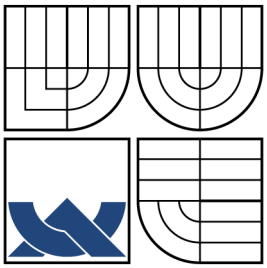
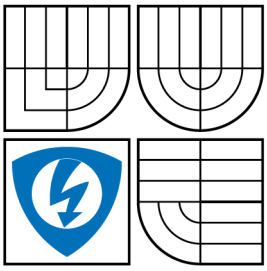


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND
COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

BEZPEČNÝ PŘENOS DAT MEZI MULTIPLATFORMNÍMI DATABÁZOVÝMI SERVERY

SAFE TRANSFER OF DATA BETWEEN MULTIPLATFORM DATABASE SERVERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JIŘÍ NAVRÁTIL
AUTHOR

VEDOUCÍ PRÁCE — ING. IVO LATTENBERG, PH.D.
SUPERVISOR

BRNO 2008

ZDE VLOŽIT LIST ZADÁNÍ

Z důvodu správného číslování stránek

ABSTRAKT

Práce pojednává o možnostech synchronizace serverů, zamýšlí se nad možností komunikace multiplatformních klientských aplikací s databázovým serverem. Dále navrhuje řešení pomocí aplikačního serveru a demonstruje jeho realizaci v běžném vývojovém prostředí.

KLÍČOVÁ SLOVA

SQL, MySQL, Firebird, Aplikační server, TCP/IP, Delphi, Java

ABSTRACT

This work deals with options of servers synchronization, possibilities of communication among client applications and database server. It also sets up solution using the application server and demonstrates its realization in common development environment.

KEYWORDS

SQL, MySQL, Firebird, Aplikační server, TCP/IP, Delphi, Java

Navrátil J. *Bezpečný přenos dat mezi multiplatformními databázovými servery*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2008. Počet stran 79., Počet stran příloh 0. příloh 0. Vedoucí práce Ing. Ivo Lattenberg, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Bezpečný přenos dat mezi multiplatformními databázovými servery“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....

(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

ÚVOD

Tato práce se zabývá průzkumem možností multiplatformní komunikace.Popisuje jednotlivá vhodná prostředí pro řešení problému , shromažďuje vhodné příkazy či syntaxe k praktickému využití. Dále navrhuje postupy a metodiky pro řešení přenosu dat.

Stěžejním prvkem této práce je vývoj aplikačního serveru, který umožňuje sjednotit klienty, jenž jsou provozovány na různých platformách. Práce se zabývá podrobně tím, jak vystavět veškerou komunikaci tohoto aplikačního serveru. Podrobně zkoumá možnosti použitých komponent. Zaměřuje se na reálné použití této technologie a testuje její možnosti.

1 DATABÁZE A JAZYK SQL

Tato kapitola je s malým pozměněním přejata ze zdroje [1] a slouží jako základní uvedení do problematiky databází a SQL.

Pojem databáze dnes není zcela jistě nikomu cizí. Lidé mají potřebu evidovat a shromažďovat informace už odpradávna. Celá dnešní moderní společnost je postavena na databázových systémech, od evidence občanů, přes zdravotnictví, hospodářství, školství, až po letectví, výzkum, nebo síť mobilních telefonů.

1.1 Vývoj databází

Termín databáze je dnes natolik známý, že snad i úplný laik ví, co si má pod tímto pojmem představit. Nemusí vědět, jak se s databází pracuje, jak je reprezentována v paměti, nebo jak se v ní vyhledává. Ví ale zcela jistě, že slouží k ukládání dat. Asi nejlépe představitelný tvar databáze je podoba jakési tabulky, ve které jsou data uložena. V této tabulce můžeme data pohodlně vyhledávat, třídit, nebo filtrovat.

Počátky databází sahají až do konce devatenáctého století. Již v roce 1890 vytvořil Herman Hollerith, na žádost vlády USA, první automat, který dokázal pracovat s děrnými štítky. V roce 1911 spolu s dalšími společníky založil firmu IBM. Ta se výpočetní technikou zabývá až dodnes. Dalším významným produktem IBM byl stroj UNIVAC I. Jednalo se o první digitální počítač pro komerční využití a byl vytvořen v roce 1935. IBM ho vyrobila také na základě žádosti vlády USA a sice kvůli zavedení Social Security Act. Jedná se o číslo přidělené každému občanovi USA (obdoba našeho rodného čísla). Tento počítač tedy musel zvládnout vedení údajů o více než 26 milionech obyvatel USA. Byl součástí projektu Electronic Discrete Variable Automatic Computer (EDVAC) z University of Pennsylvania. Mozkem těchto počítačů byl stále hlavně děrný štítek.

V 60. letech minulého století vzniká současný pojem databáze, entita, atribut entity a vazba mezi entitami. To jsou základní pojmy, o kterých se na úvod zmíním.

Databázi si lze představit jako soubor dat, který slouží pro popis reálného světa (např. evidence školní knihovny, sklad chemikálií, evidence studentů). Entita je prvek reálného světa (např. člověk, stroj, vyučovaný předmět, město), který je popsán svými charakteristikami (vlastnostmi). Ty se většinou považují za atribut (např. jméno, příjmení, stav, plat, hmotnost).

Dalším důležitým pojmem je vazba mezi entitami. Jednotlivé entity odpovídající prvkům z reálného světa, mají mezi sebou určitý vztah. Např. každý člověk má právě jedny osobní údaje vedené na magistrátě, na oddělení občanských průkazů. To hovoříme o vazbě typu 1:1. Dalším typem je vazba 1:N, jíž bude odpovídat např. skutečnost, že jeden člověk může vlastnit více kreditních karet (ale jedna kreditní

karta může být vlastněna pouze jedním člověkem). Posledním typem vazby je M:N. Zde není žádné omezení, příkladem by mohla být situace, že student na vysoké škole si může zapsat několik různých předmětů (ale jeden předmět může být zároveň zapsán více studenty).

V této době vzniká ještě jeden pojem, a to databázový model. Ten byl zaveden zejména matematiky, jako prostředek pro popis databáze. Zpočátku se používaly modely dvojího typu: hierarchický (založen na modelování hierarchie mezi entitami se vztahy podřízenosti a nadřízenosti) a dále síťový (vychází z teorie grafů, uzly v grafu odpovídají entitám a orientované hrany definují vztahy mezi entitami). V 70. letech se uvedené databázové modely ukázaly být nedostatečné (objevily se problémy s realizací a implementací vazby M:N), a proto vznikl relační model, který se stal standardem a používá se dodnes.

1.2 Relační databáze

Podívejme se nyní blíže na relační model[5]. Základním pojmem je relace. Relaci, aniž bych zaváděl jakékoliv matematické definice, si lze představit jako tabulku, která se skládá ze sloupců a řádků. Sloupce odpovídají jednotlivým vlastnostem (atributům) entity. Údaje v jednom řádku tabulky zobrazují aktuální stav světa. Budu mít např. tabulku Zaměstnanec, která bude popisovat entitu pracovníka ve firmě. Sloupce tabulky budou: ČÍSLO,JMÉNO,PŘÍJMENÍ,DAT_NAR,PLAT a SMLOUVA_OD uvádí datum, od kterého je pracovník v naší firmě zaměstnán. Představme si, že reálnému světu odpovídá následující naplnění tabulky:

CISLO	JMENO	PRIJMENI	DAT_NAR	PLAT	SMLOUVA_OD
1	Jiří	Navrátil	30.9.1984	12000	1.1.2007
2	Petr	Novák	20.2.1984	10500	1.1.2006
3	Jan	Olsina	1.5.1983	14000	1.3.2007

Tabulka je základním stavebním kamenem pro budování celé databáze. Relace tedy odpovídá celé tabulce a prvku relace odpovídá jeden konkrétní řádek. Jeden řádek bývá často nazýván databázovým záznamem. Soubor tabulek (relací) pak tvoří celou databázi (relační schéma).

U tabulek ještě chvíli zůstaňme. Jedna tabulka nám popisuje nějakou entitu. Za sloupce v tabulce zvolíme ty atributy, které o dané entitě chceme evidovat a které nás zajímají. Nyní je nejvyšší čas říci si něco o tvorbě relačních tabulek. Tvorba "dobře navržených" tabulek je klíčový a důležitý úkol zejména z hlediska dlouhodobého. (Pokud bychom v nějakém databázovém systému, který již nějakou dobu běží v ostrém provozu, nalezli nějakou chybu a zjistili bychom, že spočívá ve špatně na-

vržené databázi, mělo by to katastrofální důsledky, a to nejen po finanční stránce.) Než budu pokračovat dál, vysvětlím další základní pojmy z oblasti relační databáze.

1.3 Základní pojmy

Hodnotou většinou rozumíme uživatelská data. Každý sloupec v tabulce má svůj datový typ (např. celé číslo, řetězec, datum, logická hodnota, apod).

Pro práci s databázovými tabulkami je užitečné (ne-li přímo nutné) mít alespoň jednu položku (sloupec), jejíž hodnota nám bude jednoznačně identifikovat záznam v tabulce. Pokud taková položka nebude příliš velká (např. v počtu bajtů), zvolíme ji za tzv. primární klíč. V našem příkladu tabulky ZAMĚSTNANEC lze za primární klíč zvolit položku ČÍSLO. Primární klíč má tu vlastnost, že jeho hodnota je jedinečná, tj. pro žádné dva řádky v tabulce nemůže nastat situace, že by hodnota primárního klíče byla totožná. Databázové systémy většinou umožňují definovat jako primární klíč n-tici položek, např. dvojici nebo trojici položek. V takovém případě se mohou některé položky v klíčích opakovat, ale nesmí být shodné všechny položky dvou primárních klíčů najednou.

Neméně důležitá je i funkční závislost. Funkční závislosti si lze představit jako tvrzení o reálném světě. Například plat zaměstnance závisí na tom, jakou vykonává funkci, tj. plat závisí na funkci. Druhým příkladem by mohla být výše jízdného, která závisí na délce vlakové trasy. Takových příkladů z běžného života bychom našli mnoho.

1.4 Normální formy

Pojem normálních forem se používá ve spojitosti s dobře navrženými tabulkami. Správně vytvořené tabulky splňují 4 základní normální formy[5].

1.4.1 1. normální forma (1NF)

První, nejjednodušší, normální forma (značíme 1NF) říká, že všechny atributy jsou atomické, tj. dále již nedělitelné (jinými slovy, hodnotou nesmí být relace). Mějme např. tabulku ADRESA, která bude mít sloupce JMENO, PRIJMENI a BYDLISTE. Naplnění tabulky nechť odpovídá reálnému světu:

JMENO	PRIJMENI	Bydliste
Jiří	Navrátil	Albertova 1020 Kromeriz
Petr	Novák	Ceska 1144 Brno
Jan	Olsina	Ostravská 84 Praha

Pokud bychom v této tabulce chtěli vypsat všechny pracovníky, jejichž PSČ je rovno určité hodnotě, dostali bychom se do potíží, neboť bychom to nemohli zjistit přímo a jednoduše. A to proto, že atribut BYDLIŠTĚ není atomický, skládá se z několika částí: ULICE, CISLO, MESTO a PSC. Správný návrh tabulky, který bude respektovat 1NF bude vypadat následovně:

JMENO	PRIJMENI	ULICE	CISLO	MESTO	PSC
Jiří	Navrátil	Albertova	1020	Kromeriz	76701
Petr	Novák	Ceská	1144	Brno	61400
Jan	Olsina	Ostravská	84	Praha	16000

Obecně bychom se měli snažit, aby obsahem jedné databázové položky byla právě jedna hodnota (určitého databázového typu).

1.4.2 2. normální forma (2NF)

Tabulka splňuje 2NF, právě když splňuje 1NF a navíc každý atribut, který není primárním klíčem je na primárním klíči úplně závislý. To znamená, že se nesmí v řádku tabulky objevit položka, která by byla závislá jen na části primárního klíče. Z definice vyplývá, že problém 2NF se týká jenom tabulek, kde volíme za primární klíč více položek než jednu. Jinými slovy, pokud má tabulka jako primární klíč jenom jeden sloupec, pak 2NF je splněna triviálně. Nechť máme tabulku PRACOVNÍK, která bude vypadat následovně (atribut ČÍSLO_PRAC značí číslo pracoviště, kde daný pracovník pracuje, atribut NÁZEV_PRAC uvádí jméno daného pracoviště):

Cislo	JMENO	PRIJMENI	CISLO_PRAC	NAZEV_PRAC
1	Jiří	Navrátil	2	lisovna
2	Petr	Novák	3	vrátnice
3	Jan	Olsina	4	výpočetní centrum

Jaký primární klíč zvolíme v této tabulce? Pokud zvolíme pouze CISLO, je to špatně, neboť zcela určitě název pracoviště, kde zaměstnanec pracuje, není závislý na číslu pracovníka. Takže za primární klíč musíme vzít dvojici (ČÍSLO,ČISLO_PRAC). Tím nám ovšem vznikl nový problém. Položky JMENO, PRIJMENI a NAZEV_PRAC nejsou úplně závislé na dvojici zvoleného primární klíče. Ať tedy děláme, co děláme, nejsme schopni vybrat takový primární klíč, aby tabulka splňovala 2NF. Jak z tohoto problému ven? Obecně převedení do tabulky, která již bude splňovat 2NF, znamená rozpad na dvě a více tabulek, kde každá už bude splňovat 2NF. Takovému "rozpadu" na více tabulek se odborně říká dekompozice relačního schématu. Správně navržené tabulky splňující 2NF budou vypadat následovně (tabulka PRACOVNIK

a PRACOVISTE):

Cislo	JMENO	PRIJMENI	CISLO_PRAC
1	Jiří	Navrátil	2
2	Petr	Novák	3
3	Jan	Olsina	4

CISLO_PRAC	NAZEV
2	lisovna
3	vrátnice
4	výpočetní centrum

Dále si všimněte, že pokud tabulka nesplňuje 2NF, dochází často k redundanci. Konkrétně v původní tabulce informace, že pracoviště číslo 10 se jmenuje "studovna", byla obsažena celkem dvakrát. Redundance je jev, který obvykle nesplnění 2NF doprovází. O tom, že redundance je nežadoucí, netřeba pochybovat.

1.4.3 normální forma (3NF)

Relační tabulky splňují třetí normální formu (3NF), jestliže splňují 2NF a žádný atribut, který není primárním klíčem, není tranzitivně závislý na žádném klíči. Nejlépe to opět vysvětlí následující příklad. Mějme tabulku PLATY, která bude vypadat takto:

Cislo	JMENO	PRIJMENI	FUNKCE	PLAT
1	Jiří	Navrátil	technik lisu	12000
2	Petr	Novák	vrátný	10500
3	Jan	Olsina	matematik	14000

Pomineme zatím fakt, že tato tabulka nesplňuje ani 2NF, což je základní předpoklad pro 3NF. Chci zde jen vysvětlit pojem tranzitivní závislost. Nebudeme přemýšlet, co je primární klíč, na první pohled vidíme, že konkrétně atributy JMÉNO, PŘÍJMENÍ a FUNKCE závisí na atributu ČÍSLO (ten by nejspíš byl primárním klíčem). Dále můžeme vidět, že atribut PLAT zřejmě je funkčně závislý na atributu FUNKCE a pokud vememe v úvahu, že ČÍSLO - FUNKCE a FUNKCE - PLAT, dostaneme díky jevu nazývanému tranzitivita, že ČÍSLO - PLAT. Postup, jak dostat tabulky do 3NF, je podobný jako v případě 2NF, tj. opět provedeme dekompozici (tabulka FUNKCE a PLATY):

Cislo	JMENO	PRIJMENI	FUNKCE
1	Jiří	Navrátil	technik lisu
2	Petr	Novák	vrátný
3	Jan	Olsina	matematik

FUNKCE	PLAT
technik lisu	12000
vrátný	10500
matematik	14000

1.4.4 Boyce-Coddova normální forma

Poslední prakticky užívanou formou je tzv. Boyce-Coddova normální forma (BCNF). Tabulka splňuje BCNF, právě když pro dvě množiny atributů A a B platí: A-¿B a současně B není podmnožinou A, pak množina A obsahuje primární klíč tabulky. Tato forma zjednodušuje práci s tabulkami, ve většině případů, pokud dobře postupujeme při tvorbě tabulek, aby splňovaly postupně 1NF, 2NF a 3NF, forma BCNF je splněna.

1.5 Jazyk SQL

Po delším úvodu, který ale považuji za velmi důležitý pro pochopení základních principů a pojmů ze světa databází a který vám poskytuje návod, jak správně navrhnout databázové tabulky, se konečně dostávám k jazyku SQL.

Historie jazyka SQL spadá do 70. a 80. let. První standard byl přijat v roce 1986 (označován jako SQL86)[5]. Časem se však projevily některé nedostatky. Opravená verze je z roku 1992 a je označována jako SQL92. Ten je v oblasti relačních databází standardem dodnes. Zkratka SQL značí Structured Query Language. Jazyk v sobě zahrnuje nástroje pro tvorbu databází (tabulek) a dále nástroje na manipulaci s daty (vkládání dat, aktualizace, mazání a vyhledávání informací).

SQL patří mezi tzv. deklarativní programovací jazyky, což v praxi znamená, že kód jazyka SQL nepíšeme v žádném samostatném programu (jako by tomu bylo např. u jazyka C nebo Pascal), ale vkládáme jej do jiného programovacího jazyka, který je již procedurální. Se samotným jazykem SQL můžeme pracovat pouze v případě, že se terminálem připojíme na SQL server a na příkazový řádek bychom zadávali přímo příkazy jazyka SQL.

Jak už jsem se zmínil, SQL se skládá z několika částí[5]. Některé části jsou určeny pro administrátory a návrháře databázových systémů, jiné pak pro koncové uživatele a programátory. První částí jazyka SQL je jazyk DDL - Data Definition Language. Jedná se o jazyk pro vytváření databázových schémat a katalogů. Způsob

ukládání tabulek definuje jazyk SDL - Storage Definition Language. Třetí částí pro návrháře a správce je jazyk VDL - View Definition Language, určující vytváření pohledů (pohled si lze představit jako virtuální tabulku složenou z různých jiných tabulek). Poslední částí, kterou se budu převážně zabývat, je jazyk DML - Data Manipulation Language, který obsahuje základní příkazy INSERT, UPDATE, DELETE a nejpoužívanější příkaz SELECT. S jazykem DML pracují nejvíce koncoví uživatelé a programátoři databázových aplikací.

2 KOMUNIKACE SERVERŮ A SYNCHRONIZACE JEJICH DAT

Nyní jsme si popsali základní principy databází a umíme si představit jejich strukturu. Tato znalost pro nás bude dále důležitá při návrzích přenosů dat mezi těmito databázemi. Jelikož se budeme snažit o co nejvyšší univerzálnost , užijeme již zmíněný deklarativní jazyk SQL. Ten nám zajistí kompatibilitu mezi databázemi SQL a MySQL. Připouštím , že se zde dají najít drobné odlišnosti v závislosti na verzích a pod.Avšak pokud se nebudeme pouštět do nějakých nových specifických rozšíření , na nekompatibilitu patrně nenarazíme.

2.1 Základní SQL deklarace pro přenos dat

Nyní si uvedeme základní SQL deklarace, bez nichž se při realizaci komunikace neobejdeme.[5]

Pro přihlášení k databázi:
mysql -u uživatelské_jméno -pheslo jméno_vaší_databáze

Pro vložení řádku do tabulky:
INSERT INTO jméno_tabulky [(jména sloupců)] VALUES (seznam hodnot)

Editace vybraných položek dabulky:
UPDATE jméno_tabulky SET sloupec = hodnota, ... [WHERE podmínky]

Mazání řádků tabulky:
DELETE FROM jméno_tabulky [WHERE podmínky]

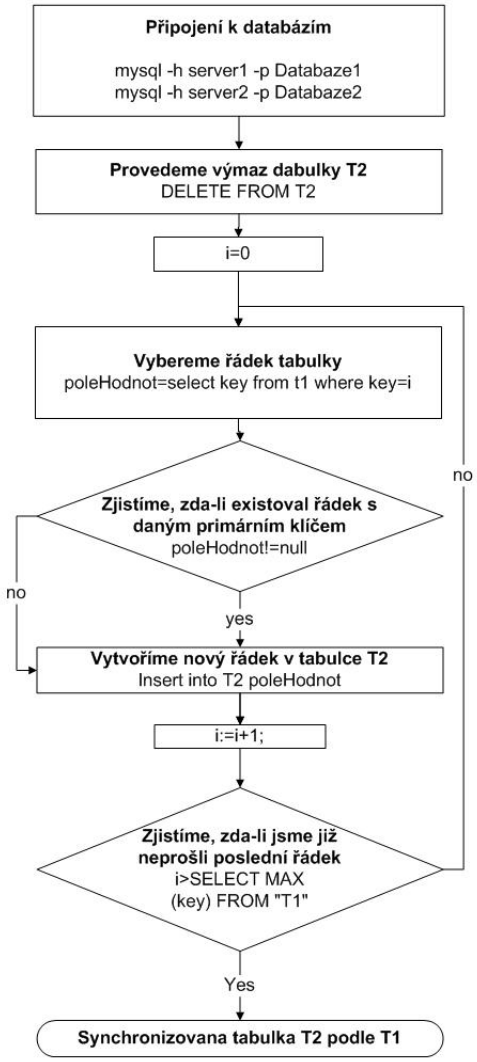
2.2 Synchronizace dat serverů

Nastínili jsme si potřebné deklarace, abychom mohli začít uskutečňovat přenosy dat. Jenže chceme-li bezpečně přenést tabulky z jednoho serveru na druhý, narazíme na další problémy. Především je otázkou, jestli bereme jednu tabulku za vůdčí a všechna data v druhé tabulce považujeme za stará, pak je situace dosti jednodušší, kdy si vystačíme s kopií řádků.V opačném případě bude třeba složitější logiky pro spojení obou tabulek. My si popíšeme oba případy.

2.2.1 Jednosměrná synchronizace

Toto je ten případ, kdy považujeme jednu tabulku za vůdčí a žádné změny druhé tabulky nás nezajímají. Tedy v podstatě stačí smazat obsah druhé tabulky a nakopírovat jí nové záznamy vůdčí tabulky. Pak tento postup zopakovat pro všechny vybrané tabulky, jež chceme synchronizovat.

Ukážeme si diagram nastiňující tento případ i s příkazy SQL. Použil jsem kopírování jednotlivých záznamů po řádcích. Připouštím že tento přístup může být v rozsáhlejších databázích pomalejší, ale zase nabízí libovolné zpracování jednotlivých řádků naší programovou logikou. Navíc při složitější synchronizaci tabulek je tento postup nevyhnutelný, neboť musíme každý řádek analyzovat zvlášť a tedy nemůžeme kopírovat celou tabulku najednou.



Obr. 2.1: Diagram jednosměrné synchronizace

2.2.2 Oboustranná synchronizace

Tento případ je výrazně složitější než předchozí. Rozhodně si zde nevystačíme s obyčejnou kopií řádků.Nyní stojíme před úkolem, analyzovat jednotlivé řádky databáze a rozhodnout o nich , zda-li jsou: Nové,Editované,Smazané,či beze změny. Po té co je správně vyhodnotíme, musíme je patřičně rozkopírovat do komplementární tabulky, se kterou synchronizujeme. Navíc tato provázanost je obousměrná.Každá tabulka může mít své nové jiné záznamy a své různě editované záznamy. V každém případě musíme navrhnout algoritmus, který je bude schopen za každých okolností zpracovat a tabulky poskládat.

Jenže hned při další úvaze zjistíme, že si patrně nevystačíme s údaji v tabulkách , které chceme synchronizovat(tedy pokud tabulky již nejsou připraveny pro synchronizaci).A které záznamy nám chybí ? Především musíme být schopni říci, kdy je záznam smazaný, editovaný a kdy je nový. Možných řešení je celá řada. Já osobně navrhuji zavést do tabulky další sloupec s názvem status, který bude nést informaci, zda-li je řádek nový,editovaný, ke smazání , beze změny.Status můžeme být např jednobytová číselná proměnná a bude nabývat hodnot 0-2(viz tabulka).Je třeba podotknout, že při tomhle systému je položka s tabulky fyzicky smazána až po synchronizaci.

Ovšem naskýtá se další otázka. Jak jednoznačně poznat dané komplementární záznamy(řádky)? Jedinou odpovědí je zavést další sloupec s názvem S_Key. Tento synchronizační klíč nám pak jasně identifikuje daný jeden řádek první tabulky s právě jedním řádkem druhé tabulky. Díky tomu bezpečně víme, že řádky se mají vzájemně srovnávat.

Budeme-li chtít ještě automaticky rozhodnout o tom, který záznam se má použít, v případě že oba komplementární řádky byly změněny, budeme muset zavést další sloupeček hovořící o datu. Je otázkou, zda-li tento stav považovat za kolizi, či ne. Zde bude záležet na systému, který navrhujeme, nebo který chceme synchronizovat. V závislosti na tom , buď automaticky rozhodneme o použití řádku s novějším datumem, nebo vzneseme dotaz na administrátora.

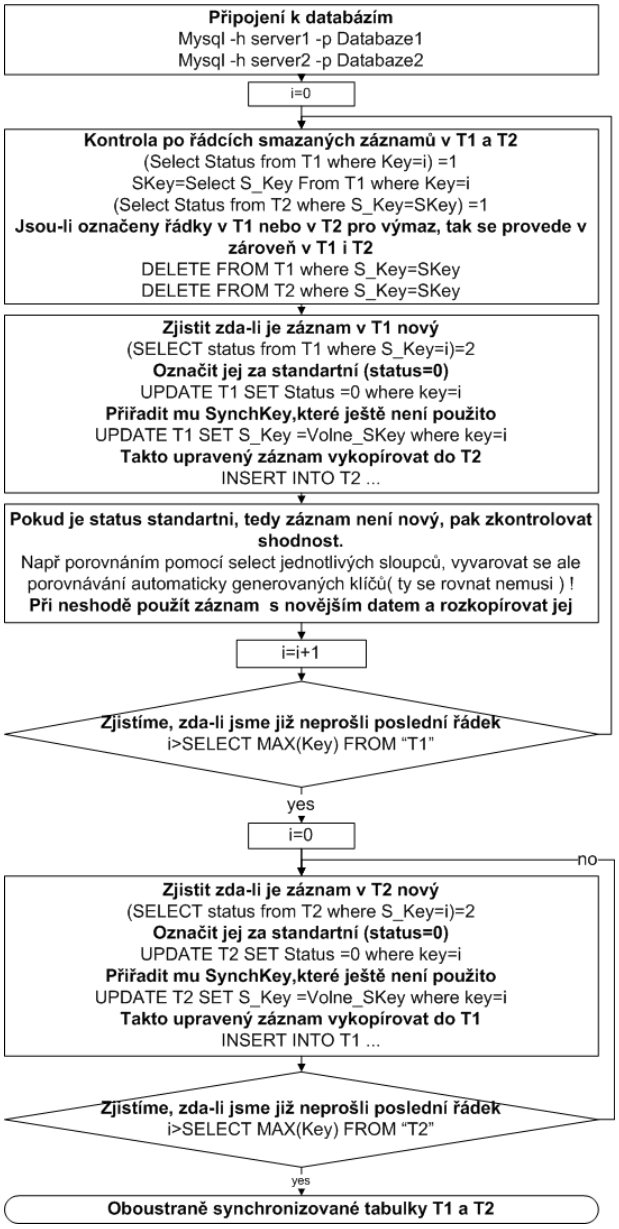
Takže rozšíření naší databáze může pak vypadat následovně:

S_KEY	STATUS	DATUM	PůvodníData....
1	0	1.1.2007	Data BezeZmeny
2	1	1.2.2007	Nový záznam
3	2	10.10.2007	Editovaný záznam

Nyní jsme připravili naši tabulku pro synchronizaci. Ještě nesmíme zapomenout na to , že budeme často přidávat S_Key novým záznamům. Takže je třeba vždy vě-

dět , jaký je aktuálně nejvyšší použitý S_Key a tím pádem můžeme při synchronizaci přidělovat novým záznamům tento klíč. Nikde jinde tento klíč nesmí být měněn ani přidělován.

Takže tohle byla příprava databáze pro oboustrannou synchronizaci a nyní si ukážeme logiku s sql příkazy opět ve vývojovém diagramu.



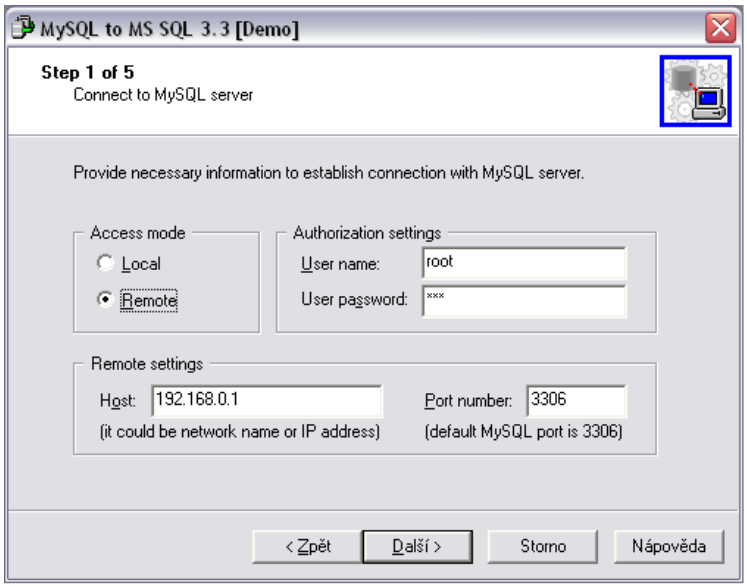
Obr. 2.2: Diagram obousměrné synchronizace

3 KOMERČNÍ NÁSTROJ MYSQLMIGRATION

Jedná se o placený nástroj pro komunikaci mezi různými typy databází. Tedy převody se neomezují jen na SQL a MySQL. Nicméně v další kapitole uvidíme, že se dají najít i jiná řešení, která tuhle výhodu nesou také a mají spoustu dalších.Ovšem dalších výhod tento nástroj nemá.

3.1 Připojení programu k serverům

Program se dokáže připojit na různé databázové servery. Ještě před spuštěním tedy vybereme vhodnou variantu aplikace , např MySQL TO MSSQL.Zde se nás postupně dotáže na přihlašovací informace k jednotlivým serverům.



Obr. 3.1: Konfigurace připojení

Následuje výběr možných převodů a výběr požadovaných databází k převodům.

3.2 Funkčnost a použitelnost

Program nám umožňuje vytvářet konverze. Je relativně flexibilní po stránce výběru dat, které chceme synchronizovat.Nicméně data přenáší pouze jednosměrně a v případě duplicity databází , či tabulek je přepisuje. Tedy neumožní nám oboustrannou synchronizaci serverů. Navíc je to nástroj jen pro konverze, takže jím nejsme schopni přenášet data mezi servery stejných typů. To jsou ovšem příliš velká omezení.

3.3 Závěr

Tento nástroj se podle mého soudu hodí pouze na jednosměrné a jednorázové konverze mezi databázemi. Pro další použití je nevyhovující. Přihlédneme-li k tomu , že je placený bude jistě zajímavé, když se ohlédneme po dalších variantách.

4 DATABÁZE POMOCÍ ODBC V PROSTŘEDÍ BORLAND DELPHI

Nyní si ukážeme, jakým způsobem řešit komunikaci mezi databázemi na rozdílných serverech pomocí programového prostředí Borland Delphi.Za tímto účelem jsem sepsal jednoduchý ukázkový program, jehož tvorbu a funkci budu nyní demonstrovat.

4.1 Úvod

Delphi, jako každý řádný vývojový nástroj mají řadu možností, jakým způsobem uskutečňovat komunikaci s okolními počítači , či počítači v internetu. Nás především zajímá opět komunikace s obecným databázovým SQL serverem, takže si ukážeme , jakým způsobem řešit tento problém.

4.2 ODBC

Jak již bylo řečeno, Delphi (podobně jako Borland C++), má řadu komponent určených pro komunikaci. Nicméně většině těchto komponent musíme databázi nějakým způsobem zpřístupnit, aby se byla vůbec schopna dostat k jednotlivým tabulkám. Nejuniverzálnějším nástrojem k řešení problému je užití ODBC.

4.2.1 Co je technologie ODBC?

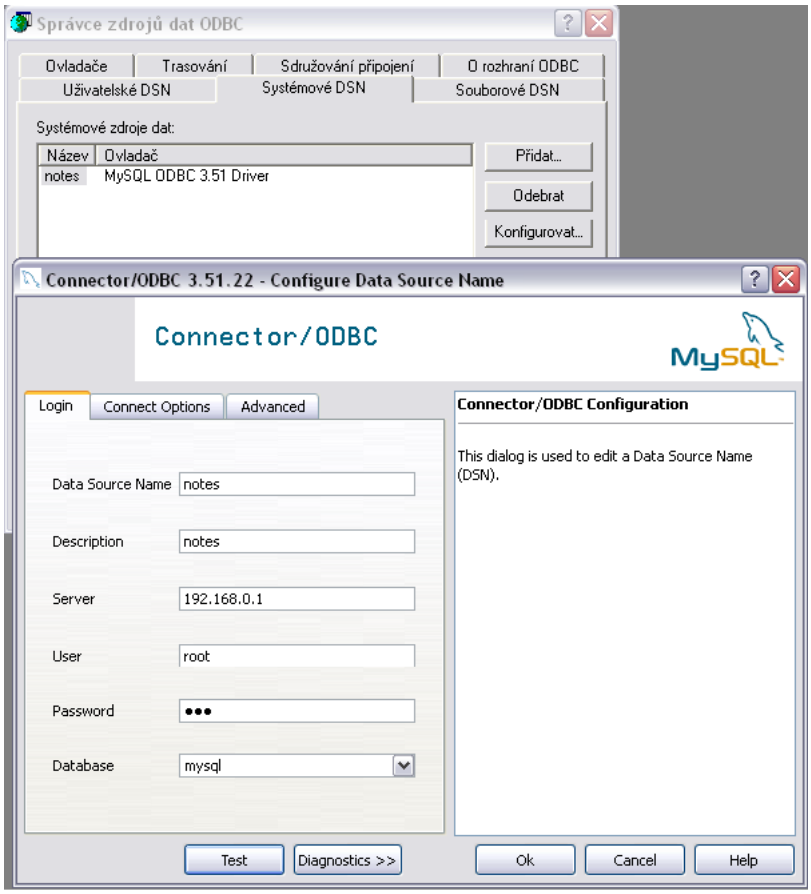
Technologie ODBC (Open Database Connectivity) umožňuje přesouvat data z jednoho typu databáze (zdroj dat) do jiného. K tomu je potřeba správný ovladač.[4] Vzhledem k tomu, že tento nástroj je velmi rozšířený, tak má velmi dobrou podporu pro všechny možné druhy databází.

4.2.2 Přínos ODBC pro řešení

Díky ODBC můžeme provést zcela obecné propojení našich komponent v Delphi s libovolnou databází. Náš programový kód nebude třeba měnit v závislosti na různých typech databází, neboť nám ODBC vše patřičně převede. Jediné co si budeme muset opatřit je správný ovladač pro danou databázi a pak již jen stačí nakonfigurovat připojení k patřičné databázi.

4.2.3 Konfigurace ODBC

Nyní si probereme nastavení technologie ODBC pro připojení k databázi. Podporu ODBC lze nalézt i v základní instalaci Windows XP. Do jeho nastavení se dostaneme z Ovládacích panelů , kliknutím na Nástroje pro správu a nyní zvolíme Datové zdroje(ODBC).Doporučuji zvolit záložku Systémové DSN.Nyní se nám zobrazí tabulka již vytvořených zdrojů dat. Je dosti možné že bude ve Vašem případě prázdná. Pokud tedy náhodou nemáme již vytvořené funkční spojení, bude třeba jej vytvořit(tlačítko Přidat...). V následující tabulce musíme vybrat vhodný ovladač pro daný typ databáze. Pokud jsme jej v seznamu našli, zvolíme jej, pokud ne, bude třeba ovladač nejprve do systému doinstalovat a teprve pak můžeme dokončit konfiguraci ODBC. My se nyní budeme chtít připojit k MYSQL databázi, takže vybereme tento ovladač.Nyní se dostaneme do samotného nastavení zdroje dat.



Obr. 4.1: Konfigurace zdroje ODBC

Nejdůležitějším nastavení najdeme v záložce Login. Tu je třeba vyplnit. Do položky "Data Source Name" napíšeme jméno zdroje dat. Pozor, nejedná se o jméno databáze na serverovém počítači, ale je to jméno jméno, které bude mít tento zdroj

dat na našem klientském počítači a to budeme rovněž zadávat do komponent v delphi. Další údaje jsou již zřejmé z obrázku (adresa serveru, a přihlášení uživatele). Pokud bude spojení úspěšné, zobrazí se nám v políčku Database seznam připojitelných databází. Vybereme z nich požadovanou a můžeme ještě provést zkoušku propojení na databázi, kliknutím na tlačítko Test. Potvrzením této tabulky máme nakonfigurovaný zdroj dat, který budeme dále používat.

Samozřejmě celý postup zopakujeme tolikrát, kolikrát se budeme potřebovat připojit k různých požadovaným serverům. V našem případě to budou 2 servery, takže budeme mít po úspěšné konfiguraci v systému 2 vlastní zdroje dat(tedy přidáme ještě další server).

4.3 Vhodné nástroje v delphi

Jelikož již máme vytvořený zdroj dat, můžeme přikročit k programovému řešení v delphi. V delphi lze nalézt spoustu komponent , které umožňují pracovat s daty či tabulkami a umožňují je vytvářet,editovat a mazat z programového kódu(např.Table). Další komponenty umožňují velmi snadné grafické zobrazení a editování ze strany uživatele(např. DBGrid). Jelikož užití těchto grafických nástrojů je poměrně snadné, nebudu se jimi příliš zabývat, navíc nejsou pro samotnou komunikaci stěžejní.

4.3.1 Komponenta Table

Tato komponenta patří do jednotky DBTables. Ve vizuálním prostředí ji najdeme standardně na záložce BDE.[2] Umožňuje přístup k datům z konkrétní tabulky nastavené databáze. V našem případě je místo nastavené databáze užit zdroj dat,což je ale velmi podobné. Komponentu je tedy třeba nakonfigurovat pro připojení k námi vytvořenému ODBC zdroji dat.To vyřešíme vyplněním její vlastnosti DatabaseName, které přiřadíme název patřičného zdroje dat.Komponenta umožňuje přístup ke všem záznamům dané tabulky. Zpřístupňuje je jak pro čtení, tak i pro zápis. Co se týče zápisu, je třeba mít na serveru patřičná oprávnění,jinak by jej nešlo použít!Záznamy je možné vytvářet, mazat, inovovat, přejmenovat.

Užité metody a properities komponenty Table[2]:

FindFirst - tato funkce hledá první záznam v tabulce a vrací true,je-li nalezen, jinak false
FindNext - tato funkce hledá další záznam v tabulce a vrací true,je-li nalezen, jinak false
FieldList - umožní přístup k názvům sloupců

FieldList.Count - vrátí počet sloupců tabulky
FieldByName(nazevSloupce) - tato metoda je spojena s aktuálním řádkem a zpřístupní data zadaného sloupce.Data jsou zpřístupněna buď pro čtení, či zápis(záleží na syntaxi).
Insert - vloží nový řádek do tabulky
Post - zapíše modifikované záznamy řádku do tabulky

4.3.2 Komponenta DataSource

Tato komponenta patří do jednotky DB. Vytváří rozhraní mezi komponentami dataset(datovým připojením) a komponentami uživatelského rozhraní na formuláři.[2] Pro řešení problému(tedy pro vlastní komunikaci mezi servery) , tahle komponenta není příliš potřebná, je jen doplňková. Umožňuje pouze vazbu na komponentu DBGrid,což je komponenta uživatelského rozhraní.

4.4 Ukázka vlastní komunikace

Nyní si předvedeme zmíněné metody v praxi. Omezíme se jen na krátkou ukázku kódu, kdy v jednom cyklu přeneseme všechny řádky z jedné tabulky(T1) do tabulky druhé(T2).

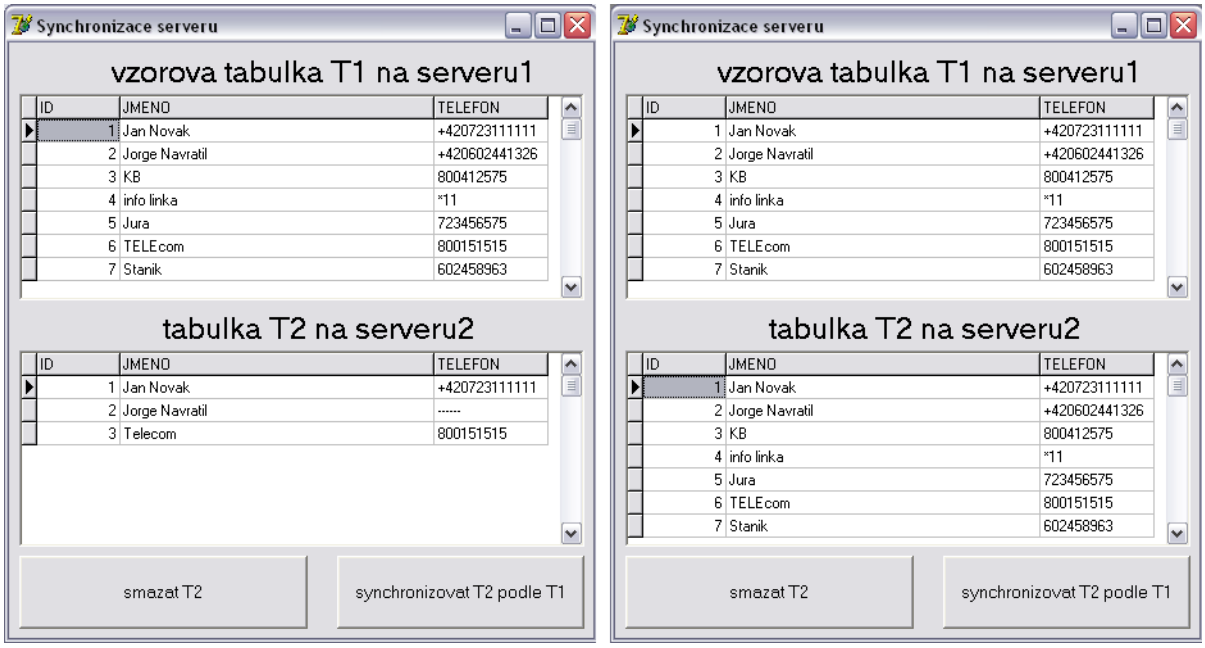
Obě tabulky se nacházejí na různých serverech. Na tyto servery, přesněji na patřičné databáze jednotlivých SQL serveru, jsou připojeny již dříve popsaným způsobem. Tedy pomocí komponenty Table, kterou pomocí vlastnosti Table.DatabaseName , navážeme na zdroj dat ODBC.

Ukázka programového kódu:

```
if t1.FindFirst=true then
begin
repeat
t2.insert;
for i:=0 to T1.FieldList.Count-1 do
T2.FieldByName(Sloupce[i]).AsString:=T1.FieldByName(Sloupce[i]).AsString;
t2.Post;
until t1.FindNext=false;
end;
```

4.5 Praktická ukázka programu

Mnou navržený ukázkový program nejprve zkontroluje zda-li souhlasí počty sloupců a názvy sloupců. Tím se ubezpečí že tabulky jsou stejné a po té provede překopírování jednotlivých záznamů.



Obr. 4.2: Vlevo je okno programu s databázemi před synchronizací a vpravo je synchronizace provedena (tabulky jsou shodné)

4.6 Závěr

Díky nástroji ODBC jsou zde všechny popsané techniky použitelné pro jakýkoli typ databáze , nejen tedy MySQL a SQL.

Tato aplikace dokáže udělat jednostrannou synchronizaci stejných tabulek na libovolných serverech. Samozřejmě by bylo možné ji rozšířit tak , aby umožňovala oboustrannou synchronizaci, ale vyžadovalo by to zavedení dalších sloupců do synchronizovaných tabulek, aby bylo možné srovnávat jednotlivé záznamy.

Vzhledem k tomu , že návrh probíhá v prostředí Borland Delphi, je možné takovou aplikaci rozšířit zcela libovolně a řešit i pokročilejší či specializovanou problematiku, jež se jiná prostředí určená pro pouze pro přenosy databází vůbec nezabývají.

5 SYNCHRONIZACE DATABÁZE NA MOBILNÍM ZAŘÍZENÍ S DATABÁZÍ NA PC

5.1 Úvod

Nyní se zkusíme zamyslet nad tím, jakým způsobem řešit případ, kdy potřebujeme synchronizovat data z SQL serveru, či jiného typu databáze, která je umístěná na PC se zařízením jako je mobilní telefon a pod.K synchronizaci budeme využívat internetového rozhraní.Tedy ze strany mobilního zařízení to bude nejspíš představovat datovou komunikaci pomoci GPRS/EDGE. Předpokládejme ze naše databáze na PC bude telefonní seznam, který bychom chtěli přenést z PC databáze na mobilní telefon. Naše řešení se může opřít o to, že takřka každý mobilní telefon podporuje java 1.0, nebo 2.0, či nějaká další rozšíření. Dá se říci, že i na pár let starém modelu najdeme minimálně javu ve verzi 1.0, což nám bude zcela stačit pro to , abychom dokázali přenášet data do aplikace na mobilním telefonu. Lze tedy říci, že naše řešení bude velmi obecné a nebude vůbec záviset na příslušném typu mobilního zařízení. Jen na okraj zmíním , že Java Microedition podporuje(nebo ji lze doinstalovat) i na spoustu handheldů (či jiných "kapesních počítačů").

5.2 Základní problematika

Zdá se , že hned na začátku narazíme na drobné úskalí v podobě podpory databází prostřednictvím Java Microedition. Asi těžko budeme instalovat na tuto platformu SQL , či MySQL server pro pohodlný přístup k datům. Jednak sehnat takový typ aplikace by bylo velmi problematické (pravděpodobně vůbec neexistuje) a navíc takové řešení by bylo velmi neohrabané a velmi náročné na systémové prostředky mobilního zařízení. Nesmíme také zapomínat na to, ze java je emulované prostředí, tedy že mezi samotný procesor a program je vsunuta Java Virtual Machine. Což samozřejmě přináší na jedné straně multiplatformnost, ale na straně druhé snížení výkonu příslušného programového kódu. Tudíž by takové řešení by mohlo být i zcela nad rámec systémových možností většiny takových zařízení. Takže bychom najednou ztratili i jistou obecnost a použitelnost.Bude třeba se ubírat jinou cestou.

5.3 Řešení databáze

Jak již jsem řekl, musíme zapomenout na SQL databázi. Ještě je otázkou, jaký typ databáze máme na PC. Jestli se jedná o vlastní databázi, či SQL. V případě SQL databáze budeme muset udělat nejprve převod do vlastního databázového formátu.

Převod by opět mohl probíhat již probranými synchronizačními technikami, kdy z databáze například postupně vykopírujeme jednotlivé řádky a vložíme je do naší vlastní databáze. V případě že používáme jen vlastní databáze, potom se převody nemusíme zabývat a přikročíme k návrhu vlastní databáze.

5.4 Uspořádání v paměti

Jelikož vše řešíme vlastními silami, tedy není tu sql server, který za nás ukládá data do paměti a nějakým způsobem je uspořádává, budeme muset data do paměti ukládat ručně a navrhnout si vlastní struktury a jejich uspořádání.

Nejjednodušším a nejpřímějším uspořádáním takových dat je pole. Díky tomu můžeme mít snadný přístup k jednotlivým "řádkům" převáděné tabulky.Vytvoříme-li dvourozměrné pole, získáme tak i snadnou indexaci sloupců tabulky.Dále si budeme muset někam zaznamenat názvy sloupců. Např. na to můžeme použít další ,již jen jednorozměrné pole sloupců.

Původní tabulka:

RADEK	JMENO	PRIJMENI	CISLO	OPERATOR
1	Jiří	Navrátil	602111222	O2
2	Petr	Sedlařík	605123222	T-Mobile
3	Jan	Olsina	607111444	Vodafone

Dvourozměrné pole s daty tabulky

1	2	3	4	5
1	Jiří	Navrátil	602111222	O2
2	Petr	Sedlařík	605123222	T-Mobile
3	Jan	Olsina	607111444	Vodafone

Jednorozměrné pole s názvy sloupců:

1	2	3	4	5
RADEK	JMENO	PRIJMENI	CISLO	OPERATOR

Pro ukázku zavedeme předpoklad, že indexy pole jsou číslované od 1. Budeme-li chtít v programovém kódu se dotázat například na 2. řádek a druhý sloupec a uložit jej do proměnné s typu string uděláme to následovně.
s:=PoleSDaty[2,2];

Pokud nás bude zajímat název sloupce příslušného záznamu, stačí užít

nazevSloupce:=poleSloupcu[2];

(Použito syntaxe Borland Delphi, Pascal)

Nyní máme vyřešenou strukturu dat v paměti. Tento návrh je snadno implementovatelný jak v prostředí typu Borland Delphi, či v prostředí Java,neboť N-rozměrné pole patří k výdobytkům všech programovacích jazyků.

5.5 Vygenerování databázového souboru

Data máme dle předchozího návrhu uložená v paměti RAM, což z dlouhodobého hlediska nebude asi zrovna dostatečné. Musíme tedy vygenerovat nějaký soubor, který uložíme na pevný disk.

5.5.1 Data se statickou strukturou bloku

Data se statickou strukturou jsou data s pevně definovanou délkou jednotlivých záznamů(bloků).Jinými slovy všechny "řádky tabulky" budou mít stejnou bytovou délku. Bude se měnit pouze počet jednotlivých řádků, neboť by bylo velmi neohrabané jej na začátku pevně definovat a blokovalo by se tím zbytečně velké množství paměti. Přepsat statickou strukturu paměti do souboru je poměrně jednoduché. Stačí pevně nadefinovat délky jednotlivých záznamů (tohle již vychází již z dat v paměti RAM) a zapsat je v přesném pořadí do souboru. Následně lze velmi jednoduše opačným algoritmem v přesném pořadí znovu soubor načíst a zavést data do operační paměti.

5.5.2 Data s dynamickou strukturou souboru

Nabízí se otázka, co s nějakými nedefinovanými řetězci, které mají velmi proměnnou délku. V tom případě by nebylo statické uspořádání nejlepší. Neboť by muselo být zavedené nějaké výrazné omezení délky záznamu a nebo by bylo třeba alokovat a v souboru vyhradit velké množství paměti, které by většinou zcela padlo nazmar. Jediná možnost, jak tuto situaci řešit je použití dynamické struktury pro problémové (délkově různorodé) prvky databáze, případně dynamicky ukládat všechny. Zde je generování souboru o něco složitější , neboť se nemůžeme opírat o nějakou pevnou bytovou strukturu. Budeme muset tedy použít ukončovací znaky pro jednotlivé řetězce. Tento přístup je vhodný především pro textové řetězce, neboť nemají povolené všechny znaky a tak můžeme jednoznačně identifikovat ukončovací znak. V případě

číselných proměnných je tato identifikace velmi problematická, zde je nejlepší volit statický přístup, neboť i v samotné paměti počítače jsou většinou číselné proměnné statické.Tedy vyhradit si pevně délku pro uložení proměnné a nepoužít ukončovací znak.

5.5.3 Zvolení vhodného typu souboru

Nyní si umíme představit, který přístup je vhodný pro jaký typ záznamů, Budeme-li dělat zcela obecný systém, bude skutečně lépe použít dynamickou strukturu. Nicméně vzhledem k tomu, že našim předpokladem je přenášet tabulku telefonního seznamu(či podobnou databázi), která obsahuje délkově vyrovnané položky a s přihlédnutím na omezené systémové a zobrazovací schopnosti mobilního zařízení, nebude nikterak na škodu, použijeme-li statický přístup.

5.6 Zpřístupnění dat z PC/serveru

Jelikož PC strana je samotný server, lze základní komunikaci navrhnout tak, že serverová aplikace vygeneruje příslušný databázový soubor (popsáno výše) a uloží jej do něj přesného umístění. V příp. pokročilejšího systému sdělí java aplikaci jeho umístění. Nyní budeme uvažovat jednosměrný tok dat ze serveru ke klientům. Tedy java aplikace bude znát přesné umístění databázového souboru v internetu a pro daný typ synchronizace bude tento soubor stejný. Budou-li se synchronizovat třeba ještě "skladové zásoby", budou reprezentovány dalším souborem, jehož umístění bude opět pevně nadefinováno.

5.7 Komunikace v javě

5.7.1 Potřebné interface a třídy pro komunikaci[3]

StreamConnection- je interface definující potřebné vlastnosti pro komunikaci.Je součástí balíčku javax.microedition.io.

InputStream- Tato abstraktní třída je supertřídou reprezentující příchozí proud bytů.Je součástí balíčku java.io.

5.7.2 Zprovoznění komunikace

Zde uvádím ukázku části programového kódu v jave, který slouží pro komunikaci přes internet. Takto se lze připojit na danou adresu a pomocí proudového připojení stáhnout příslušný soubor.Nejprve proběhne vytvoření připojení a následně cyklus

while probíhá tak dlouho, dokud přicházejí data.Všechna přijatá data jsou zapsána do bytového pole inBuf. Viz komentáře části programu.

```
try
{
StreamConnection strConNet = null;
// vytvořím instanci interface StreamConnection
InputStream inStrNet = null;
// vytvořím instanci třídy InputStream
strConNet = (StreamConnection)Connector.open(adresa);
// zavolám metodu třídy Connector, která nám vrátí instanci třídy Connection. Tu pak přetypujeme na instanci StreamConnection a přiřadíme ji do proměnné strConNet - tj. na proudové připojení
inStrNet = strConNet.openInputStream();
// otevírám rouru neboli proud příchozích dat
int celkemNacteno=0;
int praveNacteno=0;
byte [] inBuf = new byte [2000];
// vytvořím statické pole bytu, do kterého dáváme příchozí data
while (true) {
praveNacteno=inStrNet.read(inBuf, celkemNacteno, 10);
// vrátí počet načtených bytu (maximálně 10) a vrátí hodnotu -1 pokud už není co číst
// deklarace funkce read(poleBytu,od kolikátého bytu zapsat data, kolik načíst bytu maximálně)
if (praveNacteno!=-1) celkemNacteno+=praveNacteno; else break;
}
}
catch (Exception e)  // odchytí výjimku
return ("Nastala chyba: " + e.toString());
}
```

5.8 Zhodnocení a další možnosti

V této sekci byla popsána jednosměrná synchronizace, kdy primární data vytváří server a rozesílají se ke klientům. Tento návrh počítá s klienty na mobilních zařízeních s java microedition. Nicméně není problém tento návrh přenést do klasické javy, která může běžet i na běžných počítačích a tudíž lze rozšířit pole působnosti

tohoto návrhu. Dále by stálo za úvahu rozšíření této jednostranné synchronizace na oboustrannou, tedy umožnit klientovi měnit záznamy.Zde by musela být dále ošetřena přístupová práva k serveru, popsané programové funkce by mohli být podobné, většinou k nim existují ekvivalenty v podobě funkcí pro odchozí připojení(např. InputStream x OutputStream).

6 MULTIPLATFORMNÍ OBOUSTRANNÉ PŘENOSY MEZI KLIENTEM A SERVEREM

Doposud popsaná řešení se zabývala buď přenosy či synchronizacemi mezi jednotlivými servery a nebo jednostranný přenosem databáze do mobilního zařízení, pomocí stažení souboru. Pokud však budeme chtít zajistit plnohodnotný přístup na SQL databázi z několika platforem brzy narazíme na problém.

6.1 Moderní požadavky a jejich úskalí při implementaci

Předchozí kapitole bylo popsáno , jak zajistit přenos celé, části databáze do mobilního zařízení. Zvolená metoda využívala základních vlastností javy , se kterou jde velmi snadno a rychle přistupovat k souborům umístěných na internetu. Díky tomu se dalo na pár řádcích kódu realizovat stažení souboru, pak již jen stačilo jej vhodně dešifrovat dle známé struktury. Tedy poskládat k sobě příslušná data a zobrazit je uživateli, či mu v nich umožnit vyhledávat.

Ovšem tento způsob je použitelný jen pro malé objemy databází a také jen tehdy , když bylo požadavkem přenášet každému klientovy stejnou část aktualizované databáze,či celou databázi. Ovšem uvědomíme-li si fungování dnešních databázových systémů a také rozsáhlost databází se kterými klient musí pracovat, pak je toto řešení naprosto nevyhovující.

Jednak je třeba zajistit klientovi plnohodnotnou komunikaci ze serverem. Tedy ne jen umožnit jednostranně stahovat předem definované úseky dat, ale také mu umožnit data přidávat, případně modifikovat. Taktéž nepostradatelnou částí je vyhledávání v databázi. Všechny tyto zmíněné případy , kdy klient dává jednoduché požadavky na server, je třeba tedy realizovat jinak, nežli stahováním souborů.Nejoptimálnější cestou je posílat serveru předem definované (datově neobjemné) povely, které jednak bude možno velmi rychle přenést i přes pomalá mobilní připojení a taktéž tím odpadne jistá zátěž serveru.

6.1.1 Možnosti PC klienta

Ideální tedy je, aby klientská aplikace rovnou komunikovala s databází. V případě klientů napsaných pro PC , např. v programovacím jazyce Delphi, nebo C++, to jistě není problém. Většina známých jazyků skutečně disponuje prvky pro snadné připojení databází a zpřístupnění požadovaných tabulek. Potom se již dá velmi snadno

pracovat s tabulkami, buď pomocí SQL příkazů vhodně implementovaných do daného programového prostředí, nebo použitím metod zvolených komponent, či knihoven, které nám nabízí rovnou naše programové prostředí.

6.1.2 Možnosti J2ME klienta

Vyjdeme-li z toho, že Java Microedition je značně ořezaná od plnohodnotné verze Javy, je celkem jasné, že tento klient bude mít daleko méně možností.

Jak již bylo řečeno dříve, pro přímou komunikaci s SQL databází nemá Java Microedition 2.0 žádné prostředky. Výrobce zřejmě nepočítal s tím, že by bylo někdy vhodné navazovat tato spojení i z těchto platforem. Toto tvrzení ovšem nevylučuje existenci mobilního zařízení, které tuto komunikaci dokáže realizovat. U javy Microedition je běžné, že k základním funkčnostem Java Microedition 2.0, si daný výrobce přidá svůj rozšiřující balíček. Ovšem nejedná se o obecný standart a pokud by takový balíček existoval, pak bude aplikovatelný jen na malou škálu mobilních zařízení. Jinými slovy, i kdybychom takovou podporu našli je pro obecné použití zcela nevyhovující. Toto je ovšem velmi nepříjemný fakt, který prakticky znemožňuje přímou komunikaci s databází.

6.1.3 Otázka zabezpečení

Další otázkou je, zda-li je vůbec vhodné (či do jaké míry) zpřístupňovat uživateli naši databázi. Jak již jsme řekli , uživatel potřebuje mít práva zápisu. Pokud by se vše odehrávalo tak, že by měl mít uživatel možnost modifikovat každou část naší databáze, pak se ovšem dostáváme k nepříjemnému faktu, že kdokoli jiný, ji může také celou zničit, či poškodit. Samozřejmostí dnešních databázových SQL serverů je existence uživatelů a možnost uživatelům přiřazovat různá práva k různým tabulkám. Nyní by se mohlo zdát, že je po problému. Tato otázka zabezpečení by dostačovala možná pro nějaké firemní řešení , ovšem pro obecné použití je tato technika stále nevyhovující. Proč? Důvod je jasný. Každému z uživatelů můžeme zajistit (grantovat) práva na dané tabulky. Ovšem je zcela jasné, že v obecném systému budou existovat tabulky , do nichž budou zapisovat různí uživatelé.(bude to např. potřebné z hlediska použité databázové struktury). Ovšem zároveň žádný z uživatelů by neměl mít plnou moc nad všemi daty z tabulky.

Končí tím tedy možnosti zabezpečení na databázi ? Jistě že ne. Moderní databáze nám nabízí další techniky v podobě uložených procedur (Stored Procedures). Každá dynamicky se rozvíjející dnešní databáze již tyto techniky podporuje.

Každá databáze může používat svůj jazyk pro psaní těchto procedur. V reálném prostředí se buď jedná o známé platformy (např. C), či jejich blízké modifikace.

V rámci uložených procedur lze realizovat další funkčnosti databáze, na kterou nám běžné dotazovaní pomocí sql nestačí. Zvláště výhodný je ten fakt, že tato funkčnost je realizována ihned na serveru, takže díky tomu lze klientovi zpřístupnit např. jisté části databáze, tedy zavést konkrétnější bezpečnostní prvky, než ty, které plynou z přímého přidělování práv k daným tabulkám.

6.2 Zhodnocení faktů

Z průzkumu tedy vyplývá, že jisté (mnohdy dostačující zabezpečení) lze provést pomocí uložených procedur. Na druhou stranu má tak stále klient (tím i útočník) možnost přímého dosahu databáze. Jinými slovy, mezi databází a klientem není žádná další vrstva. Je tedy otázkou administrátorů databáze, jestli tímto způsobem dokáží zabezpečit celou databázi (a jestli odstraní všechny možné nabouratelné prvky).

Je tedy zřejmé že z hlediska plné bezpečnosti a autorizace by bylo ještě lepší , kdyby se mezi databázi a klienta vložila další "vrstva".

Dále musíme mít na paměti onen nepříjemný fakt, že J2ME klient, se kterým chceme také komunikovat, neumí přímo komunikovat s naší SQL databází. Je tedy bezpodmínečně nutné , aby se mezi klienta a databázový server vložil nějaký další prostředník.

Podíváme-li se na existující řešení komunikací s databázemi, zjistíme, že tento způsob se běžně používá a tato zprostředkovatelská aplikace nese pojmenování "Aplikační server".

7 APLIKAČNÍ SERVER - ZÁKLADNÍ POZNATKY

7.1 Co je to aplikační server

Aplikační server tvoří vrstvu mezi operačním systémem a aplikacemi. Podobně, jako operační systém poskytuje základní funkce programům (například pro přístup k souborovému systému, nebo ke správě procesů), poskytuje aplikační server často používané funkce enterprise aplikacím. Vytváří další vrstvu abstrakce, aby bylo psaní aplikací jednodušší. Příkladem takových funkcí mohou být podpora transakčního zpracování požadavků, persistence objektů do databáze, výměna zpráv mezi aplikacemi a další. Nabízí se samozřejmě otázka, co to vlastně je enterprise aplikace a jak se liší od běžné aplikace. Není to nic složitého, de facto se jedná o běžnou aplikaci, na kterou jsou kladeny určité nároky co se týče spolehlivosti, dostupnosti, robustnosti, výkonnosti. Typická je také potřeba obsloužit současně velké množství požadavků (klientů). Klasickými zástupci enterprise aplikací jsou moderní webové aplikace a řešení postavená na servisně orientované architektuře (SOA).[10]

7.2 Využitelné možnosti aplikačního serveru

7.2.1 Multiplatformnost komunikace

Tato aplikace nám otevírá velké možnosti v oblasti komunikace, synchronizace, multiplatformnosti klientu a databází. Můžeme díky ní transformovat a realizovat nejrůznější typy jinak nekompatibilních komunikací,či synchronizací.

Např. můžeme unifikovat požadavky klientů na nejrůznějších platformách(aby byla komunikace kompatibilní) a pak se s těmito klienty připojit na aplikační server. Aplikační server dále vyhodnotí požadavky klientů a směřuje je např. do daných databázových struktur. Z databází následně získá potřebná data, popřípadě na základě klientských požadavků přímo modifikuje databázi, a návratová data pak opět pošle klientským aplikacím.

7.2.2 Zabezpečení databáze

Ovšem není to jediná možnost, kterou lze využít při realizaci našeho projektu. Díky tomu , že jsme vložili mezi databázi a klientskou aplikaci tento aplikační server, můžeme realizovat jakékoli typy zabezpečení a uživatelských práv k jednotlivým prvkům v databázi. Všechna práva , restrikce atd. si lze naprogramovat.

Pro začátek v našem řešení využijeme tyto poznatky např. při autorizaci uživatele. Klient tedy zadá login (jména a heslo). Požadavek přijde na aplikační server,

který si sám rozhodne o tom, jestli tohoto klienta přijme , zamítne , či jaká práva mu přidělí. Následně po úspěšné autorizaci zprostředkovává klientovi povolenou komunikaci. Tedy klient nemá žádnou možnost, jak se přímo dostat k databázi a nemůže ji nikterak atakovat (pouze přes aplikační server, který má však režii plně pod kontrolou).

Ovšem aby se toto dalo sto procentně realizovat, je třeba mít na paměti, že databáze nesmí nijak komunikovat s nikým jiným, nežli s aplikačním serverem. Tedy musí ji být odepřeno jakékoli nedůvěryhodné vnější okolí.Tedy je např.přípustné umožnit přímý přístup jiným administrátorským aplikacím v důvěryhodné síti.

7.2.3 Přístup k několika databázím, rozklad zátěže, synchronizace

Jelikož je aplikační server plnohodnotnou aplikací, můžeme s ním provádět i komplikovanější věci, než-li zprostředkovávat spojení klientovi. Především vyjděme z faktu, že aplikační server bývá navázán na databázi, případně databáze. Tedy lze jím, bez vědomí klienta zpřístupňovat různé fyzické databázové servery, ať již v závislosti na typu požadavku(informace), nebo v závislosti na momentálním zatížení databázových serverů. Takže z toho plyne , že aplikační server se dá použít i pro rozložení zátěže na jednotlivé databázové servery. V reálném světě existuje velké množství databázových systémů, které nemůže obsloužit jeden jediný počítač a tak musejí využívat mnohdy i několik desítek dalších serverů.

Taktéž by se takový aplikační server dal využít pro synchronizace databází, neboť pokud má přístup k několika různým databázím současně a pokud rovněž zprostředkovává veškeré toky dat ke klientům, pak disponuje všemi potřebnými informacemi proto, aby mohl synchronizovat databázové servery mezi sebou.Neboť může rovnou nová data od klientů např. rozkopírovávat ihned na více databázových serverů, příp. v přesně vytyčený čas provádět synchronizaci atd. Ovšem toto jsou v reálném nasazení velmi komplikované záležitosti.

7.3 Přínos pro řešení

Z předešlé kapitoly využijeme především možnosti zabezpečení a možnost multiplatformní komunikace. Jelikož následující realizace bude primárně počítat s jedním fyzickým serverem, na kterém bude nainstalován aplikační a databázový systém, není třeba příliš uvažovat rozkládání výkonu na několik různých fyzických serverů.

Jen pro doplnění, u skutečně výkoných databázových systémů nemusí být aplikační server v pravém slova smyslu představován jednou jedinou aplikací, ale jeho funkčnost může zabezpečovat i soustava serverů.

7.4 Metodiky realizace aplikačního serveru

Nyní jsem nastínil, co je to aplikační server, dokonce i jak a na co jej použít. Ovšem skutečný problém nás teprve čeká. Musíme vybrat vhodné programové prostředí ve kterém takovou aplikaci vytvoříme.Možností je celá řada. Jak již bylo zmíněno, aplikační server je v podstatě naprosto běžná aplikace, na kterou jsou však kladeny nesmírné nároky zejména v oblasti komunikace a stability. Z toho tedy plyne, že teoreticky se dá naprogramovat v jakémkoli vývojovém prostředí, které disponuje podporou síťové komunikace a podporou přístupu k databázím. Také není od věci, že by toto vývojové prostředí mělo tvořit efektivní kód.

7.4.1 Existující aplikační servery

Jelikož v moderních komunikacích není pojem aplikační server nikterak neznámý, tak nás jistě nepřekvapí existence návrhových prostředí přímo pro aplikační servery. Často jsou tyto systémy draze placené a vyžadují speciální jazyk , příp. nějakou mutaci jazyka existujícího. Ovšem dají se najít i open source projekty. Např. Jboss,Tomcat,GlassFish.Tyto servery využívají ke svému běhu většinou java machine, tedy musí být podpora javy instalována na serverovém PC. Sami pak programátorovi nabízejí využití javovské syntaxe pro realizaci hlavní funkčnosti aplikačního serveru. Výhodou těchto serverů je, že mají již v sobě obsaženu základní funkčnost pro komunikaci, kterou by jinak programátor musel sám zabezpečit.

Nevýhodou, zvláště open-source projektů, je velmi špatná počáteční srozumitelnost, či neexistence návrhových prostředí. Připouštím , že toto hodnocení je velmi začátečnické, ale pokud se má programátor, který se nově seznamuje s tímto prostředím, spokojit pouze s administrační konzolí, pak je to jistě velké úskalí.

Další otázkou je efektivnost aplikačního serveru psaného v emulovaném prostředí javy. Tady se můžeme jen dohadovat, obecně však nelze popřít tvrzení, že jakákoli emulace musí být ztrátová. Ovšem pro reálná výkonnostní posouzení těchto produktů nemám dostatečné informace.

7.4.2 Aplikační server v Borland Delphi 2007

Jak již bylo zmíněno, pro neznalého člověka představují tyto open-source systémy dosti vysoké úskalí. Vyjdeme-li z předpokladu, že běžný člověk zná dobře vývojové prostředí např.Borland Delphi nebo C++,či Microsoft Visual C++ pak je jistě zajímavou variantou zamyslet se nad realizací v těchto jazycích. Já sám jsem se v důsledku vysokých zkušeností s vývojovým prostředí Delphi rozhodl prozkoumat právě jeho možnosti. Po letmém průzkumu jsem zjistil, že Delphi disponuje velkou

škálou komponent jednak využitelných pro obecné připojení mezi klientem a aplikačním serverem (např soketová připojení) a zároveň že má dostatečnou podporu pro propojení s databází.

Použitím tohoto vývojového prostředí tedy odpadá (pro člověka znalého) nějaké komplikované zkoumání neznámých systémů, ovšem tím pádem musí mít více namysli optimalizace a musí se více zamýšlet nad tím, kterou metodiku přístupu použije, neboť některé metodiky v Delphi nemusí být (a ani nejsou) optimalizovány pro takové vysokokapacitní použití.

8 APLIKAČNÍ SERVER V DELPHI 2007 - KOMUNIKACE S KLIENTY

8.1 Úvod, rozsah platnosti tohoto řešení

V předchozích kapitolách jsme si nastínili možnosti aplikačních serverů a možné platformy pro vývoj. Z těchto informací jsem dospěl k závěru, že bude velmi zajímavé podívat se na tuto problematiku rovnou z pozice běžně používaného vývojového prostředí a proto jsem se rozhodnul zvolit platformu Borland Delphi 2007.

Proto dále zmiňované metodiky budou tedy konkretizovány především pro tuto platformu. Taktéž musím upozornit, že z hlediska dynamičnosti vývoje zvláště síťových komponent, je pro plnou platnost následného řešení nutné dodržet i rok vydání tohoto vývojového prostředí, příp. doinstalovat do starších verzí podporu nových komponent.

Ze svých zkušeností mohu říci, že velká část použitých metodik by se měla prolínat i s možnými metodikami v Borland C++ 2007. Jelikož obě prostředí mnohdy využívají velmi podobných, či stejných komponent.

8.2 Komunikační protokol - úvahy

Jelikož se nyní budeme vytvářet zcela vlastní řešení, bude nutné si vytvořit nějaký řád komunikace. Jinými slovy, unifikovat komunikaci tak, aby byla stále stejná bez závislosti na platformě. Taktéž odstranit různorodost datových struktur, která by se mohla vyskytnout např. mezi Javou a Borland Delphi.Toto vyřešíme zavedením vlastního, jasně definovaného protokolu.

Ovšem abychom mohli protokol správně sestavit, musíme projít několik dalších úvah.Zamyslet se nad vhodným síťovým protokolem, vhodně definovat velikosti přenášených struktur, zajistit přeposílání základních požadavků klienta a základních stavů serveru ke klientovi atd.

8.2.1 Síťový protokol UDP

UDP protokol (User Datagram Protocol) je jedním ze sady protokolů internetu. O protokolu UDP říkáme, že nedává záruky na datagramy, které přenáší mezi počítači v síti. Někdy je označován jako nespolehlivý, ale to je velmi zavádějící označení. Na rozdíl od protokolu TCP totiž nezaručuje, zda se přenášený datagram neztratí, zda se nezmění pořadí doručených datagramů nebo zda se některý datagram nedoručí vícekrát.

Protokol UDP je vhodný pro nasazení, které vyžaduje jednoduchost nebo pro aplikace pracující systémem otázka-odpověď (např. DNS, sdílení souborů v LAN). Jeho bezstavovost je užitečná pro servery, které obsluhují mnoho klientů nebo pro nasazení, kde se počítá se ztrátami datagramů a není vhodné, aby se ztrácel čas novým odesíláním (starých) nedoručených zpráv (např. VoIP, online hry).[13]

8.2.2 Síťový protokol TCP/IP

Tvůrci protokolů TCP/IP naopak vycházeli z předpokladu, že zajištění spolehlivosti je problémem koncových účastníků komunikace, a mělo by tedy být řešeno až na úrovni transportní vrstvy. Komunikační podsíť pak podle této představy nemusí ztrácet část své přenosové kapacity na zajišťování spolehlivosti (na potvrzování, opětné vysílání poškozených paketů atd.), a může ji naopak plně využít pro vlastní datový přenos. Komunikační podsíť tedy podle této představy nemusí být zcela spolehlivá - může v ní docházet ke ztrátám přenášených paketů, a to bez varování a bez snahy o nápravu. Komunikační síť by ovšem neměla zahazovat pakety bezdůvodně. Měla by naopak vyvíjet maximální snahu přenášené pakety doručit (v angličtině se v této souvislosti používá termín: best effort), a zahazovat pakety až tehdy, když je skutečně nemůže doručit - tedy např. když dojde k jejich poškození při přenosu, když pro ně není dostatek vyrovnávací paměti pro dočasné uložení, v případě výpadku spojení apod. Předpokládá jednoduchou (ale rychlou) komunikační podsíť, ke které se připojují inteligentní hostitelské počítače.[12]

8.2.3 Zvolení vhodného síťového protokolu a optimalizace přenosu

Z teorie tedy vyplývá, že pokud použijeme UDP protokol, budeme se muset starat při našich vlastních přenosech o kontrolu správnosti dat. Což je jistě zbytečná komplikace, takže bude ideální dále počítat s bezpečným protokolem TCP/IP a tomu dále přizpůsobovat naše komunikační schéma.

Jedno z přizpůsobení je tedy naprosto jasné, je zbytečné síťovou komunikaci zatěžovat s jakýmkoli vlastním ověřováním správnosti přenosu dat.

Dovolím si mírně předběhnout, při návrhu jsem právě testoval i zmíněné další ověřování i přes použité TCP/IP. Výsledek tohoto testu byl až zarážející, došlo skutečně k velmi markantnímu poklesu rychlosti mezi serverem a klientem. Příčinou všeho bylo zasílání velmi malých paketů (např jejich funkční informace obsahovala jen jednotky bytu) a následné čekání na potvrzovací paket.A až po jeho doručení pokračovalo zasílání dalších informací.Toto byl princip starších sítí a proto není dobrý pro dnešní řešení, zvláště když používáme zabezpečený protokol TCP/IP.

Dále z teorie a z mého testu vyplývá, že bychom se měli vyvarovat posílání zbytečně malých paketů. Jinými slovy bychom naši odesílanou informaci neměli sekat zbytečně na velké množství malých paketů, neboť pak velké procento skutečně přenášené informace padne na síťovou režii (hlavička paketu ...) a ne na přenos našich dat.

Pro řešení tohoto problému máme dvě možnosti. První možností je, že si všechny nerozměrné datové typy (např,integer,real, či krátký řetězec) naskládáme do nějaké větší struktury a tu pak odešleme najednou.Tím pádem se nám vytvoří pakety o ideální velikosti, protože samotné rozsekání informace na pakety si již zajistí síťová režie, která je pro tuto práci optimalizovaná.

Další možností je zadání dat pro odeslání po malých částech, ale v rámci síťové komponenty zapnout buffer. Data je potom možno do komponent skutečně plnit po nerozměrných typech a na závěr použít příkaz k vyprázdnění bufferu. Tento způsob jsem použil ve svém programovém zpracování a také bych jej doporučil pro jednodušší implementaci.

Programový kód pak může vypadat následovně

```
// odesílá se integer (velikost následného stringu)
AContext.Connection.IOHandler.Write(length(polozka.Sloupec1));
// odesílá se samotný string
AContext.Connection.IOHandler.Write(polozka.Sloupec1);
// odesíla se kontrolní znak (char) o ASCI hodnotě 127
AContext.Connection.IOHandler.Write(#127);
// Povel k vyprázdnění bufferu, tedy doposud bufferovaná data k odesláni jsou nyní odeslána
AContext.Connection.IOHandler.WriteBufferFlush;
```

Tento kód je vytržen z kontextu, který by byl třeba pro jeho plné pochopení. Kompletní princip této komunikace bude vysvětlen v dalších kapitolách.

8.2.4 Stavové a kontrolní znaky, hlášení o chybách

Při naši komunikaci bychom také měli mít na paměti všechny úlohy a stavy serveru (příp.i klienta), do kterých se během komunikace dostane.Např při při práci s databází víme , že server se bude nacházet buď v bloku hledání položky pro klienta(SELECT), nebo v bloku přidávání nové položky. Musí tedy pro tyto úkony existovat nějaké jednoznačné značky(znaky) SELECT/INSERT. Taktéž pro ně musí

platit, že mají svoji přesně definovanou pozici v rámci komunikačního schématu. Splněním těchto požadavků můžeme potom jednoznačně server uvést do módu hledání, či vkládání.

Dále je vhodné, aby po vykonané práci zahlásil server klientovi stav READY. Tuto událost lze sdělit i jedním bytem, protože dle komunikačního schématu server i klient přesně ví, kdy má být přesně zasláno. Tento znak dále může sloužit k potvrzení, či informaci o úspěšném provedení či neprovedení daného úkonu. Např. si představme situaci, kdy klient přidává na server nový záznam a v průběhu přidávání server zjistí(ať již aplikační, či databázový), že jej z nějakých důvodů nemohl přidat. Tuto informaci je samozřejmě zapotřebí ohlásit klientovi. Server to vyřeší další dvojicí kontrolních znaků(bytů) PŘIJATO/ODMÍTNUTO.

Taktéž můžeme do komunikačního schématu vložit nějaké další znaky, které nemusí nutně symbolizovat zprávu serveru či klienta o úspěšnosti či neúspěšnosti, ale můžou sloužit jen jako kontrolní znak pro ověření dané pozice v komunikaci. Vzhledem k tomu, že používáme zabezpečený protokol TCP/IP není toto zrovna nutné. Ale takové porovnávání se např. hodí při odlaďování vlastního kódu, kdy se může stát , že klient a server z nějakých důvodů nemají úplně kompatibilní komunikaci. Např jsme zapomněli do jednoho z nich zapsat povel k odeslání či přijetí nějaké malé proměnné. I takový zdánlivý detail přivodí zkázu komunikačního schématu. Díky vloženým kontrolním znakům lze pak při podrobné analýze (bytový výpis záznamu komunikace do souboru,na displej nebo krokování kódu) zjistit poslední úspěšné (ještě synchronizované místo) a tím přibližně lokalizovat místo chyby.Konec konců tyto kontrolní znaky zabírají velmi málo (pár bytů) a tak nemusí být v komunikačním schématu obsaženy jen při odlaďování, ale mohou sloužit ke kontrole trvale. Uvědomíme-li si, že v dnešní době co chvíli přichází nějaká další inovace a s ní i odlaďování nové aplikace, pak se nám zase mohou brzy shodit a díky nim opět snadno odhalíme nekompatibilitu nové klientské aplikace a snadno lokalizujeme problém.

8.3 Komunikační protokol - sestavení, ukázka

8.3.1 Vysvětlení pracovní terminologie a symboliky, zavedení předpokladů

Tato část bude věnovaná praktické ukázce přenosového protokolu.Proto bude zapotřebí si zavést jistou symboliku.Volím tak z důvodu lepšího porozumění a kratšího následného zápisu.

Demonstrační tabulky se skládají vždy z 5 sloupců. První sloupec je pouze číslem řádku a je navigací pro čtenáře. Další sloupec určuje zařízení jemuž náleží prostor

pro komunikaci. Použité zkratky Server-S, Klient-C. Třetí sloupec vysvětluje úlohu přenášeného bloku dat, tedy k čemu slouží tento blok dat při komunikaci nebo co se jím přenáší za data.Sloupec velikost určuje rozměr bloku. Tento rozměr může být buď pevně definován, v tomto případě je v tabulce číslo udávající počet bytů a nebo jeho rozměr může být definován dynamicky(byl zaslán v předchozím přenosovém bloku) a v tabulce je tato skutečnost označena zkratkou Dyn.Poslední sloupec HODNOTY udává možné nabývající hodnoty (pokud je lze předem jednoznačně specifikovat).Jedná se především o stavové znaky (např READY s hodnotou 32) a o kontrolní znaky.Tyto hodnoty vyjádřené číslem jsou skutečnou hodnotou bytu (v rozsahu od 0 do 255), nebo je lze interpretovat také jako dekadickou ASCI hodnotu znaku (tedy charu).

Následující upřesnění se týká číselných proměnných. V případě odesílání délky záznamu je v tabulce velikost takové proměnné deklarována jako 4 byty.Jedná se tedy o klasický integer (v pojetí Delphi!)

Dále bych chtěl upozornit na údaj v závorkách,ve sloupci "Význam bloku dat".Tento údaj je přesným názvem reálného sloupce v databázové SQL tabulce. Z pohledu klienta není vůbec podstatný, ovšem z pohledu programátora je tomu jinak. Jelikož dotaz klienta budeme přesměrovávat, je zapotřebí jména reálných sloupců znát. Pro úspěšné navázání dále demonstrované aplikace se serverovou databází je nutné mít v dané tabulce(s názvem TelSeznam) tyto sloupce o těchto typech:Jmeno(varchar), Adresa(varchar), TelCislo(varchar), Poznamka(varchar),Obrazek(Blob).

8.3.2 Nástin praktické funkčnosti demonstrovaného protokolu

Abychom lépe pochopili smysl všech užitých prvků, bude nejlépe, když si celý systém demonstrujeme na nějaké praktické ukázce. Tedy naše serverová databáze bude představovat jednoduchý telefonní seznam (libovolně rozměrný). Každý jeden záznam tabulky bude představovat jednoho "telefonistu". O telefonistovi budou uvedené tyto údaje:Jméno(1.sloupec), Adresa(2.sloupec), Telefonní číslo(3.sloupec), Poznámka(4.sloupec). Dále může být ke zmíněnému kontaktu připojen obrázek,představovaný binárními daty uloženými v tabulce(typ BLOB). Všechny tyto údaje budou považovány přenosovým protokolem za dynamické, tedy mohou mít teoreticky nelimitovanou délku(ovšem v praxi by bylo dobré jistou max. hranici nastavit).

8.3.3 Autorizační část

Dříve než začne jakákoli věrohodná komunikace s klientem, je nejprve třeba provést autorizaci.Tedy zcela správně nejprve musíme provést vytvoření samotného spojení.

Ovšem tyto techniky závisejí až na zvolených metodikách síťové komunikace, takže o nich bude pojednáno ve zvláštní kapitole. Nyní se budeme soustředit pouze na přenosový protokol.

Tedy první co se stane po fyzickém navázání spojení je to, že server vyzve klienta(stavovým bytem 127) ke sdělení základních technických informací o sobě. Tato část je zde zejména proto, aby bylo možné rozpoznat typ klienta. Mobilní zařízení,či PC klient (příp.jsou možné další typy klientů). Dále je zde prostor pro uvedení dalších nepovinných informací o verzi klientské aplikace atd. Toto považuji vhodné zařadit z důvodu diagnostiky , abychom věděli jaké verze konkrétních klientů se nám připojují.V našem případě prozatím stačí jen základní rozlišení PC/Mobilní klient.Ovšem s dalším možným rozvojem takového řešení se může stát, že tuto informaci budeme požadovat a po té bychom museli měnit naše komunikační schéma (neboť by nebylo po změně kompatibilní).

Tedy s touto kompletní znalostí můžeme v průběhu dalšího vývoje jednoznačně zařadit příslušného klienta a příp. s ním komunikovat podle pro něj definovaného schématu.

Dále server pošle kontrolní znak(s hodnotou 101) , že informace přijal a že očekává autorizační údaje. V případě že kontrolní znak od serveru vůbec nepřijde, je zcela jasné že základní komunikace neproběhla správně. Klient by měl na tento stav zareagovat timeoutem a připojení ukončit. Ovšem tento scénář je za standardních okolností málo pravděpodobný.

Po přijetí kontrolního znaku klientská aplikace ví, že vše probíhá správně a zasílá serveru svoje logovací údaje.Server ověří jejích platnost a následně navrátí stavový byte s informací o úspěšnosti.Bytová hodnota 32 označuje stav READY,tedy autorizace proběhla a server očekává další požadavky od klienta.Naopak hodnota 0 označuje stav DENIED,kdy dojde k ukončení komunikace ze strany serveru, uzavře se i fyzické připojení. Klient samozřejmě může akci provést celou znovu (od prvního řádku), např. s jiným loginem.

	S/C	Význam bloku dat	Velikost	Hodnoty
1	S	Připojen	1	127
2	C	TypKlienta(PC/Mobil)	1	PC-1/M-2
3	C	Řetězec informací o klientovi	30	—
4	S	Výzva k autorizaci	1	101
5	C	Velikost uživatelského jména	4	—
6	C	Uživatelské jméno	Dyn	—
7	C	Velikost hesla	4	—
8	C	Heslo	Dyn	—
9	S	Úspěšnost autorizace(READY)	1	OK-32/Denied-0

8.3.4 Dotazovací část

Do této části se můžeme dostat až po předchozí úspěšné autorizaci.Pokud tedy obdržel klient od serveru v autorizační části stav READY, může položit serveru požadavek k nalezení dané položky.

Klient tedy musí nejprve přepnout server do modu SELECT. Pošle tedy příslušný stavový byte (s hodnotou 1).Aby se zabránilo zbytečným prostojům při dotazech a také zbytečnému narůstání síťové režie, je zbytečné protokol vystavovat tak, aby klient čekal na další potvrzení této akce. Jednoduše řečeno klient zašle najednou celý svůj požadavek se všemi informacemi tak, aby server po jeho obdržení mohl rovnou provést jeho vyhledání.

Prozatím server obdržel požadavek pro přepnutí do módu SELECT.Následuje přenesení řetězce s dotazem. Dále klient zasílá informaci, zda-li bude chtít přijat BLOB (Binary large object) přidružený k danému řádku v databázi. V našem případě bude BLOB představován bitmapou. Samozřejmě není nutné, aby byl BLOB přidružen ke každému záznamu. Tím pádem ale může nastat okamžik kdy sice požadujeme návrat BLOBu , ale on v databázi není. S těmito všemi případy se musí náš přenosový protokol vyrovnat.

Jsme tedy ve fázi, kdy server zpracovává klientův požadavek. V této chvíli klient pouze čeká. Pro úplnost server nyní přesměrovává svoje požadavky na databázový server. Po jejich zadání zase on sám čeká na zpřístupnění nalezených dat, která následně přeposílá klientovi.Možná, že to z výše popsaného vyznívá jako nějaký velký prostoj v komunikaci. Ovšem musíme si uvědomit, že s dnešními optimalizovanými databázemi (zejména pro vyhledávání) se celá tato chvíle, při nezatíženém serveru, vejde do jednotek milisekund.

Server tedy úspěšně vyhodnotil dotaz klienta. Posílá klientovi kontrolní znak (o hodnotě 3). Dále vrací počet vyhledaných záznamů. Protokol je obecný, takže server takhle může vrátit 0 až n záznamů. Ovšem v rámci zjednodušení a v rámci demonstrace budeme v dalším programovém ztvárnění počítat pouze s případem, kdy server vrací počet záznamů 0,nebo 1. Považujeme-li totiž sloupec jméno zároveň za primární klíč databáze, pak nám toto zjednodušení nevadí, neboť nemůže nastat případ, kdy dojde k navrácení většího počtu záznamů.

Nyní může nastat několik případů. Server takovou položku nenalezl. Tím pádem klientovi žádnou datovou oblast posílat nebude. V tabulce je tato datová oblast označena dvojitými čárami.Ovšem tento stav je naprosto korektní a tak po té, co klientovi pošle "počet nalezených záznamu" je roven 0, rovnou skočí na stav READY(řádek

29) a je připraven od klienta dostat jakýkoli další požadavek.Jinými slovy přeskočí oblast řádků 18-28. V případě dalšího požadavku se opět postupuje od řádku 11.

Pokud je záznam nalezen, posílá se datová oblast.Datová oblast se skládá vždy z délky následného stringu(4 bytová číselná proměnná), a pak se zasílá samotný string(o takto definované délce).V našem případě máme 4 stringové sloupce v tabulce, tedy se tato operace provede celkem 4x.Po té zasílá server kontrolní znak(s významem konec textové části).

Dále nastává větvení jednak podle předešlého požadavku klienta, kdy uvedl zdali zaslat BLOB(v našem případě obrázek), či nikoli.Tedy v případě že klient obrázek nechce, server pouze zašle velikost obrázku(standardně nula) a bez jakéhokoli rozhodování končí blok stavem READY.(tedy přeskakuje řádek 28).V případě že je požadavek na zaslání BLOBU (obrázku), ovšem databáze žádný obrázek nemá, pak se provede naprosto stejná operace. Velikost obrázku se zašle nulová a řádek 28 je opět přeskočen.Ve zbylém případě, kdy je požadavek na zaslání a obrázek je přidružen k řádku tabulky, je na řádku 27 skutečně zaslána velikost reálného binárního objektu a následně je (viz řádek 28) tento objekt přenesen.Server pak opět přechází do stavu READY a očekává další požadavky.

	S/C	Význam bloku dat	Velikost	Hodnoty
11	C	Přepnutí serveru do stavu SELECT	1	1
12	C	Délka dotazu(řetězce)	4	—
13	C	Samotný dotaz	Dyn	—
14	C	Zaslat BLOB přiřazený k záznamu	1	Ano-127/Ne-0
15	-	Server zpracovává požadavek	—	—
16	S	Vyhledání proběhlo	1	K-3
17	S	Počet vyhledaných záznamů	4	—
18	S	Velikost 1.sloupce	4	—
19	S	1.Sloupec (Jmeno)	Dyn	—
20	S	Velikost 2.sloupce	4	—
21	S	2.Sloupec (Adresa)	Dyn	—
22	S	Velikost 3.sloupce	4	—
23	S	3.Sloupec (TelCislo)	Dyn	—
24	S	Velikost 4.sloupce	4	—
25	S	4.Sloupec (Poznámka)	Dyn	—
26	S	Konec textové části	1	K-3
27	S	Velikost Blob(není v databázi=0)	4	—
28	S	Zaslán Blob	Dyn	—
29	S	Server READY	1	32

8.3.5 Přidání nového záznamu

Přidání nového záznamu bude již podobné dříve zmíněným principům. Samozřejmostí je, že je také zpřístupněno až po počáteční autorizaci.Klient tedy opět zašle požadavek na přepnutí serveru do patřičného stavu (INSERT). Následně zašle datovou strukturu. Nejprve tedy její textovou část(formát byl již zmíněn) a následně zasílá opět velikost BLOBu(obrázku).Nula znamená obrázek nebude přiložen ani přenesen, jiná velikost značí velikost binárních dat, která se následně přenesou(řádek 42 se provede).Nyní je drobný rozdíl v tom, že server musí sdělit svůj stav, nebo lépe úspěšnost provedení. A to udělá opět stavovým bytem, kdy zašle v případě úspěšného přidání bytovou hodnotu 127.V opačném případě je zaslána 0.Následně server tak jak tak přejde do stavu READY.

	S/C	Význam bloku dat	Velikost	Hodnoty
31	C	Přepnutí serveru do stavu INSERT	1	2
32	C	Velikost 1.sloupce	4	—
33	C	1.Sloupec (Jmeno)	Dyn	—
34	C	Velikost 2.sloupce	4	—
35	C	2.Sloupec (Adresa)	Dyn	—
36	C	Velikost 3.sloupce	4	—
37	C	3.Sloupec (TelCislo)	Dyn	—
38	C	Velikost 4.sloupce	4	—
39	C	4.Sloupec (Poznámka)	Dyn	—
40	C	Konec textové části	1	K-3
41	C	Velikost Blob(není=0)	4	—
42	C	Zaslán Blob	Dyn	—
43	S	Server přijal/Odmitnul	1	P-127/Od-0
44	S	Server READY	1	32

8.4 Serverová implementace soketového připojení

V předchozí kapitole jsme si demonstrovali informační část přenosu. Tedy již víme jaká data a kdy máme přenášet.Ovšem zatím nebylo zmíněno jak to technicky zařídit ze strany serveru. O tom bude právě pojednávat následující kapitola.

8.4.1 Požadované vlastnosti, výběr vhodných komponent

Nyní stojíme před prvním ze dvou úkolů o zprovoznění fyzické komunikace aplikačního serveru. Jak již bylo vysvětleno v předchozím textu, aplikační server je vlastně jakousi mezivrstvou a tak vlastně komunikuje hned na dvou stranách. Nyní se budeme zamýšlet nad tím, jaké parametry musí naše realizace splňovat pro komunikaci s klientem.

Typů připojení existuje velká řada. My musíme hledat takové, které nám zabezpečí jednak kompatibilitu se všemi klienty (tedy i s Javou) a také by mělo reflektovat navržený přenosový protokol.Proto zvolíme klasické soketové připojení, kdy bude serverová aplikace naslouchat na zvoleném portu a klient pak dostane přiřazen patřičný soket. Celá komunikace bude tedy jen pouhým "surovým" posíláním dat. Tedy nebudeme využívat žádných FTP,HTTP ani jiných serverů (kterými Borland Delphi také disponují).Srozumitelnost posílaných dat zajistí právě navržené komunikační schéma.

Víme tedy přibližně jaký typ připojení zvolit. Ovšem co očekáváme od kapacit takového připojení ? V nejjednodušším případě existují blokující soketová připojení, kdy se v daný okamžik reálně komunikuje pouze s jedním klientem. Toto pro jednoduchý případ jistě stačí, ovšem v konkurenci moderních aplikací a požadavků na velký počet simultálních spojení tento jednoduchý princip neobstojí.Také by bylo velmi nežádoucím faktem, kdyby náš aplikační server brzdila zařízení s nízkou konektivitou. Např. pokud by se v daný timeslot zasílala jen data nějakému mobilnímu klientu přes GPRS, pak by mohlo v tento okamžik docházet k nevytížení celkové konektivity serveru.A to i přesto, že by ostatní klienti požadavky měli, ovšem byli by v pořadí.

Tento způsob je sice realizovatelný, ovšem pro aplikaci ve stylu aplikačního serveru naprosto nevhodný. Musíme tedy najít nějaký jiný způsob přístupu. Naprosto ideální je, pokud by naše řešení dokázalo pro každé klientské spojení vytvořit jedno vlákno a v tom odehrávat celou komunikaci s jedním klientem. Tedy pro připojených 10 klientů by existovalo 10 vláken, která by byla zpracovávána zcela nezávisle. Každý klient by si mohl stahovat data svoji libovolnou rychlostí, aniž by v daný moment výrazně snižoval možnou přenosovou kapacitu.Sníží ji pouze o to, co skutečně protáhne, neboť zbylou šířku připojení využijí další současně připojení klienti.Tento způsob je nejefektivnější pro zpracování.Nyní zbývá najít vhodné komponenty, pro takové řešení.Potěšující zprávou je, že takové komponenty v Delphi skutečně existují a proto se na ně zaměříme v další kapitole.

8.4.2 Indy components verze 10

Tyto komponenty jsou běžnou součástí nových Delphi 2007 a najdeme je v záložkách INDY servers,INDY I/O handlers a INDY clients. I starší verze DELPHI obsahovaly tento balík, ovšem v průběhu vývoje zaznamenaly tyto komponenty četných změn a proto spoustu zde zmíněných metodik nemusejí tyto starší verze podporovat.Ovšem tyto komponenty by měli být open-source a v tom případě není problém staré komponenty inovovat na požadovanou verzi.

Výhodou těchto komponent je možnost skutečně multivláknové zpracování klientských požadavků.[11]

Komponenta TIdTCPServer

Jedná se o ústřední komponentu, které definujeme základní prvky našeho připojení.Z nápovědy se Delphi se dočteme (překlad): Tato komponenta je navržena jako multivláknový TCP server.Umožňuje několikanásobná vlákna, která naslouchají požadavky klientů pro připojení,akceptují nová klientská připojení, zpracovávají další klientskou komunikaci... Minimálně jedno vlákno je vždy vytvořeno pro naslouchání.[11]

Nejvýznamnější vlastnosti(properities) této komponenty jsou:

DefaultPort - Udává číslo portu, na kterém bude server naslouchat.

MaxConnestions - maximální počet současných připojení

Active - Při hodnotě true začíná komponenta naslouchat(ovšem musí být definováno číslo portu)

IOHandler - Zde se tvoří návaznost na komponentu IOHandler, která nám zpřístupňuje samotný přenos dat(dle zvoleného typu komponenty)

Nejvýznamnější události(events) komponenty IdTCPServer:

OnExecute - Událost je volána při jakémkoli klientském požadavku na server.Z této události se rozbíhají veškeré reakce serveru.Výhodou je, že její volání je zcela nezávislé na aktuální komunikaci s více klienty, neboť komponenta automaticky pro tato volání vytváří nová vlákna. Přístup na daného klienta je potom velmi jednoduchý, neboť se na něj dostaneme přes navrácený TIdContext (viz příklad):

```
procedure TForm1.IdTCPServerExecute(AContext: TIdContext);
var velikost:integer;
begin
```

```
//Čtení dat od klienta
velikost:=AContext.Connection.IOHandler.ReadInteger;
//Poslání dat ke klientovi (v tomto případě byte s hodnotou 32)
AContext.Connection.IOHandler.Write(#32);
....
end;
```

Komponenta TIdServerIOHandlerStack

Je to podpora pro multi vláknový server. S její pomocí můžeme snadno posílat data klientům, či od nich data přijímat.[11].Dále si uvedeme nejvýznamnější metody. Jsou to: ReadInteger, ReadString, ReadBytes, ReadCHar...

Všechny tyto metody slouží pro načítání příchozích dat od klienta. Slůvko následující za READ určuje který že daný typ bude načítán. V programovém kódu se metody READ chovají tak, že dané vlákno je pozastaveno do té doby, než je přijat příslušný počet bytů definovaných konkrétní metodou read.Po přijetí se pokračuje dalším řádkem.

Dále metoda Write slouží k zápisu.Je obecná, takže využívá přetížení a tudíž do ní můžeme dávat zmíněné typy a tyto typy budou automaticky odeslány.

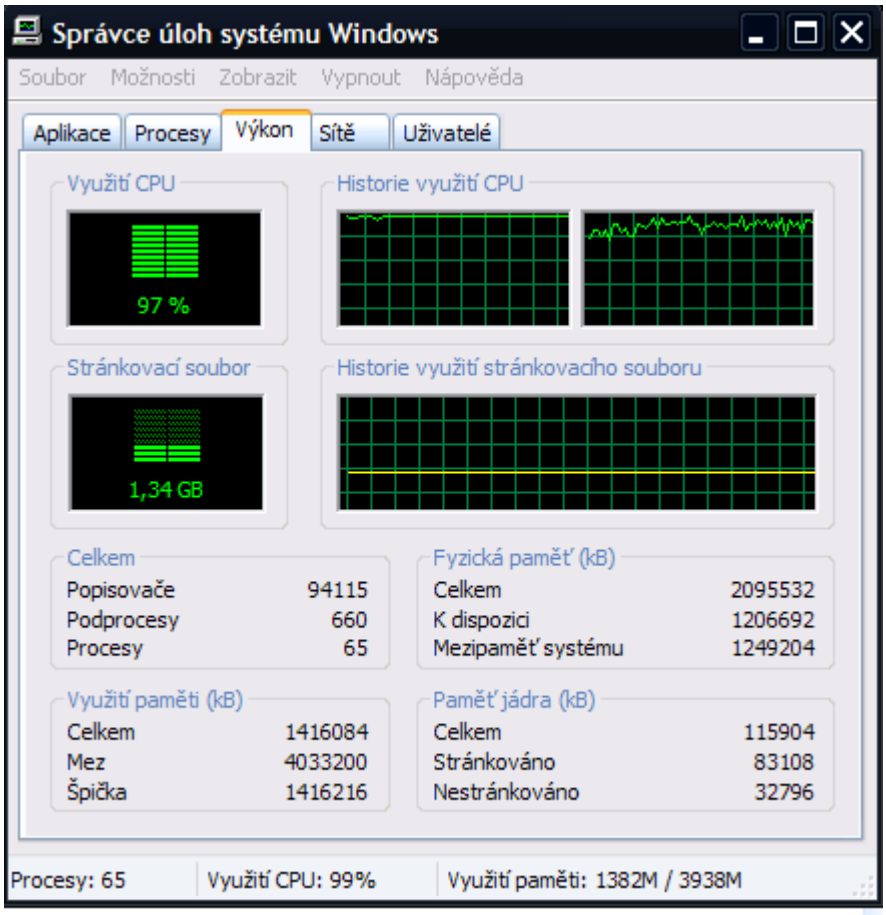
Nastínili jsme si použité funkce INDY Components, aby bylo možné pochopit základnímu programovému zpracování. Je tedy zřejmé, že jsem nezmínil všechny využité detaily, neboť by se tím zbytečně navyšoval obsah práce. Pro naprostou konkretizaci zmíněných faktů je vhodné se opřít o nápovědu Borland Delphi, kde jsou podrobně rozepsány přesné definice funkcí a vlastností, příp. další využitelné funkce.

8.4.3 Vícevláknové zpracování, testy rozložení zátěže na Dual-Core

Z předchozího textu vyplývá, že při použití těchto komponent máme velmi dobře zpracovánu komunikaci klient - aplikační server. Ovšem skýtá se zajímavá otázka. Pokud komponenty v sobě nativně podporují tvorbu samostatných vláken pro klientská připojení, může být této vlastnosti využito pro rozklad zátěže na více jádrový procesor ? V nápovědě Delphi se lze dočíst pouze o multivláknové implementaci, ovšem z toho ještě nelze jednoznačně usuzovat , jestli se zpracování požadavků rozloží na více jader.Abych dokázal na tuto otázku s určitostí odpovědět, provedl jsem malý test na svém stolním počítači s procesorem Intel DualCore 2,2Ghz. Komunikaci jsem ponechal běžet naprosto standardně, ovšem při každém dotazu jsem server

zatížil zbytečnou velmi náročnou operací(cyklické dělení s plovoucí desetinnou čárkou).Tuto operaci jsem samozřejmě připsal do programového kódu v metodě IdTCPServerExecute.Výsledek byl potěšující. Při jednom klientovi s cyklickým dotazováním působil server plnou zátěž na jedno jádro.Druhé bylo takřka bez zátěže(pouze systémové procesy). Při připojení druhého klienta došlo ihned k zatížení druhého jádra. Procesor tedy skutečně pracoval na 100procent.

Zmíněné měření zátěže bylo provedeno pomocí správce úloh.



Obr. 8.1: Test více vláknových operací aplikačního serveru a jejich rozložení na procesoru Dualcore

Závěrem musím říci, že se tedy jedná o plnohodnotná vlákna a pokud by byl použit na tento aplikační server zvlášť fyzický server s více jádrovým procesorem, pak bude jeho výkon tímto aplikačním serverem plně využit (v případě že bude třeba jej využít).

8.4.4 Zátěžový test soketové komunikace, závěr

V posledním bodě tohoto návrhu jsem se ještě rozhodl, že provedu další test (dokazující stabilitu připojení). V této době ještě nebyl aplikační server navázán na databázi a proto stále posílal stejná data.Jenže šlo pouze o otestování komunikace client - aplikační server, takže to bylo i žádoucí. Tentokráte jsem již odstranil zbytečnou zátěž v podobě cyklického dělení.Celý test spočíval v tom, že jsem postupně připojil velké množství klientů. Jednak přes local(127.0.0.1), a dále na dalším počítači, který byl připojen přes 100Mbit síť. Všichni klienti se cyklicky dotazovali serveru, tedy každý klient neustále generoval síťový provoz. Navíc jsem některým klientům zadal stahovat BLOBy z databáze a zbylým nikoliv,aby se výrazně navýšilo zatížení sítě a také aby se navýšila různorodost síťové komunikace.

Všem klientům byly korektně dotazy zodpovídány po celou dobu testu (10minut).Klienti mohli přerušit spojení, či vytvořit nové, bez jakéhokoli poškození jiného existujícího spojení.

Konkrétnější záznamy testu neuveřejňuji, neboť nepojednávají o uceleném chování navrhované aplikace. Navíc jejich úloha spočívala pouze k praktickému ověření, jestli dané komponenty skutečně zvládnou tuto náročnou funkci nutnou pro aplikační server.Konkrétní test s naměřenými hodnotami bude uveřejněn až v závěru práce, neboť bude hodnotit až celkové fungování zpracovaného projektu.

Nyní tedy z testu vyplývá, že máme velmi dobře zpracovánu komunikační část aplikační server - klient a můžeme se s chutí pustit do dalších implementací potřebných pro náš aplikační server.

9 TVORBA KLIENTSKÝCH APLIKACÍ

9.1 Implementace soketového připojení - Klient v Delphi

Jelikož realizace klienta je prováděna ve stejném vývojovém prostředí jako server, tak nám nyní musí být jasné, že ke většině potřebných informací jsme se již dostali při realizaci samotného serveru. Navíc klientská aplikace je po komunikační stránce mnohem jednodušší než-li serverová. Proto si krátce zmíníme odlišnosti tohoto zpracování.

9.1.1 Rozdílné komponenty

Jediný rozdíl v komponentách je ten, že nyní místo komponenty TIdTCPServer použijeme TIdTCPClient. Jelikož se jedná jen o obyčejného klienta, tak v základním zpracování nebude třeba využívat žádné z jeho událostí.Pouze bude zapotřebí správně nastavit jeho vlastnosti(properities). Ty nejzásadnější vlastnosti jsou PORT a HOST.Port je opět číslo portu a musí se shodovat s číslem portu nastaveném na serveru.HOST slouží pro zadání IP adresy serveru.

Další komponenta TIdServerIOHandlerStack zůstává bez změny. Taktéž budeme využívat stejné metody pro zápis a čtení(Write,ReadInteger...). Ovšem k jistě změně tu dochází, nyní veškeré požadavky nebudeme směřovat na TIdContext, který nám vracela serverová verze komponenty v metodě IdTCPServerExecute.Důvod je prostý.Jednak zde není událost execute a navíc tu není více vláken, tedy není třeba rozlišovat jednotlivé typy připojení.

Implementaci si tedy ukážeme na malém příkladu

```
//Zápis dat na server(zapsán byte s hodnotou 1)
IdTCPClient.IOHandler.Write(#1);
//Čtení dat ze serveru do proměnné Cislo
Cislo:=IdTCPClient.IOHandler.ReadInteger;
```

Jak je vidět, zde se na příslušný IOHandler dostáváme přímo z patřičného klientského připojení. V případě serveru je metodika ve skutečnosti podobná, ovšem je maskovaná v komplikovanější funkci multivláknového zpracování a volání pomocí událostí. Tedy v případě serveru je tento způsob implementace nevyhovující.

9.1.2 Rozdílné chování klienta

Zde si musíme uvědomit především ten fakt, že celý klient standardně funguje již pouze v rámci jednoho vlákna. Tedy např. u serveru platí následující tvrzení: Funkce READINTEGER(či kterákoli jiná funkce READxxx)zastaví příslušné vlákno, dokud mu nejsou poslána očekávaná data, bez ovlivnění ostatních probíhajících komunikací.Toto je v obecné rovině pravda i u klienta, s tím rozdílem, že zde je vlákno jediné. Navíc jsou v tomhle vlákně standardně zpracovávány i všechny ostatní systémové zprávy této aplikace. Jinými slovy v tomto bodě lidově řečeno celá aplikace zatuhne do doby, než dostane očekávaná data.S tímto chováním je třeba počítat a řešit jej např. zavedením timeoutu, po jehož vypršení se již na data nečeká a provedou se všechna systémová volání aplikace. V případě že byl timeout nastaven na nízkou hodnotu, mělo by být možné se na data pro kontrolu znovu dotázat, neboť fyzické připojení bylo stále zachováno.V případě vysoké hodnoty timeoutu lze předpokládat, že data již zřejmě nedojdou a je lepší zahájit komunikaci celou znovu.

9.2 Implementace soketového připojení - Klient v J2ME

Zmíněné soketové připojení jsme mimo jiné zavedli kvůli kompatibilitě s mobilní platformou - javou. Nyní si přiblížíme použité techniky pro zprovoznění této komunikace. Jelikož se zde již jedná o úplně jinou platformu, budou se následující principy lišit od dříve zmíněných.Jistou výhodou javy Microedition je, že neobsahuje tolik různých možností pro implementaci stejných věcí(na druhou stranu některé chybí). Nicméně nyní nám tato vlastnost usnadní práci, protože nemusíme vybírat z nepřeberného množství dostupných komponent.

9.2.1 SocketConnection

SocketConnection je v javě základní interface určené pro soketovou komunikaci.S jeho pomocí snadno vytvoříme základní část připojení k serveru.Nejprve tedy musíme vytvořit zmíněnou instanci tohoto interface.Zároveň s tím i otevřeme toto fyzické připojení. K tomuto úkonu využijeme třídy Connector a její statické metody Open. Jako parametr metody open zadáme řetězec, který v sobě bude obsahovat typ připojení (socket), danou IP adresu a po dvojtečce se uvede PORT pro komunikaci.[3] Implementace příkazu vypadá potom následovně.

```
//deklarace interface
SocketConnection socConNet = null;
```

```
//otevření spojení
socConNet = (SocketConnection) Connector.open("socket://81.xxx.xxx.xxx:110");
```

9.2.2 DataInputStream, DataOutputStream

Tyto třídy nám umožní již konkrétní bytový přenos. Pomocí nich si totiž vytvoříme konkretní příchozí, či odchozí stream.

```
// deklarace Streamů
DataInputStream inStrNet = null;
DataOutputStream outStrNet = null;
// otevření jednotlivých streamů
inStrNet = socConNet.openDataInputStream();
outStrNet = socConNet.openDataOutputStream();
```

9.2.3 Zápis a čtení dat(streamů)

Podobně jako jsme v Delphi navazovali metody Read,nebo Write na IOHandler, zde budeme místo něj používat zmíněný DataInputStream, nebo DataOutputStream.V javě budeme muset také správně přiřadit použité metody k jednotlivým streamům. Tedy u DataInputStream používat metodu Read a v případě DataOutputStream užít metody Write.

Dále si musíme říci, že podobně jako v Delphi probíhalo bufferování odchozích dat, se děje i v javě. Proto musíme vždy nahromaděná data na konci klientského bloku z bufferu vyprázdnit příkazem flush. Tento příkaz je samozřejmě vázán na DataOutputStream.

Stejně jako v Delphi, i tady můžeme využít předdefinovaných typů k odeslání či přijetí. Slouží k tomu metody[3]: writeByte,writeInt,writeChars,readByte... Případně pro větší úseky dat (např pro binární data BLOB) můžeme využít metodu readFully, které zadefinujeme kolik bytu se má z příchozího streamu načíst do našeho programového bufferu.

Použití metod pak může vypadat následovně:

```
// odeslani uzivatelskeho jmena
outStrNet.writeInt(userName.length()*2);
outStrNet.writeChars(userName);
// vyprázdnění bufferu
outStrNet.flush();
.....
```

```
// nacteni řetězce
velikostPoslanychDat=inStrNet.readInt();
strBuf=new StringBuffer();
for (int i=0; i¡velikostPoslanychDat/2;i++)
strBuf.append( inStrNet.readChar() );
```

Ještě si dovolím podotknout, že všechny tyto metody by měly být uzavřené do TRY aby se u nich daly odchytávat výjimky(např. při ukončení spojení...) Jinak by také mohlo dojít k pádu celé aplikace.

9.2.4 Závěr tvorby java klienta

Samotná komunikace java klienta není příliš obtížná. Navíc vše zjednodušuje fakt, že na java klienta jsou kladeny mnohem menší nároky než-li na server. O něco problematičtější již může být začlenění těchto komunikačních principů do nějakého rozumného grafického výstupu. Ovšem tato problematika je značně rozsáhlá a navíc není náplní této práce.Proto se jí nebudeme zabývat.

10 APLIKAČNÍ SERVER V DELPHI 2007 - KOMUNIKACE S DATABÁZÍ

10.1 Volba SQL databáze

O databázových systémech v tomto projektu již padlo nemálo slov. Jelikož se celou dobu bavíme o zprostředkování dat z SQL databáze, měli bychom si ve stručnosti alespoň něco vyhledat o existujících databázích.

10.1.1 PostgreSQL

PostgreSQL je plnohodnotným relačním databázovým systémem s otevřeným zdrojovým kódem. Má za sebou více než patnáct let aktivního vývoje a má vynikající pověst pro svou spolehlivost a bezpečnost. Běží na všech rozšířených operačních systémech včetně Linuxu, UNIXů a Windows. Stoprocentně splňuje podmínky ACID, plně podporuje cizí klíče, operace JOIN, pohledy, spouště a uložené procedury. Obsahuje většinu SQL92 a SQL99 datových typů, např. INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL a TIMESTAMP. K systému existuje kvalitní volně dostupná dokumentace včetně českých překladů FAQ a FAQ pro o.s. fy. Microsoft. Výkonnostně nezaostává za srovnatelnými komerčními systémy a v častokrát je i předčí.[6]

PostgreSQL umožňuje běh uložených procedur napsaných v několika programovacích jazycích, v Perlu, v Pyhon, v jazyku C nebo v speciálním PL/pgSQL, jazyku vycházejícím z PL/SQL fy. Oracle. Existují PostgreSQL varianty JDBC, ODBC, dbExpress, Open Office, PHP, .NET Perl nativních rozhraní. K PostgreSQL existuje překladač Embedded SQL pro C a C++. Dále existuje experimentální podpora SQL/PSM - standardizovaného jazyka pro návrh uložených procedur v ANSI SQL.

Předností systému PostgreSQL je rozšiřitelnost. Systém může být bezproblémově rozšiřován o nové datové typy, funkce operátory, agregační funkce, procedurální jazyky.

10.1.2 MySQL

MySQL je databázový systém, vytvořený švédskou firmou MySQL AB. Jeho hlavními autory jsou Michael Monty Widenius a David Axmark. Je považován za úspěšného průkopníka dvojího licencování – je k dispozici jak pod bezplatnou licencí GPL, tak pod komerční placenou licencí.

MySQL je multiplatformní databáze. Komunikace s ní probíhá – jak už název napovídá – pomocí jazyka SQL. Podobně jako u ostatních SQL databází se jedná o dialekt tohoto jazyka s některými rozšířeními.

Pro svou snadnou implementovatelnost (lze jej instalovat na Linux, MS Windows, ale i další operační systémy), výkon a především díky tomu, že se jedná o volně šiřitelný software, má vysoký podíl na v současné době používaných databázích. Velmi oblíbená a často nasazovaná je kombinace MySQL, PHP a Apache jako základní software webového serveru.

MySQL bylo od počátku optimalizováno především na rychlost, a to i za cenu některých zjednodušení: má jen jednoduché způsoby zálohování, a až donedávna nepodporovalo pohledy, triggery, a uložené procedury. Tyto vlastnosti jsou doplňovány teprve v posledních letech, kdy začaly nejčastějším uživatelům produktu – programátorům webových stránek – již poněkud scházet.[7]

10.1.3 Firebird

Firebird je odvozen ze zdrojového kódu Borland InterBase 6.0. Je open source a nemá žádné dvojité licence. Dá se použít pro komerční účely nebo pro open source aplikaci, neboť je absolutně zdarma. Firebird je úplný databázový stroj. Může obsluhovat databáze od několika mála KB až po gigabajty s dobrým výkonem a skoro bez údržby.

Seznam základních rysů firebirdu:

Plná podpora uložených procedur,multigenerační architektura,po instalaci není třeba žádná konfigurace,přímá podpora pro všechny hlavní operační systémy,velké množství způsobů jak přistupovat k databázi(nativní/API, dbExpress ovladače, ODBC, OLEDB, .NET provider, JDBC nativní typ 4 ovladač, Python modul, PHP, Perl atd.) [8]

10.1.4 Další databázové systémy

Všechny tři zmíněné databázové systémy mají vždy nějaké plné vydání ve variantě open-source. Ovšem nejsou to jediné databázové systémy se kterými se můžeme setkat. Mezi další známé databáze patří např MS SQL z produkce Microsoftu. Ovšem tato databáze je již komerční. Volně dostupná je pouze její velmi odlehčená varianta. Podobně je to i s databází Oracle, či s databázovým serverem od IBM.

10.1.5 Shrunutí, zvolení vhodné databáze

Máme tedy databáze komerční a nekomerční. Z důvodu existence kvalitních open-source databází se zaměříme zejména na ně.

Nejprve si ovědomme skutečné nasazení,či využití našeho databázového serveru. Po databázi budeme především chtít velmi rychlé vyhledávání. Navíc lze tvrdit, že naše požadavky budou velmi jednoduché. Defakto si vystačíme s vyhledáváním podle indexovaného sloupce. Mám namysli to, že u sloupce by mělo být definováno, že se podle něj hledají záznamy v databázi. Databáze tak bude pracovat podstatně rychleji. Tyto naprosto základní prvky nám umožňuje drtivá většina databází.

Proto jsem se rozhodl při svém výběru preferovat elegantnost, jednoduchost a rychlost před robustností. Tím snad nechci říci, že by např.PostreSQL byla špatnou databází, či databází nevyhovující pro naše řešení. Již z dříve zmíněného plyne, že PostreSQL je velmi kvalitní a velmi univerzální databází. Nicméně my bychom její funkce nyní vůbec nevyužili a proto bude lepší se zaměřit na nějaké jednodušší databáze.

Mezi takové elegantní databáze patří My-SQL a také Firebird, jehož název je trošku novinkou, ale jinak se jedná o skutečně osvědčenou technologii(Interbase).Pro Firebird nám velmi nahrává ještě jedna okolnost, že tato databáze vznikla na základě databáze Interbase. Uvědomíme-li si, že DELPHI mají velmi dobrou podporu pro komunikaci s touto mateřskou databází(neboť je produktem Borlandu), tak je to jistě silný argument pro její použití.

To ovšem neznamená, že bychom nemohli využít např. My-SQL. Proto se v dalším řešením zaměřím na tyto dva zmíněné databázové systémy.

10.1.6 Základní datové typy SQL databáze

Zmíníme si některé základní datové typy SQL databáze, s nimiž bychom se mohli při řešení setkat.[8]

Celočíselné typy: SMALLINT,INTEGER,BIGINT
Desetinný typ: FLOAT
Řetězec(ekvivalent Stringu): VARCHAR
Typy data a času: Date, Time
Binární typ: BLOB (Binary Large Object)

Z popsaných typů využijeme především varchar, pro textové řetězce. V našem ukázkovém řešení se nám bude hodit pro zmíněný záznam daného kontaktu(Jméno, Adresa, TelCislo...) . Dále využijeme binární objekt BLOB pro možnost uložení bitmapy k danému řádku tabulky.

Ještě podotknu že s binárním typem blob je jinak nakládáno než-li z ostatními typy.Proto je skutečně optimalizován pro ukládání i velkých dat (např.audio,video souborů). Binární typ blob není fyzicky součástí daného řádku. V daném řádku je

pouze ukazatel na tato data. Proto vložené rozměrné binární struktury dále nezatěžují např.vyhledávací proces.

10.2 Připojení na databázi v Delphi

Již jsem pověděl, že narozdíl od J2ME je Delphi velmi robustní nástroj. Což znamená, že stojíme před nelehkým úkolem. Musíme totiž vybrat nejlepší komponenty pro práci s danou databází. Vezmeme-li v úvahu ten fakt, že přes naše komponenty můžou být připojeny stovky klientů do databáze a že s ní mohou simultáně komunikovat, bude třeba tyto komponenty volit velmi uvážlivě. Rovněž nebude na škodu, porovnáme-li jednotlivé komponenty v praxi a zamyslíme-li se nad jejich efektivitou.

Pro přesnost budeme uvažovat jen ty komponenty které nám přímo nabízí Delphi 2007.

10.2.1 BDE

BDE (Borland Database Engine). Tato databázová "klasika" je sice už řadu let předmětem pochybovačných poznámek týkajících se kvality a výkonu, stále má v Delphi silnou pozici. A to i přesto, že byl oficiálně ohlášen konec podpory BDE, takže se již nedočkáme žádných inovací a vylepšení. BDE je však stále výhodným řešením pro ty, kteří potřebují pracovat s lokálními databázemi, neboť má přímý přístup k databázovým souborů Paradoxu, Accesu a dalších (pravda, starších) systémů řízení báze dat. BDE obsahuje snad nejširší paletu podpůrných nástrojů a funkcí pro práci s nejrůznějšími databázemi. Pokud nám nestačí možnosti, které obsahuje přímo BDE, je možné využít propojení s databázemi prostřednictvím ODBC, čímž sice ztratíme na výkonu, na druhou stranu však získáme možnost pracovat snad se všemi databázemi. Používání BDE je relativně jednoduché, ovšem na druhou stranu je trochu obtížnější šíření a instalace aplikací postavených na BDE: zákazníkův počítač musí obsahovat správně nakonfigurované BDE. Komponenty pro BDE se v Delphi vyskytují na stránce "BDE" palety komponent.[9]

Z toho plyne, že BDE je mocný nástroj a při použití ODBC je velmi univerzální. Jen na okraj dodám, že použití ODBC bylo již demonstrováno v úvodních kapitolách. Ovšem zde je univerzálnost vykoupena na úkor výkonu, jelikož je vše převáděno přes ovladač ODBC. Navíc BDE je skutečně zastaralou platformou a zdá se, že je určeno zejména pro klientskou práci s lokální databází. Proto nebude vhodné,abychom využili BDE za základní stavební prvek naší komunikace s databází. Ovšem pokud bychom někdy potřebovali zpřístupnit nějakou nepodporovanou

serverovou databázi, či ji převést pomocí našeho aplikačního serveru, tak možnost máme právě pomocí BDE.

10.2.2 dbExpress

Citace z "Umíme to s Delphi"

Tato technologie je nejnovější a v určitém směru také nejvíce doporučovanou technologií (či spíše mechanismem, neboť dbExpress není řádným databázovým strojem, ale pouze mechanismem přístupu k více druhům databází). Její ohromnou výhodou je rychlost a efektivita. Ta je způsobena především tím, že pracuje pouze s jednosměrnými datovými sadami, dále neukládá záznamy do vyrovnávací paměti, negeneruje žádné interní dotazy apod. Na druhou stranu není přímo možné pracovat s komponentami pro editaci databázových údajů, neboť dbExpress (vzhledem k uvedeným vlastnostem) např. neumožňuje přejít na předchozí záznam. Hodí se tak nejvíce, když potřebujete získat data z databáze, ale nepotřebujete je modifikovat (např. do HTML reportu). Ale i v ostatních případech je možné si "vypomoci" použitím dalších komponent, které vytvoří mechanismus pro "normální" práci s daty (např. komponenta ClientDataSet). Použití dbExpress nevyžaduje (na rozdíl od BDE) žádné dodatečné instalace u zákazníka a navíc umožňuje přenositelnost na Linux, neboť dbExpress je k dispozici i v Kylixu. Příslušné komponenty jsou k nalezení v paletě komponent na stránce dbExpress. Použití dbExpress je v současnosti asi nejdoporučovanější ze všech možností, které v Delphi máme.[9]

Podíváme-li se na aktuální Delphi 2007, máme zde již v této komponentní sadě skutečně předchystánu řadu dostupných ovladačů pro jednotlivé databáze.Z nejvýznamnějších si uveďme My-Sql,Interbase(tedy i Firebird),MsSql... Proto tyto komponenty komponenty budou zahrnuty do dalšího řešení.

10.2.3 Interbase

Delphi obsahuje velmi propracovanou podporu databází InterBase (samozřejmě především proto, že tuto databázi donedávna firma sama vyvíjela). Prostřednictvím komponent z palety Interbase můžeme pracovat s databází InterBase skoro stejně efektivně, jako kdybychom si otevřeli přímo klienta tohoto databázového stroje. Interbase je v současnosti poměrně progresivní variantou, neboť Borland uvolnil zdrojové kódy, čehož se okamžitě chopila open-source komunita a (při zachování kompatibility) vytvořila databázovou platformu Firebird (http://www.firebirdsql.com/), která je velmi zajímavou (a též bezplatnou) alternativou k databázím typu MySQL pro vývoj webových aplikací, neboť na rozdíl od MySQL disponuje pokročilými možnostmi jako např. vnořené dotazy SQL nebo podpora transakčního zpracování. A

kromě toho je výborně podporována v Delphi.[9]

Znamená to tedy, že máme k dispozici ideální podporu pro firebird rovnou od výrobce vývojového prostředí. Této vlastnosti při řešení opět využijeme.

10.3 Připojení pomocí dbExpress

Nyní si demonstrujeme základní prvky toho, jakým způsobem funguje připojení pomoci dbExpress k databázi. Abychom byli schopni toto připojení v praxi reálně vytvořit či otestovat, je třeba mít na počítači nainstalované zmíněné databáze firebird, nebo MySql.

10.3.1 Komponenta TSQLConnection

Tato komponenta je základním prvkem pro připojení k databázi. Komponentě je třeba nastavit příslušný ovladač. To se provede pomocí jejich vlastností (properties) ConnectionName a DriverName. Pro zadání dalších parametrů přípojení, jako je hostname, ,login ... slouží vlastnost Params. Vzhledem k tomu že komponenta umožňuje připojení na několik druhů databází je tedy přesný obsah vlastnosti Params závislý na zvoleném ovladači. Komponenta si také přímo v object inspectoru říká o dané knihovny, které rovněž souvisí s výběrem databáze. Jednu ze zmíněných knihoven (vlastnost VendorLib) lze nalézt přímo v nainstalované databázi a zbylou přímo v instalaci Delphi(vlastnost LibraryName).

Pokud komponentu tvoříme až za běhu programu může potřebná definice vlastností vypadat následovně. Ukázka slouží pro propojení s databází Firebird.

```
sqlconnection.ConnectionName:='IBConnection';
sqlconnection.DriverName:='Interbase';
sqlconnection.LoginPrompt:=false;
sqlconnection.GetDriverFunc:='getSQLDriverINTERBASE';
sqlconnection.KeepConnection:=true;
sqlconnection.LibraryName:='dbxint30.dll';
sqlconnection.LoadParamsOnConnect:=false;
sqlconnection.VendorLib:='gds32.dll';
with sqlconnection.Params do
begin
Add('DriverName=Interbase');
Add('Database=DATA');
Add('RoleName=RoleName');
Add('User_Name=jorge');
```

```
Add('Password=jorge');
Add('ServerCharSet=');
Add('SQLDialect=3');
Add('ErrorResourceFile=');
Add('LocaleCode=0000');
Add('BlobSize=-1');
Add('CommitRetain=False');
Add('WaitOnLocks=True');
Add('Interbase TransIsolation=ReadCommited');
Add('Trim Char=False');
end;
```

Jen na okraj poznamenám, že ve jménu ovladače musí být skutečně položka Interbase, přestože se naše databáze jmenuje Firebird. Jak již bylo řečeno Firebird je rozšířením Interbase a je s ním zpětně kompatibilní.

10.3.2 Komponenta TSQLQuery

Tato komponenta slouží pro realizaci čtení či zápisu dat do databáze. K realizaci stačí využít vlastnost SQL, do které zadáme sql příkaz. Je-li to SQL příkaz pro vyhledání (návratový), např.Select, pak po naplnění vlastnosti SQL zavoláme metodu OPEN. V případě, že je to příkaz bez návratu, např INSERT, pro jeho provedení zavoláme metodu ExecSQL.

V případě zpřístupnění dat z tabulky (pomocí metody Open) můžeme s navrácenou hodnotou(tedy vyhledanými záznamy) přímo pracovat jako s konkrétními řádky tabulky. Např.získat hodnotu sloupce jméno na daném řádku lze následujícím způsobem:

```
Jmeno:=sqlquery.Fieldbyname('JMENO').aswidestring;
```

10.4 Připojení pomocí Interbase

Tyto komponenty již budou aplikovatelné pouze na jedinou z uvedených databází a tou je Firebird (samozřejmě lze jimi provést i propojení na komerční databázi INTERBASE). Jejich výhody pro práci s Firebird uvidíme v závěrečném porovnání Interbase X dbExpress. To je také důvod, proč jsem postupně při svém zpracování přešel na tyto komponenty.Nyní si stručně popíšeme rozdíly oproti dbExpress.

10.4.1 Základní komponenty

Skladba komponent vypadá podobně jako v dbExpress. Pro přípojení na databázi slouží komponenta TIBDatabase. V jejich vlastnostech je třeba nadefinovat cestu k databázovému souboru a login. Nově musíme při své realizaci použít komponentu TIBTransaction, pro kontrolu nad prováděnými transakcemi. Pro přístup dat lze využít již známou IBQuery(obdoba SQLQuery),nebo IBTable.

10.4.2 Implementace BLOBs

Jelikož náš přenosový protokol umožňuje přenášet ze serveru i na server rozměrný formát BLOB, bude třeba tento přenos uskutečnit i mezi aplikačním serverem a databázovým server. U typu BLOB ovšem nastává drobná odlišnost. Nelze k němu totiž efektivně přistupovat pomocí zmíněné dotazovací či zapisovací funkce Fieldbyname. Tato funkce totiž formát BLOB vůbec nepodporuje. Abychom se vyhnuli nějakým těžkopádným řešením bude nejlepší když zvolíme následující postup. Textová data z řádku lze tedy načíst obvyklým způsobem. V případě že nebudeme požadovat z databáze načíst BLOB (např. klient na něj nedal požadavek), nebudeme zbytečně zatěžovat komunikaci mezi aplikačním serverem a databází. Tedy jeho načítání přeskočíme. V případě, že jej skutečně budeme chtít klientovi poslat, jeho načtení z SQL databáze provedeme pomocí streamu.K tomuto slouží funkce CreateBlobStream. Funkci samozřejmě musíme nadefinovat příslušný sloupec, ve kterém se nachází BLOB a také ji uvést do stavu čtení.Potom ji na příslušném řádku tabulky zavoláme.

```
stream:=ibquery.CreateBlobStream(ibquery.Fieldbyname('OBRAZEK'),bmread);
```

V případě, že budeme požadovat zápis dat, opět vše vyřešíme pomocí streamu. Pouze při jeho vytváření uvedeme stav zápisu(bmwrite).

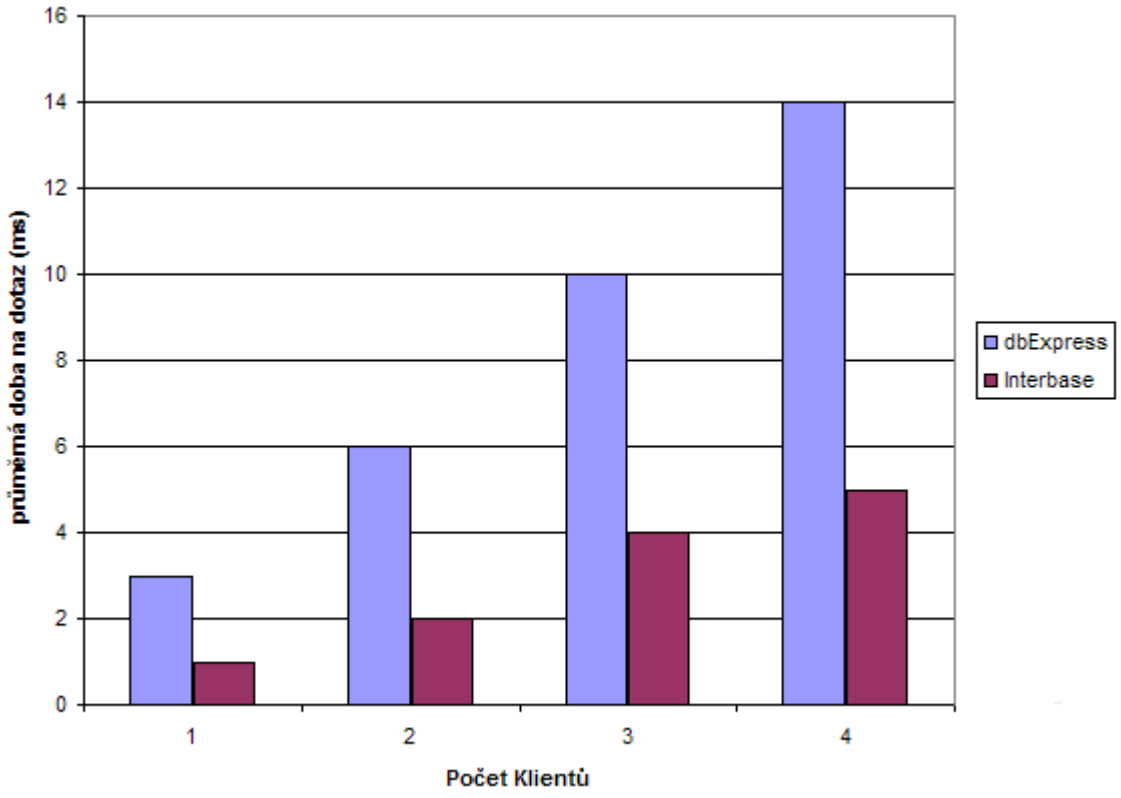
10.5 Výkonové srovnání dbExpress X Interbase

Nyní již známe spoustu možností, jakým způsobem v DELPHI realizovat připojení na databázi. Proto jsem provedl jednoduché měření demonstrující rychlost aplikačního serveru s jednotlivými komponentami. V tomto testu jsem využil Firebird databázi.Do databáze jsem umístil přibližně 1,5 milionu záznamů (některé i s binárními daty).Velikost databázového souboru se pohybovala kolem 45GBytů. Dále jsem vytvořil klienta, který prováděl neustálé náhodné dotazování.Z databáze byly vytahována pouze textová data, nikoli bloby. V klientovi bylo definováno 10000 tisíc

položek pro dotazování(stylem DOTAZ1 až DOTAZ10000). Všechny tyto položky během testu existovaly v databázi, tedy při každém kroku probíhalo vyhledání a navrácení dat.Tedy jakmile klientovi přišla odpověď, ihned poslal nový dotaz. Postupně jsem současně připojil několik klientů, kteří generovali současně zmíněný provoz.

Jelikož je vhodné hodnotit chování komponent pro komunikaci s databázovým serverem samostatně, uvádím délku zpracování dotazu z pozice aplikačního serveru.Přesněji do této doby je zahrnuta prodleva, kterou trvá položení dotazu z pozice aplikačního serveru databázi a jeho navrácení. Doba, kterou tato data putují k samotnému klientovy není zahrnuta do měření.

Konfigurace počítače na němž běžel při testu aplikační server a databáze Firebird:Intel DualCore 2,2Ghz 2GBRam (DDR2-800Mhz).



Obr. 10.1: Porovnání výkonu dbExpress a Interbase komponent

10.6 Závěr

Důvodem podstatně vyššího výkonu u Interbase komponent bude zřejmě ten fakt, že komponenty přistupují přímo do databáze a mají propracovánu velmi efektivní práci s databází, neboť byly vyvíjeny pro konkrétní jednu platformu. Oproti tomu

dbExpress je nástrojem obecnějším, přesto však také dobře propracovaným a použitelným. Závěr je tedy jednoznačný. Pro použití s databází Firebird budou nejlepší komponenty Interbase. Na nich jsem také vystavěl své koncové řešení.

11 APLIKAČNÍ SERVER V PRAXI - ZÁTĚŽOVÝ TEST

Nyní již máme vystavěnu celou základní funkčnost aplikačního serveru. Server je schopen přijímat dotazy od klienta, pokládat je databázi a výsledky navracet klientovi. Stejně tak dokáže přidávat nové položky. Pro testy a závěry tato funkčnost stačí. Ovšem není problém podle zmíněných faktu funkčnost diametrálně rozšířit. Taktéž bychom mohli do našeho aplikačního serveru začlenit více databází a provádět mezi nimi přenosy či jejich synchronizace...

Jeho chování si na závěr demonstrujeme v praxi s vyšším počtem klientů a uvedeme si příslušné časy jednotlivých vrstev při zpracování dotazů.

11.1 Úvahy nad realizací testu

Při testu jsem vycházel z předpokladu, že by bylo vhodné aplikační server s databází umístit na počítač s nižším výkonem. Naopak klienty umístit na PC s diametrálně vyšším výkonem. Důvodů bylo několik. Jednak jsem nechtěl dopustit,aby měření bylo znepřesněno přetížením počítače, na kterém jsou spuštěni klienti.Následně by pak také nemuseli dokázat plně vytížit (či přetížit) aplikační server. Další úvaha byla taková, že zvolený scénář lépe vystihuje skutečnou špičku, kdy požadavky klientů jsou v převisu nad serverem a server je musí být schopen simultalně zodpovídat.

11.2 Podmínky testu , konfigurace počítačů

11.2.1 SQL databáze

Pro test jsem zvolil databázi Firebird verze 2.1.Velikost databáze se pohybovala kolem 10 tis. řádků, velikost 100MB. Obsahovala řádky s Bloby i bez. Tabulky měli vždy 5 sloupců a byly po celou dobu testu stejné. Databáze byla umístěna na stejném počítači jako aplikační server.

11.2.2 Aplikační server

Aplikační server využíval ke komunikaci zmíněných IndyComponents(soketové připojení) a komponent Interbase. Vše z verze Borland Delphi 2007.

11.2.3 Klientské aplikace

Klientské aplikace byly uzpůsobeny ke generování cyklických dotazů. Tedy po zodpovězení předchozího dotazu generovaly dotaz nový. Byly nastaveny tak, aby generovaly dotazy na existující položky a aby byla vždy navrácena data z tabulky. Různorodost dotazů - 5 tisíc možností na zvolený typ dotazu. Jeden typ dotazů na řádky bez BLOBS, druhý na řádky s BLOBS. Dle zvoleného testu. Rozměry binárních dat se pohybovaly kolem 300KBytů a tato data byla různorodá (aby nezůstala v případné cache)

11.2.4 Konfigurace počítačů

Aplikační server a databáze Firebird: Notebook Intel Pentium 4-M 1,6Ghz, 512MBRam (DDR1- 400Mhz).

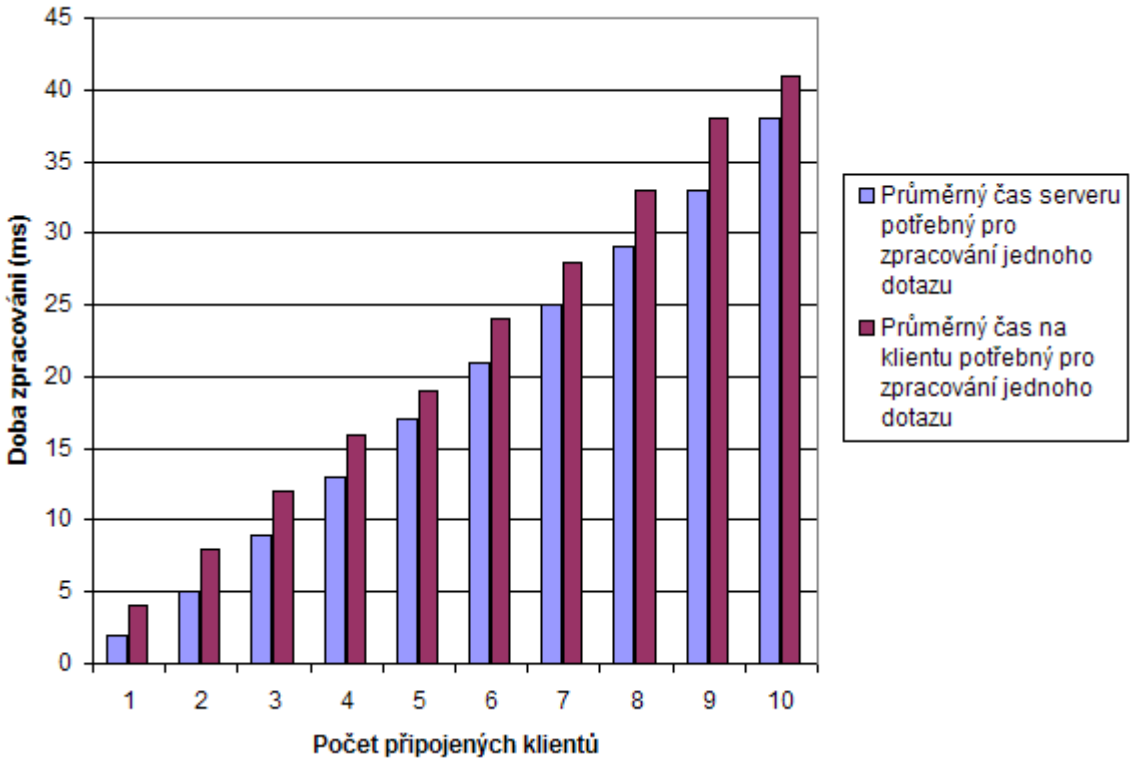
Umístění klientu: Intel DualCore 2,2Ghz, 2GBRam (DDR2- 800Mhz)

Síťová komunikace: 100Mbit/s

11.3 Průběh testu, výsledky

11.3.1 Dotazování bez požadavků na BLOBs

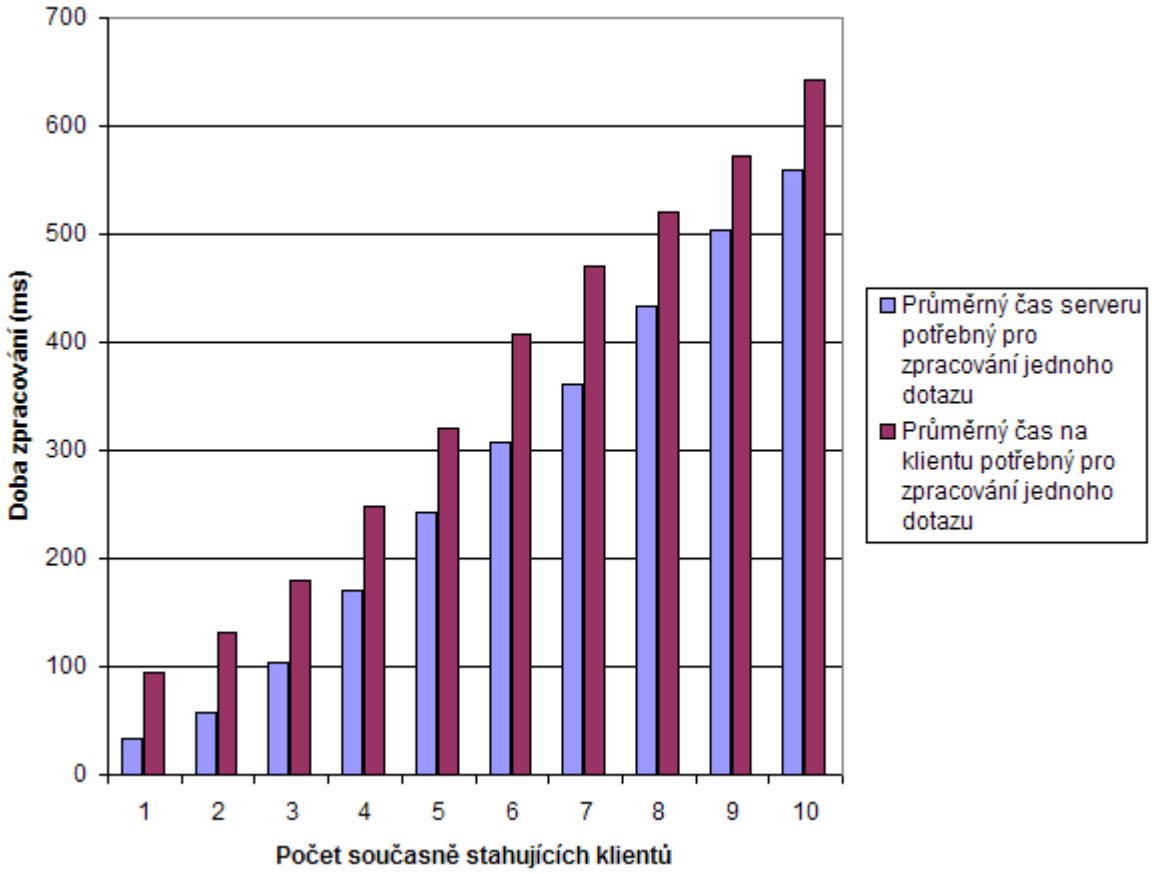
Test probíhal postupným navyšováním klientů v rozmezí od 1 do 10. Síťový provoz byl v tuto chvíli i při plném vytížení serveru velmi malý, neboť se sítí protahovala pouze velmi malá data.



Obr. 11.1: Test celého aplikačního serveru s databází.Klienti stahují pouze textová data(bez BLOBů)

11.3.2 Dotazování s BLOBs

Test probíhal postupným navyšováním klientů v rozmezí od 1 do 10. Síťový provoz byl tentokráte značně vysoký. Nárůst síťového provozu se projevoval do připojeného 3. klienta. Po té již se hodnota držela na 60 procentech využití sítě.



Obr. 11.2: Test celého aplikačního serveru s databází.Klienti stahují celé řádky i s BLOBy

11.4 Závěr testu

Test nám tedy prozradil, že aplikační server je stabilní a dokáže se vypořádat s velkým nárůstem požadavků. Při připojení max. počtu klientů došlo sice k jistému přehlcení serveru(přesněji k přehlcení fyzického počítače), nicméně v případě testu s BLOBs byly součástí odpovědí i rozměrnější binární data. Tato data bylo vždy třeba nejprve stáhnout z databáze a následně sítí přeposlat ke klientovi. Tedy zde narážíme i na jistou technickou limitaci fyzických počítačů a síťového připojení. Plného síťového provozu 100 procent nemuselo být dosaženo z několika důvodů. Jednak je toto maximum pouze teoretické. V síti existuje jistá další režie, která se do přenesených dat nepočítá (u TCP/IP). Dále byl také serverový počítač zcela vytížen, rozkládal své dostupné prostředky mezi databázi Firebird, aplikační server a síťový provoz. Proto nemohl věnovat plnou pozornost síťovému provozu a z těchto důvodů mohlo dojít k poklesu síťové rychlosti. Ale i tak se rychlost pohybovala ve

dvou třetinách reálné rychlosti sítě.

V přiložených grafech můžeme pozorovat postupné narůstání doby pro zpracování dotazu,zejména u databázového serveru. To bylo způsobeno tím, že databázový server musel zodpovídat souběžné dotazy současně připojených klientů. Tím pádem musel danému jednomu klientovi přidělovat nižší a nižší výkon pro vyhledání požadavku. To bylo příčinou natahování doby pro dotaz.

Rozdílný čas mezi zpracováním aplikačního serveru a zpracováním dotazu u klienta nám ukazuje na zpoždění, které nám do komunikace dodává tato nová vrstva (aplikační server+síť). Toto zpoždění se v případě méně náročného testu bez Blobs pohybovalo přibližně do 3 ms (dle zátěže). V náročnějším testu diametrální nárůst tohoto zpoždění není jen vinou aplikačního serveru ale působí ho zejména 100Mbitový síťový provoz. Rozměrnější binární data totiž byla bufferována v aplikačním serveru a čekala na odeslání sítí.

Dále nutno podotknout, že i přes plně vytížený počítač, které působilo velké množství dotazů od klientů(dle testu až 260 za sekundu), pracoval aplikační server naprosto správně.

12 VÝSLEDKY STUDENTSKÉ PRÁCE

12.1 Výsledky

Výsledky studentské práce vhodně rozdělené do částí.

V této práci jsem prozkoumal a shromáždil programové metodiky sloužící pro synchronizaci databází pomocí jazyka SQL a v programovém prostředí BorlandDelphi.Dále jsem navrhnul postupy jednostranných a oboustranných synchronizací a přenosů dat.

Stěžejní částí této práce je vyřešení komunikace s klienty, kteří sídlí na různých platformách.K tomu byla vyvinuta aplikace, nesoucí obecný název aplikační server. Tím byl např. zprovozněn přístup klienta v J2ME na databázový SQL server. Rovněž jsou popsány možnosti připojení tohoto aplikačního serveru na různé databáze.Byl proveden průzkum jednotlivých databází, na jehož základě byly vybrány nejvhodnější databáze pro řešení. Navíc bylo provedeno testování různých přístupů ke stejné databázi a bylo navrženo nejoptimálnější řešení. Tím došlo k výraznému zkrácení přístupových dob. Dále je zde podrobně navržen přenosový protokol, sloužící ke komunikaci aplikačního serveru a klienta. Práce taktéž klade velký důraz na použité soketové TCP/IP komponenty sloužící přímo ke komunikaci s klientem. Zabývá se jejím správným fungováním a maximálním využitím sítě. Taktéž se krátce zamýšlí nad rozkladem zatížení aplikačního serveru na více jádrový procesor.

Taktéž nabízí návrhy řešení jednotlivých klientských aplikací. Zmiňované vývojové platformy jsou Java a Delphi.

Práce zmiňuje i možné databázové systémy použitelné k tomuto řešení.

Na závěr je celý projekt testován. Naměřené hodnoty jsou uvedeny v grafu a řádně zhodnoceny.

13 ZÁVĚR

Shrnutí studentské práce.

Podařilo se mi velmi výrazně rozšířit původní návrhy, které byly součástí semestrálního projektu.Původní návrhy se týkaly komunikace s různými databázemi,jejich možnou synchronizací a také popisovaly jednoduchou komunikaci s prostředím J2ME.

Rozšíření se podařilo díky zavedení aplikace s názvem aplikační server. Tato aplikace umožnila plnohodnotnou multiplatformní komunikaci s jednotlivými klienty a s různými databázemi.

Celou dobu jsem dbal na reálné použití tohoto projektu, proto jsem vystavoval jednotlivé části aplikace zátěžovým testům již v průběhu návrhu aplikace. Jejich hodnoty většinou neuvádím, protože to byly jen dílčí ještě neoptimalizované přístupy k řešení. Hlavním úkolem prvotních testů bylo kontrolovat funkční části jednotlivých bloků aplikace, aby se zabránilo pozdějším problémům, či naprostému kolapsu. Díky tomuto přístupu byla zvolená řešení mnohem více optimalizovaná, než kdyby se jen vycházelo ze strohých popisů nápověd a manuálů.

Samotné testy bloků byly většinou uvozeny podrobnou úvahou o možnostech a potřebách tohoto dílčího řešení.

Skloubením takto uvážených a testovaných částí bylo možno vytvořit aplikaci, na kterou je skutečně možno klást velké síťové nároky, což patří mimo jiné mezi základní úlohy aplikačního serveru.

Na závěr jsem provedl kompletní ověření funkčnosti tohoto návrhu na dvou počítačích zapojených do sítě. Serverový počítač jsem postupně přetížil požadavky. Jednotlivé přístupové doby jsem zapsal a vytvořil z nich grafy hodnotící základní chování tohoto aplikačního serveru ve spojitosti s databází Firebird. Ze závěrečného testu vyplývá, že mnou navrhované řešení je schopno odolat nejen velkému počtu dotazů a klientů, ale také postupnému přetížení serverové kapacity. Po celou dobu testu se aplikační server choval zcela korektně a dle systémových možností počítače komunikoval se všemi klienty.

LITERATURA

[1] *Internetový server www.Interval.cz* [online] Seriály o SQL - Skřivan, Jaromír (4. 8. 2000) [cit. 3. 12. 2007]. Dostupné z URL: <http://interval.cz/clanky/databaze-a-jazyk-sql/>.

[2] *Manuál Borland Delhi 7* Standartní součást instalace Borland Delphi 7

[3] *Manuál JAVA J2ME*

[4] *Systémová nápověda k Microsoft Windows*

[5] *Groff James, Weinberg Paul* SQL - Kompletní průvodce, Computer Press, ISBN: 80-251-0369-2

[6] *Wikipedia - PostgreSQL* [online] [cit. 28. 5. 2008]. Dostupné z URL: < http://www.pgsql.cz/index.php/PostgreSQL>.

[7] *Wikipedia - MySQL* [online] [cit. 28. 5. 2008]. Dostupné z URL: < http://cs.wikipedia.org/wiki/MySQL>.

[8] *Oficiální stránky Firebirdu* [online] [cit. 28. 5. 2008]. Dostupné z URL: <http://www.firebirdnews.org/docs/fb2min_cz.html>.

[9] *Seriál - Umíme to s Delphi* [online] [cit. 28. 5. 2008]. Dostupné z URL: <http://www.zive.cz/default.aspx textart=1article=110629>.

[10] *Aplikační server JBoss* [online] [cit. 28. 5. 2008]. Dostupné z URL: <http://www.root.cz/clanky/jboss-aplikacni-server/>.

[11] *Dokumentace Borlandu k Delphi 2007 (Nápověda)*

[12] *Skripta VUT - Herman,I.: Komunikační technologie* [online] [cit. 28. 5. 2008]. Dostupné z URL: <http://www.utko.feec.vutbr.cz/herman/kapitola4.pdf>.

[13] *Wikipedia - UDP* [online] [cit. 28. 5. 2008]. Dostupné z URL: <http://cs.wikipedia.org/wiki/UDP>.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

SQL Structured Query Language.Deklarativní programovací jazyk pro práci s databázemi

ODBC Open Database Connectivity.Umožňuje přesouvat data z jednoho typu databáze do jiného.

J2ME Java 2 microedition.Verze javy pro malá zařízení.

BDE Borland database engine. Typ databázových komponent a přístupů v Borland Delphi

dbExpress Typ databázových komponent a přístupů v Borland Delphi, využívající přímějších přístupů nez BDE.

Interbase Typ databázových komponent a přístupů v Borland Delphi, provádějící přímou práci s databází Firebird nebo Interbase. Interbase je taktéž název komerční databáze

Blob Binary large object. Databázový typ pro uložení rozměrných binárních dat.Vhodné pro obrázky, videa, zvuky ...

Stream Proud dat. Vytváří propojení s požadovaným zdrojem či uložištěm dat (zde s databází).Lze jej postupně načítat, či do něj zapisovat.