavky vytvoria podmnožinu relevantných dokumentov. Následne takisto IR systém vytvorí podmnožinu relevantných dokumentov. Tieto dve množiny dokumentov porovnáme a vyhodnotíme dve základné charakteristiky presnosť (*precision*) a pokrytie (*recall*). Neskôr ukážeme že tieto dve charakteristiky nie sú celkom postačujúce v hodnotení úspešnosti IR systémov a že, potrebujeme zadefinovať ďalšie charakteristiky (E, F_1 štatistiku).

Predstavme si skupinu dokumentov $D = \{D_1, D_2, ..., D_n\}$ a požiadavku Q pre ktorú je $R = \{R_1, R_2, ..., R_r\}$ množina relevantných dokumentov kde $|R|$ je počet relevantných dokumentov. Predstavme si že, systém na vyhľadávanie informácií vráti na požiadavku Q zoznam dokumentov $I = \{I_1, I_2, ..I_i\}$ a teda počet získaných dokumentov je $|I|$. Ďalej definujeme množinu $RI = R \cap I$ (obrázok 6-4).



Obrázok 6-4. vzťah medzi množinami dokumentov (relevantné a získané).

Na základe obrázku 6-4 môžeme zadefinovať dve základné charakteristiky pre vyhodnocovanie IR systémov a to: presnosť (*precision*) a pokrytie (*recall*).

Definícia 6-7: $P = \frac{|RI|}{|I|}$. *Presnosť je podiel počtu získaných relevantných dokumentov* ($|RI|$) *k počtu všetkých získaných dokumentom* ($|I|$).

Definícia 6-8: $R = \frac{|RI|}{|R|}$. *Pokrytie je podiel počtu získaných relevantných dokumentov* ($|RI|$) *k počtu relevantných dokumentov* ($|R|$).

Medzi presnosťou a pokrytím je veľmi úzka spojitosť pretože zvyšovanie jednej prináša zvyčajne pokles druhej charakteristiky. Žiadna sama o sebe nemôže vypovedať o úspešnosti systému na vyhľadávanie informácií. Na nasledujúcom príklade si predvedieme výpočet týchto dvoch charakteristík.

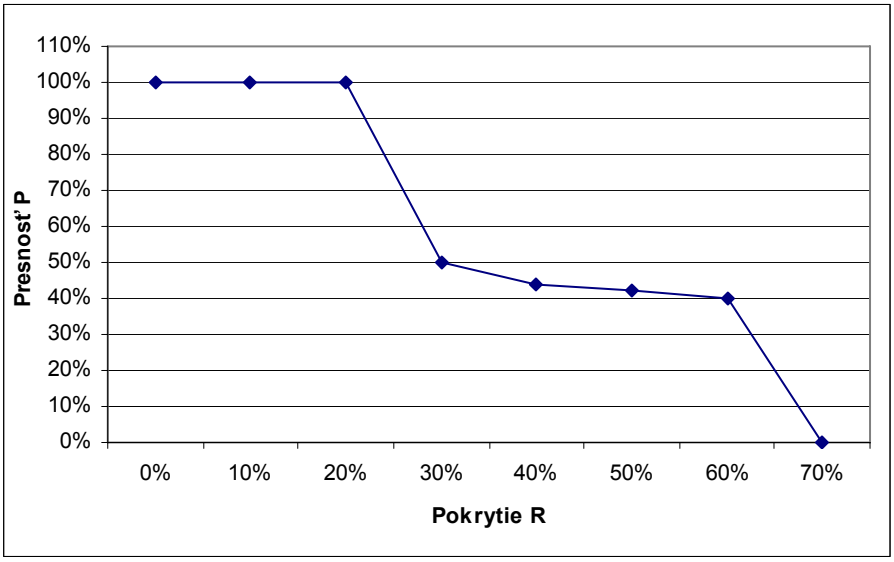
Príklad 6-2: Máme množinu dokumentov D ktorá obsahuje 100 dokumentov týkajúcich sa cestovných kancelárií. Máme dotaz Q ktorý hovorí že požadujeme tie cestovné kancelárie ktoré poskytujú dovolenky v Grécku a na priľahlých ostrovoch. Tým špecialistov vybral relevantnú množinu R obsahujúcu 10 dokumentov $R = \{r_{45}, r_{93}, r_{22}, r_{72}, r_3, r_{30}, r_{65}, r_{55}, r_{34}, r_{10}\}$ *v poradí od najrelevantnejšieho. Povedzme že systém vrátil 15 dokumentov daných touto množinou.* $I = \{i_3, i_{10}, i_{44}, i_7, i_{17}, i_{93}, i_5, i_{82}, i_{22}, i_{11}, i_{13}, i_{72}, i_2, i_{29}, i_{55}\}$. *Následne po prieniku týchto dvoch množín dostávame* $RI = \{ri_3, ri_{10}, ri_{93}, ri_{22}, ri_{72}, ri_{55}\}$. *Ak by sme počítali presnosť a pokrytie po jednotlivých dokumentoch prieniku pre prvý prienik by sme dostali* $P = 100\%$ *(jeden výber a relevantný dokument) a* $R = 10\%$ *(máme 1 relevantný z 10-tich relevantných), pre druhý výber* $P = 100\%$ *a* $R = 20\%$, *pre tretí* $P = 50\%$ *(máme tri dokumenty správne zo šiestich vybraných) a* $R = 30\%$, *atď.*

Výsledky sa nachádzajú v nasledujúcej tabuľke 6-1 a na obrázku 6-5, ktorý hovorí o vzťahu medzi pokrytím a presnosťou.

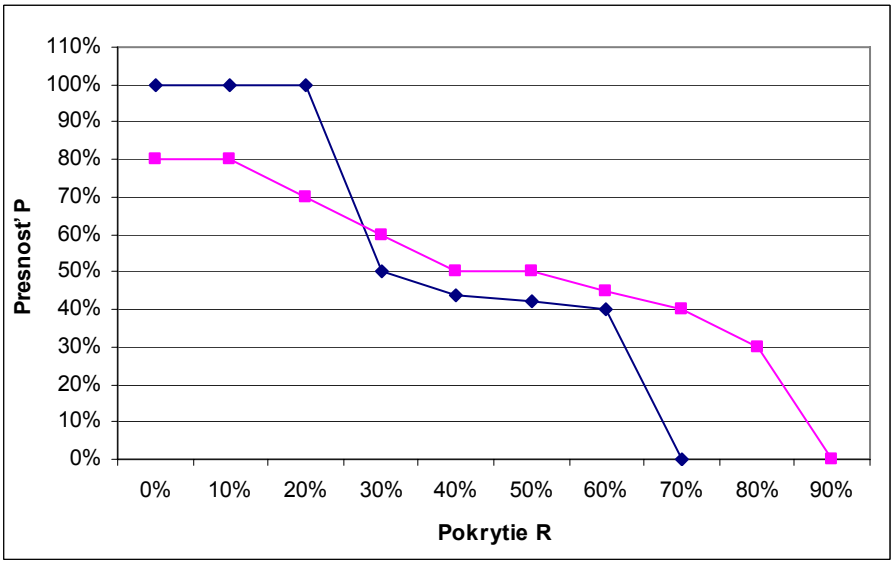
Tabuľka 6-1. Tabuľka opisujúca závislosť medzi presnosťou a pokrytím.

	1	2	3	4	5	6
Presnosť (P)	100%	100%	50%	44%	42%	40%
Pokrytie (R)	10%	20%	30%	40%	50%	60%

Hodnotu pre 0% pokrytie sme získali pomocou interpolácie a presnosť pre pokrytie väčšie ako 60% klesne na 0% pretože, nie všetky relevantné dokumenty boli systémom získané. Vo všeobecnosti je vzťah medzi presnosťou a pokrytím presne taký istý ako ukazuje obrázok 6-5. Pokiaľ jedna charakteristika stúpa druhá klesá. Otázkou zostáva podľa ktorej štatistiky vyhodnotiť úspešnosť systému na vyhľadávanie informácií a ako navzájom porovnávať rôzne systémy na získavanie informácií. Na obrázku 6-6 vidíme porovnanie dvoch rôznych systémov pomocou kriviek presnosť/pokrytie.



Obrázok 6-5. Vzťah medzi presnosťou a pokrytím.



Obrázok 6-6. Porovnanie dvoch systémov na vyhľadávanie informácií podľa kriviek presnosť/pokrytie.

Ako je vidieť jeden systém je úspešnejší pri nižšom pokrytí ale zase je menej úspešný pri vyšších pokrytiach. Je dosť veľkým problémom ako vyhodnotiť ktorý zo systémov je „lepší“ pretože toto je v podstate veľmi subjektívna otázka a sú používatelia ktorí, požadujú vysoké pokrytie a nevadí im nižšia presnosť a naopak sú používatelia ktorí, vyžadujú vysokú presnosť aj pri nie celkom dobrom pokrytí.

V príklade 6-2 sme vyhodnocovali iba jednu požiadavku systému. Samozrejme pri prevádzaných testoch, či už v laboratórnych podmienkach alebo na reálnych dátach vyhodnocujeme viac požiadaviek ako len jednu. V takomto prípade potom musíme jednotlivé výsledky spriemerovať. Najčastejšie používaný spôsob spriemerovania je spriemerovať presnosť na rôznych úrovniach pokrytia.

Definícia 6-9: $\overline{P}(r) = \sum_{i=1}^{n} \frac{P_i(r)}{n}$. *Kde* $\overline{P}(r)$ *je priemerná presnosť pre úroveň pokrytia r, n je počet požiadaviek zaslaných do systému a* $P_i(r)$ *je presnosť na úrovni pokrytia r pre i-tu požiadavku.*

Ako sme už spomenuli v predchádzajúcej časti, presnosť a pokrytie nevypovedajú celkom o kvalite systému pre získavanie informácií aj keď, ich popularita je stále veľká. Preto sa v tejto časti budeme zaoberať inými metódami vyhodnotenia systémov na vyhľadávanie informácií.

Jedným z riešení je použiť tzv. F_1-štatistiku (niekde uvádzanú iba ako F) (Shaw, 1997) o ktorej hovorí nasledujúce definícia:

Definícia 6-10: $F_1(i) = \frac{2}{\frac{1}{R(i)} + \frac{1}{P(i)}}$ *kde,* $F_1(i)$ *je harmonický priemer pre i-ty dokument v usporiadanom zozname, a* $R(i)$ a $P(i)$ *sú pokrytie a presnosť pre i-ty dokument.*

Tento vzťah sa dá ešte prepísať nasledovným spôsobom $F_1(i) = \frac{2R(i)P(i)}{R(i)+P(i)}$ (vo viacerých literatúrach je používaný práve tento vzťah). F_1 štatistika nadobúda hodnoty z intervalu $\langle 0;1 \rangle$ a čím je vyššia tým je systém na vyhľadávanie informácií úspešnejší. V matematickej štatistike sa F_1 štatistika označuje ako harmonický priemer. Keďže F_1 štatistika je kombináciou presnosti a pokrytia nadobúda najvyššie hodnoty tam kde, sú presnosť a pokrytie najvyššie. Z toho vlastne vyplýva že, dosiahnutie vysokej F_1 štatistiky je vlastne hľadaním kompromisu medzi presnosťou a pokrytím.

Ďalším riešením je použiť tzv. *E* (*Error*) štatistiku predstavenú Rijsbergenom v (Rijsbergen, 1979).

Definícia 6-11: $E(i) = 1 - \frac{1+b^2}{\frac{b^2}{R(i)} + \frac{1}{P(i)}}$ *kde,* $E(i)$ *je E štatistika pre i-ty dokument v usporiadanom zozname,* $R(i)$ a $P(i)$ *sú pokrytie a presnosť pre i-ty dokument v utriedenom zozname a b je používateľom špecifikovaný parameter ktorý vyjadruje relatívnu dôležitosť ktorú používateľ prideľuje presnosti alebo pokrytiu. V prípade že parameter* $b = 1$ *je E štatistika doplnkom k* F_1 *štatistike. Ak je* $b > 1$ *používateľ prikladá väčšiu dôležitosť presnosti ako pokrytiu a naopak keď je* $b < 1$ *prikladá väčšiu váhu pokrytiu ako presnosti. Čím je bližšie E štatistika k 0 tým je IR systém*

úspešnejší. V prípade že, $b \to 0, \text{resp.}\, b \to \infty$ *neprikladá používateľ žiadnu váhu pokrytiu resp. presnosti.*

6.7 Softvérové knižnice a systémy na podporu vyhľadávania

Apache Lucene

Apache Lucene [28] je výkonný vyhľadávací nástroj ktorý podporuje fulltextové vyhľadávanie. Software je vytvorený ako knižnica v jazyku Java a je ho možné použiť v rôznych aplikáciách kde je potrebné fulltextové vyhľadávanie, indexovanie alebo podpora textových operácií na dokumentoch. Lucene je vyvíjaný ako open source.

mnoGosearch

mnoGoSearch[29] je fulltextový vyhľadávací stroj vhodný najmä na aplikácie ktoré ako zdroj dokumentov využívajú intranet alebo internet. Je zložený z dvoch častí, indexovací mechanizmus ktorý vie spracovať rôzne typy dokumentov (HTML, PDF, DOC, XLS, MP3) za podpory ďalších voľne dostupných programov, pričom tiež podporuje rôzne protokoly ako HTTP, FTP , NEWS protokol alebo aj lokálne súbory.

Druhou časťou systému je podpora vyhľadávacieho rozhrania, ktoré je dostupné vo verzii PHP alebo ako CGI skript.

Systém je možné rozširovať prispôsobovať a konfigurovať podľa potreby aj vzhľadom nato že je vyvíjaný ako open source.

Textové operácie

Knižnica QTag [30] je pravdepodobnostný POS značkovač, teda zisťuje slovný druh jednotlivých slov pre anglický jazyk.

Knižnica SimMetrics [31] obsahuje sadu algoritmov na zisťovanie podobností slov napríklad na základe počtu Levenshteinových operácií alebo klasickej kosínusovej vzdialenosti spomínanej vyššie pre podobnosť dokumentov.

Na stemovanie angličtiny je možné použiť Porterov algoritmus ktorý je súčasťou aj knižnice Lucene. Na lematizáciu slovenčiny je možné použiť beta verziu lematizátora[32] vytvoreného na JULŠ SAV alebo nástroj Tvaroslovník z projektu NAZOU (Krajčí, 2007).

Na správne použitie lematizačného algoritmu je najskor potrebné zisťiť jazyk dokumentu. Toto je možné pomocou nástroja NALIT (Vojtek, 2006) z projektu NAZOU.

---

[28] http://lucene.apache.org/

[29] http://www.mnogosearch.org/

[30] http://www.english.bham.ac.uk/staff/omason/software/qtag.html

[31] http://www.dcs.shef.ac.uk/~sam/simmetrics.html

[32] http://korpus.juls.savba.sk/morphology/

6.8 Zhrnutie

Oblasť vyhľadávania sa v poslednej dekáde intenzívne rozvíja. Nové poznatky z vedy a výskumu sa dajú veľmi dobre overiť v praxi, tým že ich zavedenie do prostredia internetu rýchlo poskytne spätnú väzbu inovatívnym riešeniam. Príkladom je spoločnosť Google a ich PageRank algoritmus.

Napriek tomu že sa táto oblasť rozvíja a venuje sa jej množstvo ľudí v oblasti výskumu aj komerčnej sféry, proces digitalizácie a nárast množstva informácií čoraz viac zvyšuje tlak na hľadanie nových, inovatívnych riešení v tejto oblasti.

V súčasnosti už zväčša nie je problém dostupnosť alebo cena informácií, ale identifikácia relevantnej informácie v krátkom čase. Ak sa aj požadované informácie nachádzajú v niektorom z dokumentov v IR systéme, dopracovať sa k relevantným dokumentom je často netriviálne – proces zaberá príliš veľa času, sme zahltený veľkým množstvom informácií ktoré sú len s časti relevantné, alebo potrebné dokumenty nie sú identifikované vôbec.

Súčasťou trendov v tejto oblasti je aj sémantický web, ktorého cieľom je aby webu nerozumel len človek ale aj stroje. Táto oblasť výskumu zatiaľ neprerazila úplne do praxe. Problémom je že síce existujú štandardy na sémantický popis a anotáciu dokumentov a objektov na internete avšak tvorcovia web stránok ich zatiaľ nepoužívajú pretože v tom zatiaľ nevidia výhodu. Automatické riešenia na tvorbu takýchto sémantických popisov sú zatiaľ iba čiastočne úspešné. Výhoda z nových služieb však môže prísť až potom, keď počet sémanticky opísaných dokumentov na internete dosiahne kritickú hranicu. Či už bude sémantický web úspešný alebo nie, ciele pre ktoré bol navrhnutý ostávajú. Cieľom je aby človek nemusel tráviť čas s vyhľadaním relevantných informácií ale aby ich našiel rýchlo, aby ich vôbec našiel a najlepšie s pomocou softvéru, ktorý by rozumel požiadavkám a kontextu používateľa.

Použitá literatúra

BAEZA-YATES, R. - RIBEIRO-NETO, B. (1999) *Modern Information Retrieval*, Addison Wesley; 1st edition, ISBN-10: 020139829X, ISBN-13: 978-0201398298

BRIN, S. - MOTWANI, R. - PAGE, L. – WINOGRAD, T.. (1998) What can you do with a Web in your pocket? *Data Engineering Bulletin*, 21:37–47.

DUNNING, T. (1994) *Statistical identification of language* , Computing Research Lab (CRL), New Mexico State University, Tech. Report MCCS-94-273, http://www.comp.lancs.ac.uk/computing/research/ucrel/papers/lingdet.ps

FURDÍK, K. (2003): *Získavanie informácií v prirodzenom jazyku s použitím hypertextových štruktúr*. Doktorandská dizertačná práca, FEI TU Košice.

GALAMBOS, L. (2001): Lemmatizer for Document Information Retrieval Systems in JAVA. Leszek Pacholski, Peter Ruzicka (Eds.): *SOFSEM 2001: Theory and Practice of Informatics*, 28th Conference on Current Trends in Theory and Practice of Informatics Piestany, Slovak Re-public, November 24 - December 1, 2001, Proceedings. Lecture Notes in Computer Scien-ce 2234 Springer 2001.

GALAMBOS, L. (2004a): *Multilingual Stemmer in Web Environment*. PhD Thesis, Faculty of Mat-hematics and Physics, Charles University in Prague, 2004.

GALAMBOS, L. (2004b): Semi-automatic stemmer evaluation. Mieczyslaw A. Klopotek, Slawomir T. Wierzchon, Krzysztof Trojanowski (Eds.): Intelligent Information Processing and Web Mining, *Proceedings of the International IIS*: IIPWM'04 Conference held in Zakopane, Po-land, May 17-20, 2004. Advances in Soft Computing Springer 2004, ISBN 3-540-21331-7.

GARABÍK, R. (2007): Slovak morphology analyzer based on Levenshtein edit operations, *1st Workshop on Intelligent and Knowledge oriented Technologies - WIKT 2006 Proceedings*; Michal Laclavik, Ivana Budinska, Ladislav Hluchy (Eds.); November 28 - 29, 2006; Bratislava, Slovakia; March 2007; ISBN 978-80-969202-5-9; March 2007

GOLUB, G. - VAN LOAN, C. (1996) *Matrix computations*, 3rd ed., The Johns Hopkins University Press.

GURSKÝ, P. - LENCSES, R. - VOJTÁŠ, P. (2005): Algorithms for user dependent integration of ranked distributed information, in *TCGOV 2005 Poster Proceedings*, M. Boehlen et al eds. IFIP - Universitaetsverlag Rudolf Trauner, Laxenburg, ISBN 3-85487-787-0, pp. 123-130

GYÖNGYI, Z. - GARCIA-MOLINA, H. - PEDERSEN, J. (2004) Combating Web spam with TrustRank, *Proceedings of the 30th International Conference on Very Large Databases*, Morgan Kaufmann, pp. 576–587.

Krajči, S. – Novotný, R. (2007): Hľadanie základného tvaru slovenského slova na základe spoločného konca slov, *1st Workshop on Intelligent and Knowledge oriented Technologies - WIKT 2006 Proceedings*; Michal Laclavik, Ivana Budinska, Ladislav Hluchy (Eds.); November 28 - 29, 2006; Bratislava, Slovakia; March 2007; ISBN 978-80-969202-5-9; March 2007

KOSTELNÍK, P. (2000): *Získavanie informácií s využitím algoritmov zhlukovej analýzy*, Diplomová práca, Laboratórium umelej inteligencie, http://neuron-ai.tuke.sk/~kostelni/zdroje/prace/diplom/index.html

KOŠIAĽ, I. (2002): *Spracovanie a kategorizácia textových dokumentov pre účely semiautomatického linkovania na znalostný model*. Košice. Diplomová práca. Katedra kybernetiky a umelej inteligencie, Technická univerzita v Košiciach.

MILLER, G. (1990): WordNet: An On-line Lexical Database, Special Issue, *International Journal of Lexicography*, Vol. 3, Num. 4

PAGE, L. – BRIN , S (1998a): The anatomy of a large-scale hypertextual Web search engine, *Computer Networks and ISDN Systems* 33, 107–117.

PAGE, L. – BRIN, S. – MOTWANI, R – WINOGRAD, T. (1998b). *The PageRank Citation Ranking: Bringing Order to the Web. Technical report*, Stanford Digital Library Technologies Project, 1998.

POUTSMA, A. (2001): *Applying Monte Carlo Techniques to Language Identification*, SmartHaven, Amsterdam, CLIN 2001: www.xs4all.nl/~ajwp/langident.pdf

PORTER M. (1997): An algorithm for suffix stripping, *in Readings in Information Retrieval*, San Francisco: Morgan Kaufmann,ISBN 1-55860-454-4.

RIJSBERGEN, C. (1979) *Information Retrieval*, Butterworths

RIJSBERGEN, C. – ROBERTSON, S. - PORTER, M. (1980): New models in probabilistic information retrieval. *British Library Research and Development Report, no. 5587*, British Library, London, 1980.

SALTON, G. – BUCKLEY, C.: Term weighting approaches in automatic text retrieval. *In: Information Processing and Management*, 24(5), 1988, pp.513-523.

SALTON, G. – YANG, C. – YU, C.: A theory of term importance in automatic text analysis. In: *Journal of the American Society for Information Science*, 1975, s.237-253.

SHAW, JR., W. M., BURGIN, R., & HOWELL, P. (1997). Performance standards and evaluations in IR test collections: Vector-space and other retrieval models. Information Processing and Management, 33(1), 15–36.

TVAROŽEK, M. - BIELIKOVA, M. (2007): Adaptive Faceted Browser for Navigation in Open Information Spaces. In: *Proc. of WWW 2007*, pages 1311–1312. ACM

VOJTEK, P. – GRLICKÝ, V. (2006): Identification of Natural Language using n-grams and Markov processes, In: *Tools for Acquisition, Organisation and Presenting of Information and Knowledge*. P.Navrat et al. (Eds.), Vydavatelstvo STU, Bratislava, 2006, pp.154-161, ISBN 80-227-2468-8. Workshop 26-28 September, Nizke Tatry, Slovakia.

WILLS, R. (2006) *Google's PageRank: The Math Behind the Search Engine*, Department of Mathematics, North Carolina State University, http://www4.ncsu.edu/~ipsen/ma798I/Wills.pdf

7 FIPA COMPLIANT MULTI-AGENT SYSTEMS

Multi Agent Systems technology is relatively new research area. Agent Technology seemed to be very promising since it has proposed a new paradigm for software technologies, from Object Oriented Programming (OOP) to Agent Oriented Programming (AOP). Recent technologies such as Web services, SOAP, XLM-RPC or Peer-to-Peer computing came also out of MAS but are only partial realizations of the potential of MAS.

Agents are completely different from other software technologies because all others are mainly function based technologies. The agent technology can be understood as the next step from OOP to Agent Oriented Programming (AOP).

Term agent or multi-agent systems are understood in different ways. To answer why it is needed to specify what kind of agents we are dealing with, it is only necessary to cite Jennings, one of the leading researchers in the field:

> *"Agents are like OOP in the 80's, everybody talk about it, every company tried it, every solution has it, every manager wants if, but nobody just know what it is ..."*

When we talk about agents we mean agents in a Multi-Agent system, where more than one agent is present, and where agents can interact. The term "agent" can be understood differently depending on the focus of research. In Figure 7-1 a classification of agents is presented.

In the literature many definitions of an agent can be found. Within this chapter, the AgentLink II definition for a software agent is used:

> *An agent is a computer system capable of flexible autonomous action in a dynamic, unpredictable and open environment.* (Luck, 2003)

Robotic agents are based on hardware. Software agents are based on a software program. Simulation Agents are software agents, which help to simulate a discrete system, which cannot be described by differential equations or which are too complex e.g. simulating traffic in the city. The environment is the road system (software representation) and agents will be programmed to move on a road, to slow down when another agent car is close by and maybe other simple behaviors. This way we can simulate a traffic flow and then use results in the real situation. The best known example is simulation of ants behavior. More than in other agent fields it is valid that *"Smart things can be done with Dumb Agents"* (Willmott, 2003).

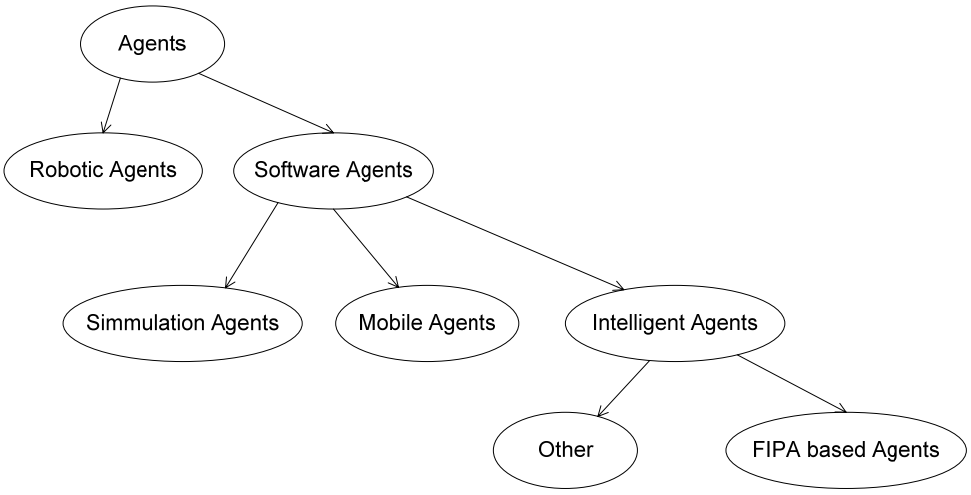


Figure 7-1. Agent Classification.

Mobile Agents are software agents which can move from one place to another. There are 2 types of mobility – strong and weak. The strong mobility means a migration of an agent with its execution state and its variables values from one computer to another. The weak mobility is when an agent migrates and caries only the code and variables values.

Intelligent Agents act on their software environment called the agency. Agents interact by passing a message among each other. They use techniques from the artificial intelligence area such as learning, reasoning or negotiation and decision support. Intelligent Agents sense and act via message passing. They can have also some other sensors to external systems and other methods to act on certain external systems. Intelligent Agents can be mobile if needed.

FIPA based agents follow FIPA[33] standards. Interaction among agents is done by the FIPA Agent Communication Language (ACL) (FIPA ACL, 2002). Agents communicate using an ontology and a content language. The agent platform needs to support a directory facilitator where all agents can register.

Many combinations between robotic and software agents or intelligent and mobile agents exist. This summary is given, because it is not clear in scientific community what is understand behind word agent.

Thus here agents are understood as software, intelligent and FIPA based Agents.

7.1 Agent Architectures and Models

Agent architectures are the fundamental engines underlying the autonomous components that support effective behavior in the real-world, dynamic and open environments. Initial efforts in the field of agent-based computing focused on the development of intelligent agent architectures, and the early years established several

[33] http://www.fipa.org/

lasting styles of the architecture. These range from purely reactive (or behavioral) agents that operate in a simple stimulus-response fashion, such as those based on the Subsumption Architecture of Brooks (Brooks, 1991A) (Brooks, 1991B) at one side, to more deliberative agents that reason about their actions, such as the class of the belief-desire-intention (BDI) agents that are increasingly prevalent (also in commercial products such as JACK[34]), on the other side. In between the two there are hybrid combinations of both, or layered architectures, which attempt to involve both a reaction and deliberation in an effort to adopt the best of each approach. Increasingly more sophisticated agents than the traditional BDI kind have also been developed, but the benefits of the increased sophistication is largely confined to well-defined areas of a need rather than offering general solutions (Luck, 2003).

Internal Agent Architectures:

- Reactive Architecture
- Belief Desire Intention Architecture – BDI
- Behavioral Architecture

Reactive Architecture is the simplest model, where agents perform predefined actions on some environmental state (e.g. thermostat).

BDI architecture is a model where an agent is divided into 3 main parts:

- Beliefs – this is agent knowledge about the current state of its environment;
- Desires – goals to achieve;
- Intentions – plans to act upon to achieve the desires.

BDI architecture (Wooldridge, 2002) is well known and lot of work has been done on it, but it has never been used in any real commercial application. However, many of its features were used later in hybrid architectures, and also most of current architectures have some aspects in common with BDI.

In the Behavioral Architecture (Wooldridge, 2002), an agent has several behaviors which are executed in sequence or on multiple levels. Some behaviors can manage and execute others etc. This architecture is most suitable for real software development, and the architecture presented in this thesis is built on such architecture.

The main focus in literature is on the externals of the agents, their communication with environment and other agents. The internal knowledge model is left for an agent creator. There are several tools, which allow to create BDI based agents but this is not sufficient for any real system, only for some simulations and tests. FIPA does not cover this area of agent systems either. FIPA specifications just describe how agents should communicate and how they can share, translate or communicate ontologies. In FIPA complaint implementations of an agent system, different approaches for building the agent knowledge model can be found.

The most advanced, but not sufficient approach is in the JADE agent system. JADE support for ontologies or agent knowledge modeling (Caire, 2002) is based on java classes. Ontology elements and its relations and properties are described as a real java object. This is powerful for its manipulation, when developing an agent code and

[34] http://www.agent-software.com/shared/home/

"brain" of the agent. Instances of ontology classes can be passed in ACL messages (see Figure 7-2) in the form of FIPA-SL language (FIPA SL, 2002). FIPA-SL is based on predicate logic.

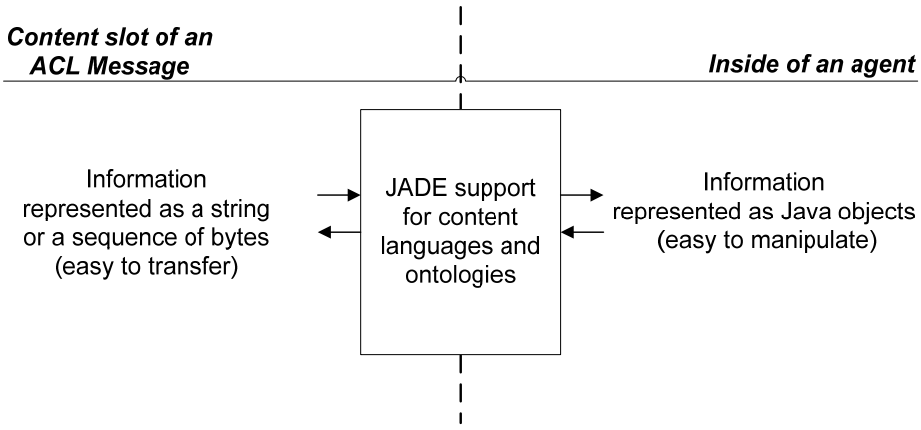


Figure 7-2. The conversion performed by the JADE support for content.

JADE developers have also defined the so called "Content Reference Model" (see Figure 7-3). This is a model for an agent memory and its basic functionalities. The model has 2 main elements – Predicate and Concept. The concept is any object of the agent environment. Predicate connects concepts and it is a certain statement about, which can be true or false. Predicates can form questions. Very important concept is AgentAction, which defines the actions an agent can take, and it can present agent goals. The agent model can be built by protégé ontology editor[35] and then exported to JADE ontology model by protégé bean generator plug-in[36].

The JADE agent model is not sufficient in several ways. A model based on java classes can not support multiple inheritance, inverse concepts and other features of semantic ontology representations such as DAML+OIL or OWL. The JADE ontology consists of Predicates to be able to create predicate logic queries and statements because in the ACL message it is transformed into the FIPA-SL language. This can be good for some applications but for knowledge management or experience management we did not find it useful for several reasons.

The predicate logic and similar structures as the JADE model are very expressive and powerful but when we wanted to visualize/present/communicate such knowledge to a user, only experts understood it (Laclavik, 2005). The second problem was the lack of a query engine for the FIPA-SL language. Such challenges can be overcome by building an agent model based on Semantic Web standards and Technologies e.g. AgentOWL[37] extension of JADE using Jena semantic library[38]. Other solution based on Sesame[39], supporting RDF/OWL agent communication is presented in (Obitko,

[35] http://protege.stanford.edu/

[36] http://hcs.science.uva.nl/usr/aart/beangenerator/index25.html

[37] http://agentowl.sf.net/

[38] http://jena.sf.net/

[39] http://openrdf.org/

2004). This work on the other hand does not present any generic internal model, it only offers a theory how RDF/OWL can be used in Agents for modeling agent memory and communication.

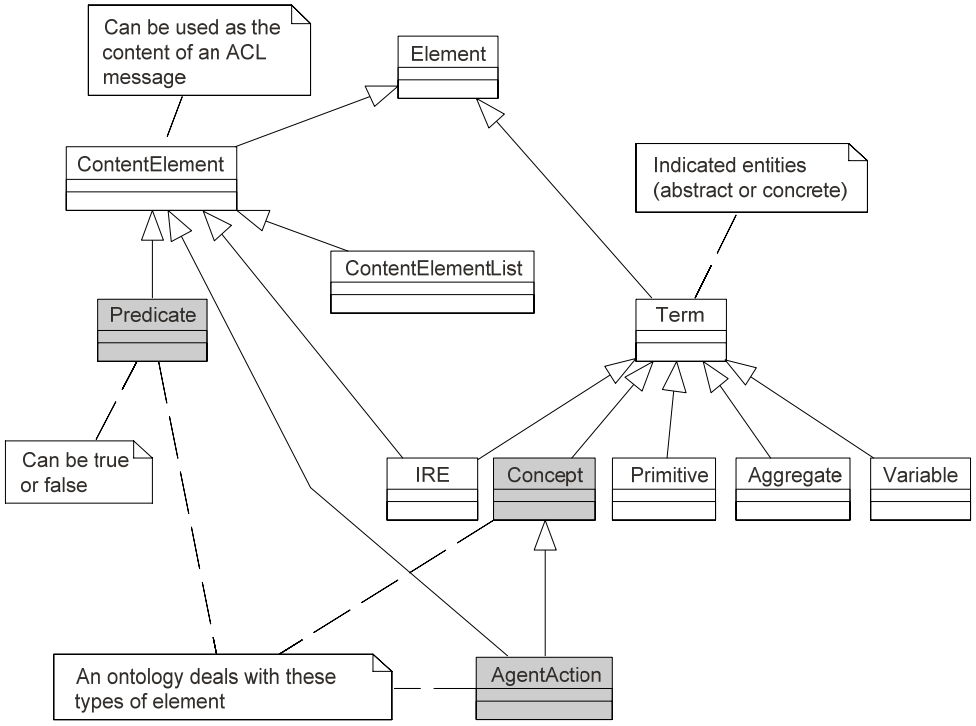


Figure 7-3. Content Reference Model.

7.2 Multi Agent Platforms

Software agents need to act in a software environment. Such environment is called an agency and its implementation is called the Agent Platform.

Many agent Platforms have been developed by the agent community. We evaluated about 50 of them (Laclavik, 2001) based on written information and also about 5 by developing agents on such platforms. (Nguyen, 2002)

First well known agent platforms are Telescript, dMARS or Aglets[40].

The FIPA was founded in 1996 and in 1998 and 2000 defined important standards. FIPA integrated the effort undertaken in this field. Since then MAS development has been standardized and previous MAS like Aglets or dMARS did not adopt new standards. There are only about 7 platforms which follow FIPA standards. Zeus, Grasshopper2 and JADE are mostly used and JADE is today probably only evolving FIPA standardized platform. JADE is currently widely used. It is used also in many European IST projects. Motorola is currently also participating in cooperation with the JADE development team.

[40] http://aglets.sourceforge.net/

JADE platform

JADE [41] (Java Agent DEvelopment Framework) is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middle-ware that claims to comply with the FIPA specifications and through a set of tools that supports the debugging and deployment phase.

The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent; agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. The full FIPA communication model has been implemented and its components have been clearly distinct and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and, finally, transport protocols. The transport mechanism, in particular, is like a chameleon because it adapts to each situation, by transparently choosing the best available protocol. Java RMI, event-notification, and IIOP are currently used, but more protocols can be easily added and integration of HTTP has been already achieved. Most of the interaction protocols defined by FIPA are already available and can be instantiated after defining the application-dependent behavior of each state of the protocol. SL and agent management ontology have been implemented already, as well as the support for user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework.

JADE is being used by a number of companies and academic groups, both members and non-members of FIPA, such as BT, CNET, NHK, Imperial College, IRST, KPN, University of Helsinky, INRIA, ATOS and many others. JADE is also most used Agent platform in Agent related scientific projects. It is available under Open Source License.

JADE include many plug-ins and extensions concerning different aspects of MAS. One of such extensions is also AgentOWL library which allows agents to use semantic model based on OWL [42] in communication and interaction as well as manipulation using Jena API and connection of agents to commercial standards such as XML or XML-RPC.

7.3 Communication and Ontology

The power of agent systems depends on inter-agent communication. Powerful agents need to be able to communicate with users, with customers, with system resources, and with each other if they want to cooperate, collaborate or negotiate. Common agent languages hold the promise of diverse agents communicating to provide more complex functions across the networked world. Indeed, as agents grow more powerful, their need for communication increases. The two agent communication languages with the broadest uptake are KQML[43] and FIPA ACL. KQML was developed in the early 1990s as part of the US government's ARPA Knowledge Sharing Effort, and is a language

[41] http://jade.cselt.it/

[42] http://www.w3.org/TR/owl-features/

[43] http://www.cs.umbc.edu/kqml/

and protocol for exchanging information and knowledge, which has been used extensively.

The Foundation for Intelligent Physical Agents (FIPA) is a nonprofit organization aimed at producing standards for the interoperation of heterogeneous software agents. The unproductive "standards war" scenario that might have arisen at one point seems now to have been avoided, with the most active participants supporting the FIPA effort, which incorporates many aspects of KQML. Europe has been a prime mover in the FIPA standardization effort, which seeks to address interoperability concerns through a sustained program. This is one area in which the visibility of agent technology is strong, with some of the most active take-up efforts from early adopters as, for example, is illustrated by the Agentcities[44] initiative. Despite their merit, KQML and FIPA ACL only deal with agent-to-agent communication. If we understand an agent as something that can act on behalf of a human or an organization, human-computer interface issues are crucial for the acceptance of agent technology. Questions remain of how a task can be delegated from a user to an agent, how user preference structures can be transferred to agents, and how the state of task execution can be adequately monitored and controlled by the user (Luck, 2003).

Communication can be understood also as main sensors for software agents, since it is how agents can learn, share knowledge and interact with their environment. Currently MAS are using mainly standardized FIPA-ACL (FIPA ACL, 2002) communication, where different content language can be used. We believe that in a future agents will get closer with semantic web Technologies and RDF or OWL will be widely used in ACL communication.

We think that recent Service Oriented Technologies and standards like WSDL, XML-RPC, SOAP or P2P technologies came out of MAS research area. For example WSDL and SOAP can be understood as a subset of what FIPA ACL is capable, however for most of applications features of XML-RPC or for most complicated interaction SOAP and WSDL is sufficient. Similarly, UDDI registry and FIPA directory facilitator specification has many things in common. We think that agents can move forward only when they incorporate existing commercial communication technologies such as XML-RPC or SOAP, and thus they will be able to communicate within the user and other existing software systems.

ACL Message Structure

A message written in FIPA Agent Communication Language (ACL) is a structured message consisting of the following elements: performative, sender, receiver, reply-to, content, language, encoding, ontology, protocol, conversation-id, reply-with, in-reply-to, reply-by. For more details see (FIPA ACL, 2002). The most important elements are performative, content, language and ontology. The remaining elements are simple elements, which control agent communication.

Performative consists of Communicative Acts which are listed in a table 7-1.

[44] http://www.agentcities.org/EUNET/http://www.agentcities.org/EUNET/

Table 7-1. Types of Communicative Acts in ACL.

Performative	Description
INFORM	The sender informs the receiver that a given proposition is true.
INFORM-REF	A macro action for sender to inform the receiver the object which corresponds to a descriptor, for example, a name.
NOT-UNDERSTAND	The sender of the act (for example, i) informs the receiver (for example, j) that it has perceived that j performed some action, but that i did not understand what j just did. A particular common case is that i tells j that i did not understand the message that j has just sent to i.
QUERY-REF	The action of asking another agent for the object referred to by a referential expression.
REFUSE	The action of refusing to perform a given action, and explaining the reason for the refusal.
REQUEST	The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act.
FAILURE	The action of telling another agent that an action was attempted but the attempt failed.
REQUEST-WHEN	The sender wants the receiver to perform some action when some given proposition becomes true.
REQUEST-WHENEVER	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.
SUBSCRIBE	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.
AGREE	The action of agreeing to perform some action, possibly in the future.

Content Languages

When agents are communicating, they exchange some content – "text" of the message. The content can be written in different languages.

- Content language FIPA-KIF: KIF – Knowledge Interchange Format [45] is a language designed for use in the interchange of knowledge among disparate computer systems (created by different programmers, at different times,

[45] http://logic.stanford.edu/kif/dpans.html

in different languages, and so on). FIPA-KIF is FIPA's KIF specification for agent communication (FIPA KIF, 2003).

- Content language FIPA-RDF: RDF – Resource Description Framework is a language used in FIPA-RDF specification. The RDF model proposes the eXtensible Markup Language (XML) as an encoding syntax, but does not prevent anyone from using alternative encoding schemes. All FIPA-RDF message contents will therefore use XML encoding, although, in principle, other encoding schemes could be used (FIPA RDF, 2001).
- Content language FIPA-SL: FIPA Semantic Language was proposed by FIPA as very similar to KIF but more precise and well defined. FIPA-SL content language. FIPA-SL (FIPA SL, 2002) is supported by agent platform JADE (Caire, 2002).
- RDF/OWL can be used also in ACL communication. This can be supported in JADE by using AgentOWL extension.

Ontology in MAS

Ontology defines the meaning of the terms in used content language and the relation among these terms. The model of agent communication in FIPA is based on the assumption that two agents, who wish to converse, share a common ontology for the domain of discourse. It ensures that the agents ascribe the same meaning to the symbols used in the message. Using ontology not only allows communication between agents but also gives the possibility for agents to reason about the concept.

Ontologies are dependent on used content language. In MAS ontologies are usually simple. The best known implementation of ontology is in JADE agent system. Ontology classes are represented by real Java classes with properties. Instances of classes are individuals – information which can be stored or communicated. However UML or object oriented ontology is not sufficient because, multiple inheritance, inverse properties and other features present in RDF or OWL can not be used. ACL by definition can use any ontology, thus when supported by software and libraries different ontology types can be used e.g. OWL.

7.4 Standardization in MAS

In the area of using software agents many MAS were developed but lacking standards. Two main standardization groups were formed:

- MASIF – Mobile Agents – first standardization focused mainly on mobile agents – This originated from the CORBA specification
- FIPA – Intelligent agents – originally focused on agent management, communication, knowledge, currently support also nomadic support (migration)

The Foundation for Intelligent Physical Agents (FIPA) is a nonprofit organization formed in 1996 to produce software standards for heterogeneous and interacting agents and agent-based systems. In the production of these standards, FIPA requires input and collaboration from its membership and from the agent's field in general to build specifications that can be used to achieve interoperability between agent-based systems developed by different companies and organizations.

FIPA specifications are divided into following areas:

- Application
- Abstract Architecture
- Agent Communication
- Interaction Protocols
- Communicative Acts
- Content Languages
- Agent Management
- Agent Message Transport
- ACL Representations
- Envelope Representations
- Transport Protocols

7.5 Agent Modeling

Agent based UML (AUML)

Multi-agent systems (MAS) are often characterized as extensions of object-oriented systems. This overly simplified view has often troubled system designers as they try to capture the unique features of MAS systems using OO tools. In response, an agent-based unified modeling language (AUML)[46] is being developed. The FIPA Modeling TC's goal, which try to develop AUML, wants to be domain independent. It tries to examine those areas where it has expertise: service-oriented architecture (SOA), business process management (BPM), simulation, real-time, AOSE, robotics, information systems and others. Instead of reliance on the OMG's UML, AUML intend to reuse of UML wherever it makes sense. The general philosophy, then, is: When it makes sense to reuse portions of UML, then do it; when it doesn't make sense to use UML, use something else or create something new.

In many cases existing UML tools can be used to define, e.g. agent interaction using UML sequence diagram.

MAScommonKADS

MAS-CommonKADS proposes an agent-oriented methodology. This methodology extends the knowledge engineering methodology CommonKADS (Schreiber, 2000) with techniques from object oriented and protocol engineering methodologies. The methodology consists of the development of seven models (Iglesias, 1998):

- Agent Model, that describes the characteristics of each agent;
- Task Model, that describes the tasks that the agents carry out;
- Expertise Model, that describes the knowledge needed by the agents to achieve their goals;

[46] http://www.auml.org/

- Organization Model, that describes the structural relationships between agents (software agents and/or human agents);
- Coordination Model, that describes the dynamic relationships between software agents;
- Communication Model, that describes the dynamic relationships between human agents and their respective personal assistant software agents;
- Design Model that refines the previous models and determines the most suitable agent architecture for each agent, and the requirements of the agent network.

7.6 Summary and Conclusion

With Agents it is like with any other technology. Agents will be successful after such solutions have reached a critical mass (Willmott, 2003), like a cell phone is useless if nobody has one.

According to Agent Technology Roadmap (Luck, 2003), which is the result of AgentLink II[47] community, there is a number of broad technological challenges for research and development over the next decade in the agent technology.

- Increase quality of agent software to industrial standard: One of the most fundamental obstacles to large-scale take-up of agent technology is the lack of mature software development methodologies for agent-based systems. Clearly, basic principles of software and knowledge engineering need to be applied to the development and deployment of multi-agent systems, but they also need to be augmented to suit the differing demands of this new paradigm.
- Provide effective agreed standards to allow open systems development. In addition to standard languages and interaction protocols, open agent societies will require the ability to collectively evolve languages and protocols specific to the application domain and to the agents involved. Some work has commenced on defining the minimum requirements for a group of agents with no prior experience of each other to evolve a sophisticated communications language, but this work is still in its infancy. Research in this area will draw on linguistics, social anthropology, biology, the philosophy of language and information theory.
- Provide semantic infrastructure for open agent communities: At present, information agents exist in academic and commercial laboratories, but are not widely available in real world applications. The move out of the laboratory is likely to happen in the next ten years, but requires: a greater understanding of how agents, databases and information systems interact; investigation of the real-world implications of information agents (for example, including the economic effects of shop-bots); and development of benchmarks for system performance and efficiency. In order to support this, further needs include: new web standards that enable structural and semantic description of information; and services that make use of these semantic representations for information access at a higher level. The creation of common ontologies, thesauri or knowledge bases play a central role here, and merits further work on the formal descriptions of

[47] http://www.agentlink.org/

information and, potentially, a reference architecture to support the higher level services mentioned above.

- Develop reasoning capabilities for agents in open environments: At present, organizational approaches do not adequately handle the issues inherent in open multi-agent systems, namely heterogeneity of agents, trust and accountability, failure handling and recovery, and societal change. The next challenge for agent-based computing is to develop appropriate representations of analogous computational concepts to the norms, legislation, authorities, enforcement, etc., that can underpin the development and deployment of dynamic electronic institutions. Similarly, virtual organizations involve dynamic coalitions of small groups that can provide more services and make more profits than an individual group. Moreover, such coalitions can disband when they are no longer effective. At present, coalition formation for virtual organizations is limited, with such organizations largely static. The automation of coalition formation will save both time and labor, and maybe more effective at finding better coalitions than humans in complex settings. Related issues include negotiation and argumentation, and domain-specific models of reasoning, both of which may be used to form such groups of agents in open environments.
- Develop agent ability to understand user requirements: At the architecture level, future avenues for learning research include developing distributed models of profile management, as well as more general distributed agent learning techniques rather than just single agent learning in multi-agent domains. Developing approaches to personalization that can operate in a standards-based, pervasive computing environment presents many interesting research challenges including, how to integrate machine learning techniques (for profile adaptation) with structured XML-based profile representations .Another are a deserving of greater activity is that of distributed profile management a task for which the agent based paradigm should be well suited. The impact of the emerging Semantic Web on approaches for wrapper induction and text-mining also requires careful study
- Develop agent ability to adapt to changes in environment: Develop agent ability to adapt to changes in environment. Even though learning technology is clearly crucial for open and scalable multi-agent systems, it is still in early development. While there has progress in many areas, such as evolutionary approaches and reinforcement learning, these have still not made the transition to real-world applications. Reasons for this can be found in problems of scalability and in user trust in self-adapting software. In the longer term, learning techniques are likely to become a central part of agent systems, while the shorter term offers application opportunities in areas such as interactive entertainment, which are not safety critical
- Ensure user confidence and trust in agents: Ensure user confidence and trust in agents. Collaboration of any kind, especially in situations in which computers act on behalf of users or organizations, will only succeed if there is trust. For this trust to be given requires a variety of factors to be in place. First, a user must have confidence that an agent or group of agents which represents them within an open system will act effectively on their behalf it must be at least as effective as the user would be in similar circumstances. Second, agents must be secure and tamper-proof, and must not reveal information in appropriately (e.g., bank account details). There is much work on system security, cryptography and

privacy which can be exploited and adapted for use in agent technology. Finally, if a user is to trust the outcome of an open agent system, they must have confidence that agents representing other parties or organizations will behave within certain constraints. Mechanisms to do this include: reputation mechanisms; the use of norms (social rules) by all members of an open system; self-enforcing protocols, which ensure that it is not in the interests of any party to break them; and electronic contracts.

As already mentioned, this summary comes from the Agent Technology Roadmap developed by many Agent researchers in Europe and world-wide. Multi-Agent Systems or Agent Oriented Programming is a very powerful method in distributed heterogeneous information systems certain degree of intelligence and autonomy is needed. In this chapter we list many research and development challenges in MAS as well as described briefly state of the art, standards such as FIPA, and MAS toolkits such as JADE, which can be exploited while using Agent Technology.

References

BROOKS, R. (1991). *Intelligence without reason.*

BROOKS, R. (1991). *Intelligence without representation.*

CAIRE, G. (2002) *JADE Tutorial Application-defined Content Languages and Ontology*, http://jade.cselt.it/

SCHREIBER, A. ET AL. (2000). *Knowledge Enginering and Management: the CommonKADS methodology*, ISBN:0-262-19300-0

FIPA ACL (2002). *FIPA ACL: FIPA Specification ACL Message Structure*, http://www.fipa.org/specs/fipa00061/

FIPA KIF (2003). *FIPA KIF Content Language Specification*, http://www.fipa.org/specs/fipa00010/

FIPA (2001), *Ontology Service Specification,* http://www.fipa.org/specs/fipa00086/

FIPA RDF (2001). *FIPA RDF Content Language Specification*, http://www.fipa.org/specs/fipa00011/

FIPA SL (2002). *FIPA SL Content Language Specification*, http://www.fipa.org/specs/fipa00008/

LACLAVIK, M. (2001). *Negotiation and Communication in Agent Systems* - Project of the PhD. Thesis

NGUYEN, G. – DANG, T. – HLUCHY, L. – LACLAVIK, M. – BALOGH, Z. – BUDINSKA, I. (2002). *Agent Platform Evaluation and Comparison;* II-SAS, Pellucid EU 5FP IST-2001-34519 RTD, Technical report Jun 2002, Bratislava, Slovakia

LACLAVIK, M. – BALOGH, Z. – NGUYEN, G. – GATIAL, E. – HLUCHY, L. (2005). Methods for Presenting Ontological Knowledge to the Users; In: L.Popelinsky, M.Kratky (Eds.): *Znalosti 2005, Proceedings*, VSB-Technicka universita Ostrava, Fakulta elektrotechniky a informatiky, 2005, pp.61-64. ISBN 80-248-0755-6.

LUCK, M. – MCBURNEY, – P. PREIST, C. (2003). *Agent Technology: Enabling Next Generation Computing, A Roadmap for Agent Based Computing*, 2003

IGLESIAS, C. – GARIJO, M. - GONZALEZ, J. - VELASCO, J. (1998). *Analysis and Design of Multiagent Systems using MAS-CommonKADS*

OBITKO, M. – MARÍK, V. (2004). OWL Ontology Agent based on FIPA proposal, In *Proc. of Znalosti 2004*, Brno, Czech Republic

WILLMOTT, S. – THOMPSON, S. – BONNEFOY, D. (2003). *Agent Cities: Towards Dynamic Value Creation in On-line Open Environments*, IST Conference Milan, Italy

WOOLDRIDGE, M. (2002). *Introduction to MultiAgent Systems*, ISBN:047149691X

EDÍCIA VÝSKUMNÝCH TEXTOV
INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Mária Bieliková, Pavol Návrat a kol.

ŠTÚDIE VYBRANÝCH TÉM SOFTVÉROVÉHO INŽINIERSTVA (3)

Pokročilé metódy navrhovania programových systémov
Pokročilé metódy získavania, vyhľadávania, reprezentácie
a prezentácie informácií

1. vydanie
Tlač Vydavateľstvo STU v Bratislave

228 strán

2007

ISBN 978-80-227-2701-3