UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Martin Breskvar

Sinhronizacija podatkovnih virov z uporabo sinhronizacijskega ogrodja Microsoft Sync Framework

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Matija Marolt

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01728/2011
Datum: 15.03.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MARTIN BRESKVAR**

Naslov: **SINHRONIZACIJA PODATKOVNIH VIROV Z UPORABO SINHRONIZACIJSKEGA OGRODJA MICROSOFT SYNC FRAMEWORK**

SYNCHRONIZATION OF DATA SOURCES WITH MICROSOFT SYNC FRAMEWORK

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V diplomskem delu preučite ogrodja za sinhronizacijo podatkovnih virov, ki omogočajo sinhronizacijo mobilnih odjemalcev s centralno podatkovno bazo. Preglejte obstoječe rešitve na tem področju in na primeru Microsoft Sync Framework analizirajte delovanje tovrstnega ogrodja. Posvetite se tudi problemom varnosti, razhroščevanja in testiranja aplikacij, ki ogrodje uporabljajo.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

I Z J A V A O A V T O R S T V U
diplomskega dela

Spodaj podpisani **Martin Breskvar**,
z vpisno številko **63030139**

sem avtor diplomskega dela z naslovom:

Sinhronizacija podatkovnih virov z uporabo sinhronizacijskega ogrodja Microsoft Sync Framework.

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matije Marolta
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki „Dela FRI“

V Ljubljani, dne 15.9.2011 Podpis avtorja:

ZAHVALA

Na tem mestu se zahvaljujem mentorju doc. dr. Matiji Maroltu za nasvete, predloge in potrpežljivost pri nastajanju tega diplomskega dela.

Zahvaljujem se Mateji za neumorno vzpodbudo ter vsem ostalim, ki so me nesebično podpirali v času mojega študija. Iskrena hvala.

To delo je posvečeno moji mami.

Kazalo

Kazalo slik

Kazalo tabel

SEZNAM UPORABLJENIH KRATIC IN POJMOV

Naziv	Opis
OCA	Occasionally Connected Applications – občasno povezane aplikacije
MS SQL Server	Microsoft SQL Server – Relacijski sistem za upravljanje s podatkovnimi bazami
MS SQL Server CE	Microsoft SQL Server CE – Enostavna relacijska podatkovna baza z majhnim odtisom
VS	Microsoft Visual Studio - razvojno orodje
Metapodatek	Podatek o podatku
MSF	Microsoft Sync Framework – sinhronizacijsko ogrodje
MSS	Microsoft Storage Service – del MSF za podporo razvoja lastnih metapodatkovnih skladišč
IIS	Internet Information Services – spletni strežnik z integrirano varnostjo
SOA	Service Oriented Architecture – storitveno usmerjena arhitektura
.NET	.NET Framework, .NET ogrodje – skupek knjižnic za razvoj aplikacij na Microsoftovi platformi
WCF	Windows Communication Foundation – aplikacijski programski vmesnik v ogrodju .NET za razvoj povezanih, storitveno usmerjenihi aplikacij
Proxy	Usmerjevalni strežnik
T-SQL	Transact-SQL – poizvedovalni jezik podjetij Microsoft in Sybase, osnovan na standardu SQL-92
DML	Database Manipulation Language – podmnožica ukazov T-SQL za manipulacijo s podatki (dodajanje, odstranjevanje, urejanje in branje)
ADO.NET	ActiveX Data Object .NET – Skupek knjižnic v ogrodju .NET za komunikacijo z različnimi podatkovnimi viri.

POVZETEK IN KLJUČNE BESEDE

Vse več podjetij se odloča za informatizacijo terenskega poslovanja. Ne glede na uspešnost informatizacije terenskega poslovanja iz poslovnega vidika, pa vse mobilne aplikacije, tehnično gledano, pesti problem varnega dostopa do centralne podatkovne baze brez komunikacijskih izpadov.

V diplomskem delu sem razložil in implementiral sinhronizacijo podatkovnih virov, s pomočjo katere lahko na mobilne odjemalce prenesemo podmnožico podatkov iz centralne podatkovne baze. Na terenu popravljene in obogatene podatke lahko kasneje s pomočjo sinhronizacije prenesemo nazaj na centralni strežnik. Sinhronizacija med odjemalci in strežnikom poteka preko WCF storitve. Demonstracijska rešitev je implementirana z uporabo sinhronizacijskega ogrodja Microsoft Sync Framework, ki za sinhronizacijo uporablja podatkovno bazo na MS SQL Server 2008 R2 in MS SQL Server CE 3.5. Podrobno so obrazložene tudi bazne procedure, ki se uporabljajo med samo sinhronizacijo. Ker je pričujoče delo nastalo na podlagi resničnega projekta v industriji, so vanj vključene tudi metode varovanja, razhroščevanja ter testiranja.

Ključne besede: Microsoft Sync Framework, MSF, sinhronizacija podatkovnih baz

ABSTRACT

More and more companies are opting for informatisation of field operations. Regardless of the outcome of the integration, looking from a business perspective, all mobile applications technically face the problem of safe access to the central database without communication failures.

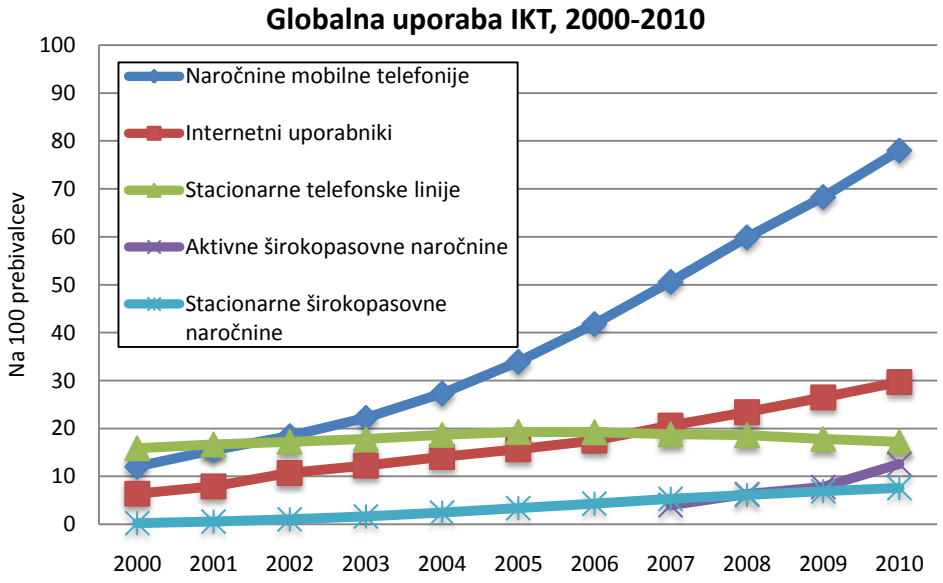
In this thesis, I have explained and implemented synchronization of various data sources, through which the end-users can transfer a subset of centralized data to their mobile devices. Enriched information from the field is later, again by means of synchronization, transferred back to the central database. Synchronization between the consuming devices and server is implemented via WCF service. Mock solution is implemented using Microsoft Sync Framework with MS SQL Server 2008 R2 and MS SQL Server CE 3.5 acting as server and client database back-ends respectively. Stored procedures, which are used during the synchronization process, are explained in detail. Since this work is based on a real-life project, it also includes information on security, debugging and unit testing.

Keywords: Microsoft Sync Framework, MSF, database synchronization

1 Uvod

V letu 1992 se je kot plod razvijalcev v podjetju IBM na trgu pojavil Simon - prvi pametni telefon. Kasneje so ameriškemu gigantu sledili tudi v podjetjih Nokia, Hewlett Packard in Ericsson. Nihče ni mogel predvideti, kako se bo obrnilo krmilo razvoja mobilnih naprav in njim namenjenih operacijskih sistemov, a si vendarle skoraj dve desetletji kasneje nihče več ne postavlja vprašanja ali je bil razvoj pametnih telefonov korak v napačno smer ali ne. Enako velja za prenosne računalnike. Več kot očitno je, da so tovrstne naprave nadvse priljubljene in uporabne, kar je razvidno tudi iz statističnih podatkov [1].

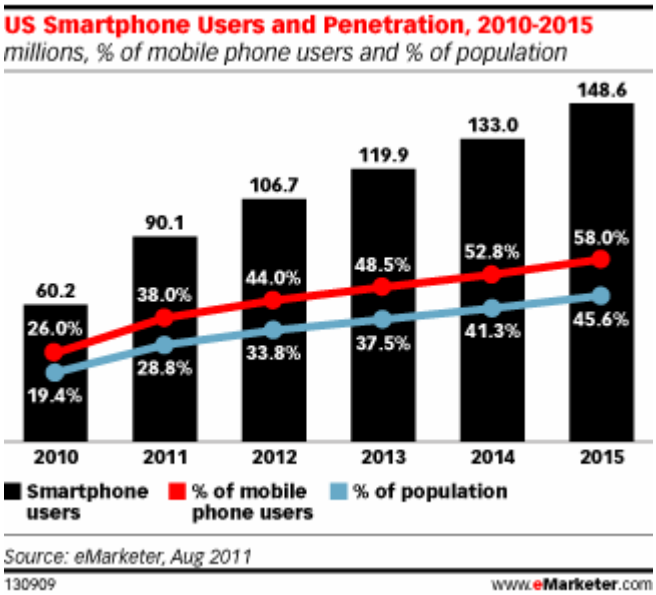
Jasno je, da se uporaba mobilne telefonije in interneta vztrajno povečuje. Grafikon na sliki 1 to potrjuje. Implicitno je iz grafa razvidno tudi, da se povečuje pokritost z GSM signalom. V letu 2003 je bila svetovna pokritost 61%, leta 2009 pa že 90%.



Slika 1: Globalna uporaba IKT

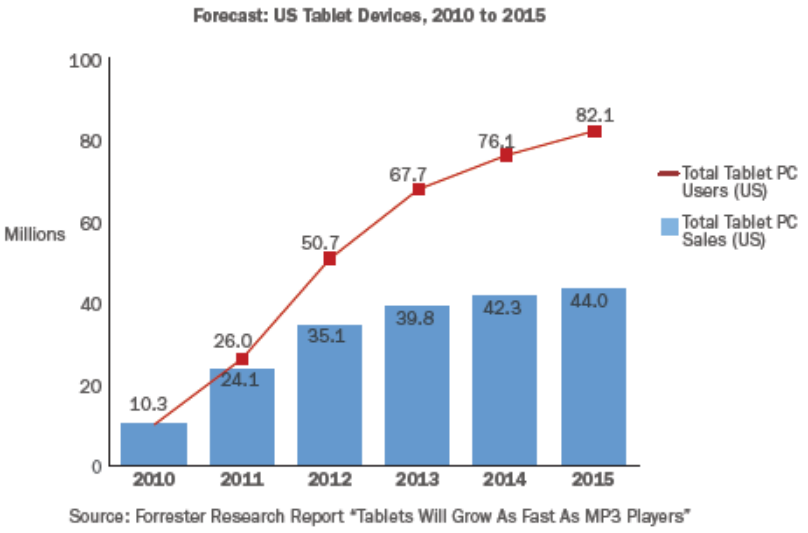
Iz grafikona je razvidno tudi dejstvo, da je stacionarna telefonija v zatonu. Po drugi strani je mobilni internet vsak dan hitrejši, kar se odraža v nizki rasti števila stacionarnih širokopasovnih naročnin [2].

Zgornja statistika zajema vse mobilne naročnine. Vedeti je potrebno, da se naročnine mobilnih operaterjev ne uporabljajo zgolj za telefoniranje. Obstajajo naprave, ki za svoje delovanje uporabljajo GSM omrežje, vendar ne za namene govora temveč za pošiljanje podatkov. Primera takšnih naprav sta recimo hišni alarm ter mobilni internet.
Za boljšo utemeljitev diplomskega dela, je na tem mestu potrebno podati statistike o uporabi pametnih telefonov, prenosnikov, tabličnih računalnikov in podobno. Tovrstne naprave namreč predstavljajo glavnino mobilnih naročnin. Uporaba pametnih telefonov je do sedaj vsako leto naraščala. Spodnji grafikon prikazuje projekcijo za prihodnost in brez dvoma lahko trdimo, da gre razvoj v prid pametnim telefonom [3].



Slika 2: Napoved uporabe pametnih telefonov v ZDA

Podobno kot uporaba pametnih telefonov, bo tudi uporaba tablic naraščala. Po nekaterih statistikah so imele ponekod tablice že sedaj boljšo prodajo kot prenosni računalniki. Kljub temu imajo prenosniki zaenkrat globalno gledano precejšnjo prednost v samem tržnem deležu. Poleg tega, so tablice strojno omejene, saj zaenkrat še ne dosegajo takšnih zmogljivosti kot prenosni računalniki [4].



Slika 3: Napoved uporabe tabličnih naprav v ZDA

1.1 Motivacija in namen diplomskega dela

Z evolucijo pametnih telefonov je prišlo do poplave mobilnih aplikacij. Inovativnost pri izdelavi operacijskih sistemov je pripeljala mobilno telefonijo na mesto, kjer se nahaja danes. Integracija mobilnega telefona v naš vsakdan in poslovanje je bila sorazmeroma hitra. Le bežno se še spominjamo časov, ko smo morali za branje elektronske pošte ali brskanje po spletu obvezno sedeti za namiznim računalnikom in ob počasni povezavi čakati na željene informacije. Vsakodnevno večje zahteve po mobilnosti in dostopnosti informacij, so nas

pripeljale do točke, kjer si praktično več ne predstavljamo dneva brez uporabe mobilnega telefona. Infiltracija mobilnih telefonov v naša življenja in miselnost "on the go", sta pripomogla k temu, da si tudi poslovanja več ne predstavljamo v obliki papirja in svinčnika, temveč v obliki mobilnega telefona in prsta oziroma pisala (angl. *stylus*).

Vsem se vedno nekam mudi in nemalokrat pisarniško delo vključuje tudi delo na terenu. Mnogokrat, če ne kar vedno, se dogaja, da je potrebno podatke iz terena prenašati iz enega medija (fizičnega ali elektronskega) na drugega in obratno. Tovrstno početje trati naš čas in energijo, ki bi ju lahko porabili bolj smotrno in ravno ta problem je botroval k informatizaciji poslovanja na terenu.

Ob dostopnosti informacij na vsakem koraku in s tem rešitvi problema dostopnosti samih informacij, pa se je pojavil nov problem. Veliko število mobilnih aplikacij praviloma svoje podatke črpa iz podatkovnih strežnikov preko medmrežja. Ti podatki se lahko hranijo na napravi ali pa so neposredno dostopni preko povezave. Ko aplikacija do podatkov dostopa preko povezave, je razvoj takšne rešitve bolj preprost in posredno cenejši, hkrati pa prinaša veliko omejitev. Naprava mora imeti stalno vzpostavljeno povezavo z medmrežjem, kar predstavlja strošek in onemogoča lokacijsko neodvisnost, saj je povezava mnogokrat pogojena z geografskim položajem mobilne naprave. Zahteva po prostem gibanju ob uporabi naprave nas torej sili k razvoju rešitev, ki bi v nekem določenem trenutku, ko povezava je na voljo, podatke prenesle na napravo, nato pa podatke uporabljale neodvisno od nje – v nepovezanem načinu (angl. *offline*). Takšne rešitve imenujemo *občasno povezane aplikacije* (angl. *Occasionally connected applications - OCAs*). Takšna aplikacija je za uporabnike seveda bolj privlačna, a žal prinaša nove težave. Kot bi mignil smo namreč ustvarili problem, ki hkrati predstavlja tudi jedro tega dela. Ustvarili smo kopijo oziroma podmnožico centralne podatkovne baze in jo shranili na napravo. Pri predpostavki, da uporabnik podatkov ne pošilja nazaj na strežnik, logičnih problemov praktično ni. Ko pa razmišljamo bolj stvarno, pridemo do ugotovitve, da uporabniki po obdelavi na mobilni napravi, podatke velikokrat želijo pošiljati nazaj na strežnik. Če upoštevamo še dejstvo, da je uporabnikov, ki uporabljajo centralno podatkovno bazo običajno več, kaj kmalu pridemo do ugotovitve, da prihaja do konfliktov in podvajanj v samih podatkih. To je seveda iz stališča praktične uporabe, kot tudi poslovanja, nesprejemljivo [5].

Ob pošiljanju podatkov nazaj na strežnik je potemtakem potrebno zagotoviti pravilno hrambo in predvsem uskladitev obstoječih podatkov z novimi, ki so posredovani iz mobilne naprave. Ta postopek imenujemo sinhronizacija podatkov.

Namen tega dela je predstaviti možne prijeme pri sinhronizaciji podatkov z uporabo Microsoftovega ogrodja za sinhronizacijo, imenovanega Microsoft Sync Framework.
Poudarjena je sinhronizacija podatkovnih baz, dotaknili pa se bomo tudi sinhronizacije datotek in map, saj je takšna sinhronizacija zlasti v kontekstu mobilnih naprav[1] lahko zelo uporabna.

[1] Pod imenom "mobilne naprave" razumemo pametne telefone, dlančnike, tablične računalnike, tablice, USB ključe in prenosne računalnike.

1.2 Predvideni rezultati in možnost njihove uporabe

Dandanes operiramo z ogromnimi količinami podatkov. Ti podatki se zbirajo in shranjujejo s stopnjo, kakršne ne pomnimo. Večina teh podatkov pa je na žalost vkleščena v specifične aplikacije oziroma formate, kar otežkoča njihovo integracijo, ponovno uporabo ter obdelavo.

V prejšnjem poglavju je bila razlaga uporabnosti sinhronizacije podatkov usmerjena v mobilno telefonijo. Kljub temu, da je svet pametnih mobilnih telefonov ena izmed najbolj očitnih izbir pri uporabi sinhronizacije, še zdaleč ni edina. Sinhronizacija podatkov se mnogokrat uporablja tudi med običajnimi PC-ji, prenosnimi računalniki, USB ključi ipd.

Predviden rezultat tega diplomskega dela je delujoča implementacija N-nivojske sinhronizacije med MS SQL Server 2008 R2 ter SQL CE 3.5 podatkovnima bazama z uvedbo WCF storitve. Tako grajena rešitev omogoča sinhronizacijo preko požarnih zidov. Poleg demonstracijske implementacije, bodo na podlagi enostavnih primerov predstavljeni posamezni deli ogrodja. Dotaknili se bomo tudi tem o varnosti, testiranju pravilnega delovanja (angl. *unit testing*) ter razhroščevanju (angl. *debugging*).

Želja avtorja je, da si razvijalci ne bi več belili glav z vprašanjem “Kako bomo sinhronizirali podatke?” temveč “Kaj in kdaj bomo sinhronizirali?”.

2 Obstoječe rešitve na področju sinhronizacije podatkov

Na tržišču obstaja precej aplikacij, ki omogočajo takšno ali drugačno sinhronizacijo. Tovrstne aplikacije je v najbolj grobem smislu moč kategorizirati v dve skupini:

- Aplikacije, ki so namenjene končnim uporabnikom (angl. *end user*) in
- Knjižnice oziroma rešitve, ki so namenjene razvijalcem in se uporabljajo za razvoj aplikacij za končne uporabnike.

Med zgoraj naštetima kategorijama je več kot očitno ogromna razlika. V prvi skupini (aplikacije namenjene končnim uporabnikom) gre za programsko opremo, ki je zmožna vršiti samo zelo osnovne sinhronizacije. V sinhronizacijski proces je običajno vključena samo ena oseba. Ta oseba poseduje več različnih naprav (PC, prenosnik, mobilni telefon, ipd.) in vrši sinhronizacijo med njimi. Aplikacije v prvi skupini, ki jih je na tem mestu morda bolj smiselno poimenovati orodja, sinhronizirajo telefonske imenike, koledarje, zapiske, datoteke in mape med posameznimi napravami. Logično, sinhronizacija lahko poteka samo med tistimi napravami, ki jih sinhronizacijsko orodje pozna, saj orodje velikokrat vsebuje tudi logiko za priključitev na samo napravo. Primer takšnga orodja je recimo programska oprema, ki jo prejmemo ob nakupu mobilnega telefona. Iz zadnjega stavka je implicitno razvidno tudi dejstvo, da mora sinhronizacijsko orodje »poznati« obe strani v sinhronizacijskem postopku. V kontekstu same sinhronizacije, ni nikjer centraliziranih podatkov oziroma centralne podatkovne baze. V prvo kategorijo spadajo tudi orodja, ki so zmožna na primer sinhronizirati koledar MS Outlook s telefonom. Tudi takšna orodja morajo poznati način hrambe podatkov na mobilnem telefonu in način hrambe podatkov na osebnem računalniku. Takšne aplikacije so okorne, vsiljujejo vnaprej določeno logiko, končni uporabniki pa so velikokrat zmedeni, saj je dandanes za potrebe vsakdanjega dela težko obvladovati na desetine tovrstnih orodij. Primerov takšnih orodij ne bomo naštevali, ker jih je enostavno preveč in niti nimajo neposredne povezave s tem diplomskim delom.

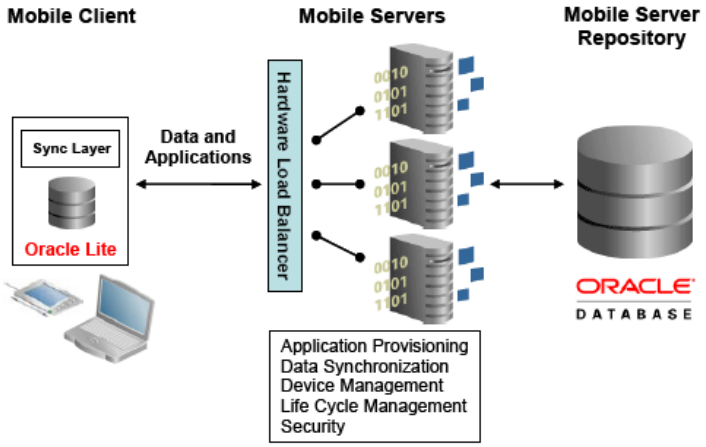
Druga kategorija aplikacij pa je z vidika sinhronizacije veliko bolj zanimiva. V iskanju sinhronizacijskih rešitev sem naletel na peščico takšnih, ki spadajo v drugo kategorijo. Pogledali si bomo nekaj obstoječih rešitev ter v nadaljevanju diplomskega dela zelo podrobno opisali eno izmed njih – Microsoft Synchronization Framework (MSF). Razlogi za izbiro slednjega so navedeni v nadaljevanju.

2.1 Oracle Database Lite 10g

Oracle Database Lite 10g, rešitev podjetja Oracle, omogoča sinhronizacijo podatkov iz lahkih odjemalcev na strežnik in nazaj. Osnovni gradnik na strani odjemalca so lokalna podatkovna baza ter knjižnice za sinhronizacijo in komunikacijo z Oracle Database Lite Mobile Server-jem.

Odjemalec lahko bazira na različnih platformah, vključno z Windows 2003/XP/Vista, Windows Mobile, Pocket PC, Linux, embedded Linux in Symbian OS. Podprta sta dva tipa podatkovne baze: Oracle Berekley DB in SQLite. Odjemalci z glavnim repozitorijem (centralno podatkovno bazo) komunicirajo preko Oracle Database Lite Mobile Server-ja. Uporabniki na odjemalcih vse potrebne knjižnice za delovanje sinhronizacije prejmejo od centralnega strežnika v obliki ene same izvršljive datoteke.
Oracle Database Lite Mobile Server je postavljen med centralno podatkovno bazo in odjemalce, kar administratorjem omogoča tudi upravljanje s samimi mobilnimi napravami.
Spodnji diagram prikazuje osnovne gradnike te rešitve.

Podatkovna baza hrani centralne podatke in metapodatke odjemalcev, kar omogoča lažji razvoj na odjemalcih.



Slika 4: Oracle Database Lite 10g

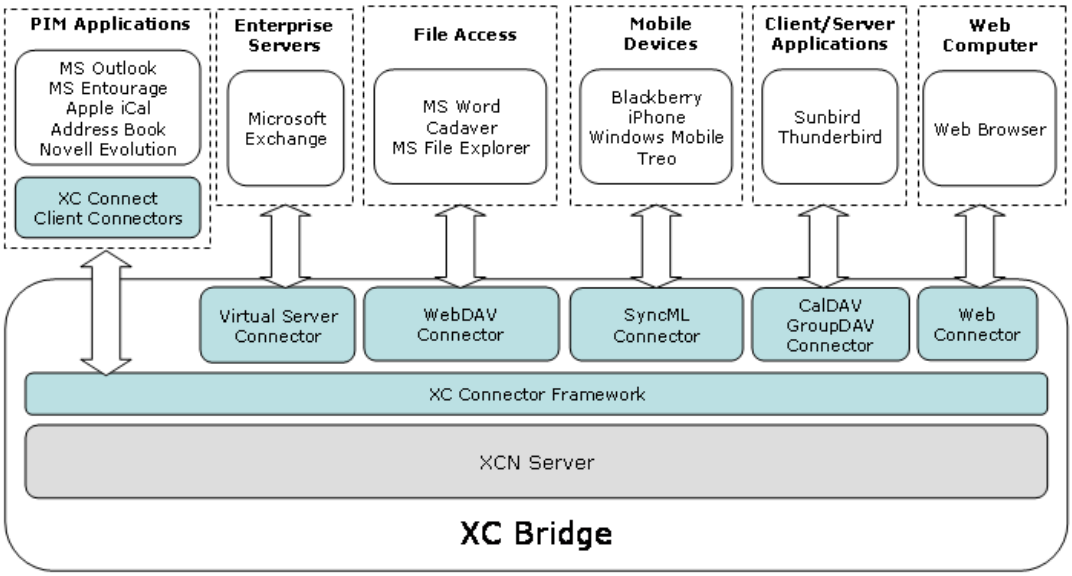
Sama rešitev je zasnovana v duhu “one click” saj je administracija v celoti v domeni administratorjev mobilnih strežnikov (Oracle Database Lite Mobile Server). Rešitev ponuja tudi različne uporabne funkcionalnosti, kot je na primer opozorilo ob nizkem stanju baterije, avtomatična sinhronizacija brez uporabnikove vednosti (aplikacjia lahko teče neprekinjeno) in podobno. Sinhronizacijski konflikti se lahko rešujejo avtomatsko (Client wins / Server wins) ali pa se shranijo v podatkovno bazo in čakajo na poseg administratorja.

Hkrati Oracle ponuja tudi Mobile Development Kit (MDK), ki razvijalcem omogoča hiter razvoj občasno povezanih mobilnih aplikacij. Dostop do sinhronizacijskih knjižnic in podatkovne baze je dostopen preko programskih vmesnikov (angl. *Application Programming Interface – API*). Razvoj aplikacij je mogoč v programskih jezikih Java in C/C++ in vseh .NET jezikih. Podatkovna baza omogoča dostop preko JDBC, ODBC in ADO.NET. Rešitev na strežniškem koncu potrebuje Oraclovo podatkovno bazo.

Gledano s stališča razvijalca je ta rešitev zelo obsežna in kvalitetna ter ponuja več kot dovolj možnosti za razširitve [6].

2.2 XC Bridge

Podobno kot podjetje Oracle, je tudi podjetje Xchange Network razvilo svojo sinhronizacijsko rešitev, ki so jo poimenovali XC Bridge. Gre za rešitev, ki omogoča sinhronizacijo podatkov med različnimi odjemalci. Ker je XC Bridge osnovan na razvojni platformi Java, deluje na praktično vseh namiznih in mobilnih operacijskih sistemih. XC Bridge za povezavo med odjemalci in strežniki uporablja stičnike (angl. *Connectors*). Možen je hiter razvoj aplikacij, saj veliko časa posvečajo ponovni uporabljivosti programske kode (angl. *code reusability*). Sinhronizacija je možna v obe smeri in podpira vse vrste podatkov. Podatki so med samim prenosom kriptirani z 128-bitnim ključem, na samem strežniku pa so varovani s pomočjo regulacije dostopa (angl. *permissions*).
Podobno kot Oracle Database Lite, tudi XC Bridge uporablja sinhronizacijski strežnik. Poimenovali so ga XCN Software Server.



Slika 5: XC Bridge

Ob pogledu na zgornjo shemo v grobem opazimo dva osnovna gradnika te rešitve:

- XCN Server – sinhronizacijski strežnik ter
- Stičnike, ki so osnovani na XC Connector Framework-u, razvojnem ogrodju za stičnike

XCN Server deluje kot vstopna točka v centralno podatkovno bazo (enako kot pri Oracle Database Lite) in nadzoruje sinhronizacijske procese. Na drugi strani stičniki obdelajo prejete podatke v XML obliki ter posodobijo lokalne podatkovne baze. Naloga stičnikov je tudi pošiljanje terenskih sprememb nazaj na XCN Strežnik. Rešitev ponuja tri vrste stičnikov:

- Virtualni stičniki: Omogočajo povezovanje različnih strežnikov in s tem praktično sinhronizacijo dveh sodelovalnih omrežij (angl. *collaboration network*).

- Namizni in mobilni stičniki: Ti stičniki so nameščeni na odjemalcih.
- Spletni stičniki: Omogočajo dostop in urejanje podatkov preko spletnega brskalnika.

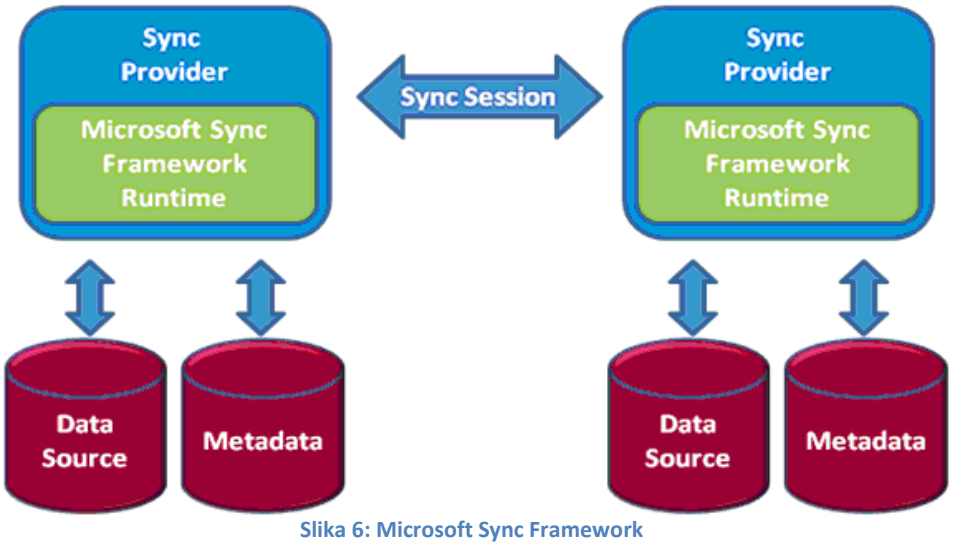
Rešitev že v samem začetku ponuja nekaj stičnikov, če pa le-ti ne bi bili ustrezni, XC Bridge ponuja XC Connector Framework – ogrodje s pomočjo katerega lahko razvijamo lastne stičnike.

Podobno kot rešitev podjetja Oracle, je tudi rešitev XC Bridge zasnovana zelo premišljeno in natančno. Je zelo prilagodljiva in interoperabilna, kar kot končni rezultat daje navidez popolnoma homogeno omrežje v teoretično (in praktično) zelo heterogenem svetu [7].

2.3 Microsoft Sync Framework

Podjetje Microsoft je že nekaj let pred lansiranjem Microsoft Sync Framework-a (MSF) ponujalo rešitev za sinhronizacijo PIM podatkov, ki se je imenovala ActiveSync. Danes se v rahlo izboljšani in razširjeni različici na trgu pojavlja pod imenom Windows Mobile Device Center. Microsoft je podlegel pritisku skupnosti ter razvil sinhronizacijsko ogrodje, ki bi omogočalo sinhronizacijo vseh mogočih podatkov. Poimenovali so ga MSF.
Ogrodje je grajeno zelo podobno kot Oracle Database Lite oziroma XC Bridge. Prva opazna razlika je ta, da MSF ne potrebuje posebnega sinhronizacijskega strežnika.

Princip sinhronizacije je zelo podoben, kot pri prej omenjenih rešitvah. Kljub temu, je na tem mestu potrebno omeniti, da sta zgornji dve rešitvi resnično rešitvi v pravem pomenu besede (angl. *solution, business solution, enterprise solution*). Torej od A do Ž. MSF ni bil ustvarjen za prodajo, ampak ker je tako želela stokovna skupnost (angl. *community*). Zato tudi ni tako megalomansko zastavljeno, kot na primer rešitev podjetja Oracle. Navzlic temu, je MSF zelo zmogljivo sinhronizacijsko ogrodje.



Slika 6: Microsoft Sync Framework

Na zgornji shemi vidimo osnovni sinhronizacijski tok. Na obeh straneh imamo podatkovne baze, ki beležijo lastne metapodatke. S temi podatkovnimi bazami komunicira

sinhronizacijski ponudnik (angl. *sync provider*). Le-ta se izvaja v kontekstu MSF izvrševalnega okolja (angl. *runtime*).
MSF deluje samo na operacijskih sistemih Windows in Windows Mobile, kar sigurno predstavlja precejšnjo omejitev. Podpira vse Microsoftove podatkone baze in strežnike ter tudi veliko konkurenčnih podatkovnih baz. Če bi recimo želeli vršiti sinhronizacijo med podatkovno bazo Oracle in SQLite, nam MSF to omogoča [8].

2.4 Primerjava

Glede na to, da je takšnih sinhronizacijskih ogrodij/rešitev malo, je smiselno narediti primerjavo med njimi.V spodnji tabeli so na levi strani navedene posamezne kategorije in funkcionalnosti.

Tabela 1: Primerjava obstoječih rešitev

	Oracle Database Lite 10 g	XC Bridge	MSF
Razširljivost	✔	✔	✔
Kompleksnost razširjanja	Dedovanje od razredov v ogrodju in uporaba MDK	Dedovanje od razredov v ogrodju	Dedovanje od razredov v ogrodju
Platforma na strežniku	Povsod kjer deluje Java	Povsod kjer deluje Java	Vsi Windows OS od XP naprej in strežniški sistemi od Windows Server 2000 naprej
Platforma na odjemalcu	Windows 2003/XP/Vista, Windows Mobile, Pocket PC, Linux,Symbian OS, Android, BlackBerry OS	BlackBerry OS, iOS (iPhone), Windows Mobile,Treo (Palm)	Vsi namizni Windows OS od XP naprej, vsi strežniški OS od Windows Server 2000 naprej, Windows Mobile
Podatkovna baza na strežniku	Oracle	Ni podatka	Ni posebnih zahtev
Podatkovna baza na odjemalcu	Berkeley DB, SQLite	Ni posebnih zahtev; podpira povezavo preko COM, CORBA, AppleEvents, JDBC, ODBC,...)	Ni posebnih zahtev
Spletna sinhronizacija	✘ Ni podatka	✘ Ni podatka	✔ (RSS/Atom)
Sinhronizacjia datotek in map	✔	✔	✔

Sinhronizacija podatkovnih baz	✔	✔	✔
Vnaprej pripravljeni stičniki	✔	✔	✔ (datoteke in mape, podatkovne baze, RSS/Atom)
Nadzor nad odjemalci	✔ (preko Oracle DB Lite Server-ja)	✘ Ni podatka	✘
Varnost povezave in dostopa do centralnega strežnika	✔	✔ 128-bitna enkripcija povezave in varnost dostopa	✔ Varnost dostopa, enkripcija povezave po želji
Povezljivost preko omrežij	✔ Spletna storitev	✔ Virtualni stičniki	✔ Posredovalni razredi (Proxy classes)
Sinhronizacija v oblak (angl. *cloud*)	✘ Ni podatka	✘ Ni podatka	✔ Windows Azure
Sinhronizacija PIM	✔	✔	✔ (potrebno je razviti lasten sinhronizacijski ponudnik)
Priložena razvojna orodja	✔ (Oracle MDK)	✘	✘
Licenca	✘ Plačljiva	✘ Plačljiva	✔ Brezplačna

Po pregledu vseh zgoraj naštetih rešitev, sem se odločil za implementacijo demonstracijske aplikacije v MSF. Razlogov za to je več. Prvi razlog je, da je MSF prosto dostopen. To nam implicitno tudi pove, da v primeru težav hitreje in lažje najdemo rešitev na medmrežju. V prid MSF govori tudi dejstvo, da za implementacijo ne potrebujemo posebne lastniško zaščitene strežniške programske opreme (angl. *proprietary software*), kot je to potrebno pri rešitvah Oracle Database Lite in XC Bridge. Zgolj za namene demonstracije sinhronizacije podatkov v podatkovnih bazah ne potrebujemo dodatnih funkcionalnosti, ki jih ponujata Oracle in Xchange Network. MSF je več kot primeren in zato sem ga izbral kot demonstracijsko platformo.

3 Opis problema in navodila za namestitev demonstracijske aplikacije

3.1 Opis problema

Po naročilu tujega naročnika smo morali v podjetju razviti rešitev, kjer bi trgovski potniki za svoje delo na terenu uporabljali pametne mobilne telefone (angl. *smartphone*). Zahteva naročnika je bila, da se podatki iz terena avtomatsko prenašajo na centralni strežnik in obratno. Naročnik je potreboval tako namizno aplikacijo za delo v podjetju, kot tudi mobilno aplikacijo. V podjetju je naročnik v danem trenutku že uporabljal MS SQL Server 2008 R2 kot centralni podatkovni strežnik, zato je morala namizna aplikacija uporabljati le-tega. Končni uporabniki uporabljajo izključno MS Windows XP. Malo pred sklenitvijo posla, je naročnik vsem trgovskim potnikom kupil pametne mobilne telefone. Mobilne telefone je poganjal operacijski sistem Windows Mobile 6.5. Tako je bila tudi iz mobilne strani postavljena omejitev glede razvojne platforme. Ker sta bila tako namizna kot mobilna platforma Microsoftova, je bil prva izbira MSF. Vse ostale možnosti so bile predrage ali pa niso izpolnjevale pogojev.

Končna rešitev je precej kompleksna in obsega okrog 120 tabel v podatkovni bazi. Namen tega diplomskega dela ni razlaga kompleksne rešitve določenega problema iz industrije, temveč bralcu predstaviti probleme sinhroniziranja raznovrstnih podatkov preko praktičnih implementacij. V naslednjih poglavjih so zato predstavljene posamezne komponente MSF, obenem pa so podani tudi opisi implementacij. V poglavju 4.5 sta podana dva preprosta sinhronizacijska primera:

- Sinhronizacija datotek in map (poglavje 4.5.1) ter
- Sinhronizacija podatkovnih baz (poglavje 4.5.2).

Pojasnjeni so tudi prijemi za razhroščevanje in testiranje sinhronizacijskih aplikacij.

3.2 Navodila za namestitev demonstracijske aplikacije

Za uspešno delovanje demonstracijske aplikacije, ki je diplomskemu delu priložena na zgoščenki, je potrebno namestiti različne knjižnice in MS SQL strežnik.

3.2.1 Microsoft .NET Framework

MS .NET Framework je programsko ogrodje, skupek knjižnic za razvoj aplikacij v .NET okolju. Namestiti je potrebno zadnjo različico tega ogrodja. Izvršljiva datoteka je dosegljiva na spletnem naslovu [9].

3.2.2 MS SQL Server 2008 R2 in podatkovna baza

Za pravilno delovanje sinhronizacije moramo namestiti MS SQL Server 2008 R2.
Na strežniku je potrebno ustvariti podatkovno bazo. Uporabnik ima dve možnosti:

1. Na zgoščenki, ki je priložena diplomskemu delu, se nahajata datoteki *DiplomaSync.msf* ter *DiplomaSync_log.ldf*. Datoteki predstavljata vnaprej pripravljeno podatkovno bazo. V MS SQL Server Management Studio-u je potrebno izvesti priklop (angl. *attach*) podatkovne baze.
2. Uporabnik lahko s pomočjo T-SQL skript sam ustvari podatkovno bazo. Datoteka *crebas.sql* v kateri se nahaja skripta se nahaja na zgoščenki.

V primeru, da bomo ubrali pot pod številko 2, je potrebno preveriti, če ima uporabnik *student* pravice nad bazo DiplomaSync.

3.2.3 Microsoft Sync Framework

Za uspešno izvajanje sinhronizacije je potrebno namestiti izvajalno okolje za Microsoft Sync Framework (MSF). Izvršljive .MSI datoteke je možno dobiti na internetnem naslovu.
Če imamo 32-bitni sistem je potrebno namestiti *SyncSDK-v2.1-x86-ENU.msi,* če poganjamo 64-bitni sistem pa *SyncSDK-v2.1-x64-ENU.msi*. Trenutna verzija MSFje 4.0. Le-ta je grajena na podlagi verzije 2.1, zato moramo prvo namestiti zgoraj navedene knjižnice.
Sledi namestitev knjižnice *SyncFx-v4.0-October2010CTP.msi* [10].

3.2.4 Internet Information Services in ASP.NET

Del demonstracijske aplikacije uporablja WCF storitev, ki gostuje na ASP.NET razvojnem strežniku, ki je del MS Visual Studio. V primeru, da bi bralec želel spletno storitev namestiti na svoj lasten spletni strežnik, potrebuje pravilno konfiguriran IIS in ASP.NET.

3.2.5 Microsoft .NET CF in MSF runtime

Odjemalec mora ravno tako namestiti .NET Framework, saj sinhronizacijske knjižnice temeljijo na njem. MS .NET CF je podmnožica knjižnic MS .NET, prirejena za manj zmogljive naprave. Iz .NET CF so tudi odstranjene knjižnice, katerih uporaba na tovrstnih napravah ni smiselna. V našem primeru namestitev ni potrebna, saj bo odjemalec Windows Forms aplikacija, ki se bo izvajala na polnem .NET ogrodju.

Hkrati mora imeti odjemalec nameščeno tudi sinhronizacijsko izvajalno okolje (angl. *runtime*). Le-ta je dostopen na naslovu, ki je naveden v poglavju 3.2.3 [11].

3.2.6 Microsoft Visual Studio 2010

Za uspešen izgrad izvorne kode potrebujemo MS Visual Studio 2010. Gre za razvojno orodje, ki omogoča razvoj raznovrstnih aplikacij na .NET platformi. Okrnjeno različico orodja je sicer mogoče dobiti zastonj, vendar za potrebe naše demonstracije potrebujemo polno verzijo [12].

3.2.7 Navodila za zagon demonstracijske aplikacije

Aplikacija sestoji iz štirih različnih projektov:

- SyncLib
- ClientSyncLib
- WCF storitev (spletna stran)
- TestClient

Potrebno je pognati datoteko *SyncDiploma.sln*. Aplikacija je nastavljena tako, da se ob zagonu zaženeta dva projekta: WCF storitev ter TestClient. V desnem spodnjem kotu, pri uri in datumu, se pojavi ikona, ASP.NET razvijalskega strežnika. Ta predstavlja WCF storitev, ki teče lokalno. Hkrati se zažene tudi Windows Forms aplikacija (projekt TestClient), s katero nadzorujemo potek sinhronizacije.

4 Sinhronizacijsko ogrodje Microsoft Sync Framework – MSF

4.1 Arhitektura MSF

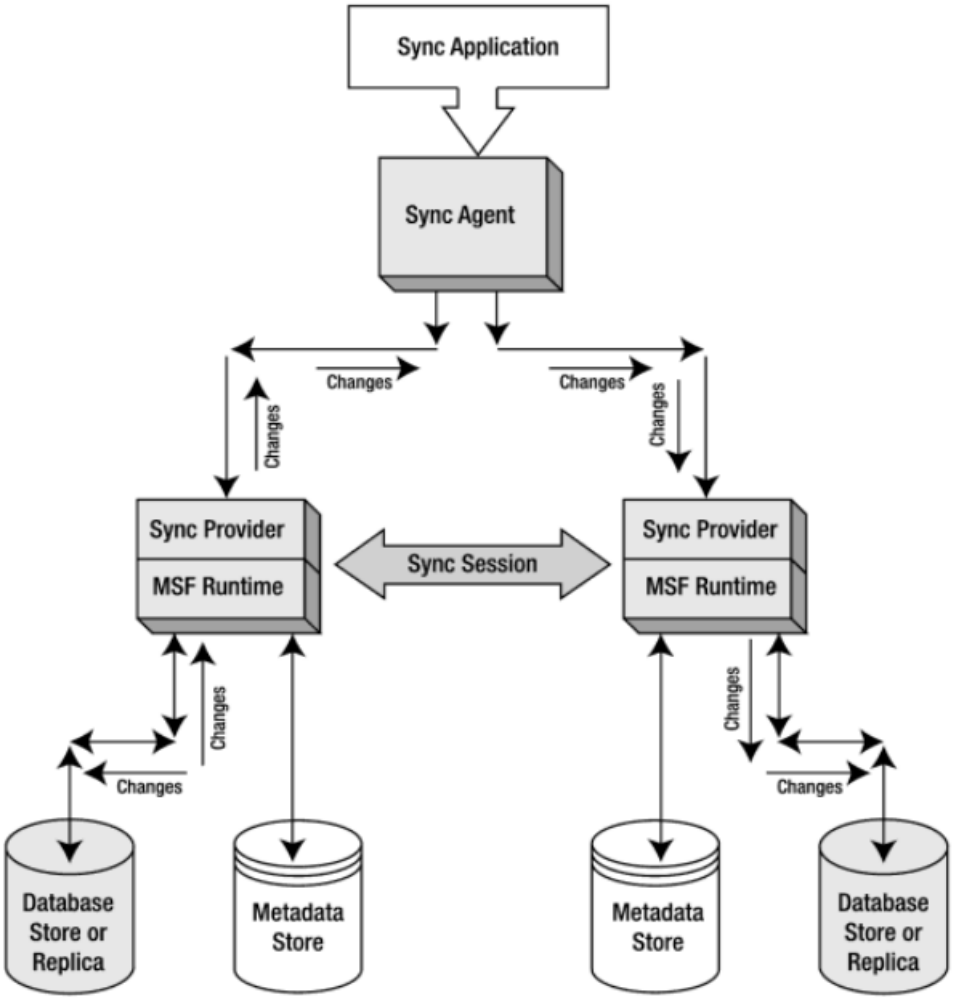
MSF je sestavljen iz štirih večjih gradnikov. Velikokrat prihaja do zmede, saj samo ime "Microsoft Sync Framework" lahko nastopa v dveh vlogah: kot produkt in kot družina produktov. Ljudje namreč velikokrat rečejo "MSF" dejansko pa razmišljajo le o prvem gradniku – *Sinhronizacijski ponudniki za podatkovne baze (Database providers).* Na shemi spodaj so navedeni glavni gradniki MSF:

- Sinhronizacijski ponudniki za podatkovne baze,
- Spletni ponudniki,
- Sinhronizacijski ponudnik za datoteke in mape,
- Jedrne komponente za razvoj lastnih sinhronizacijskih ponudnikov in ostalih komponent [13].

Slika 7: Arhitektura MSF

4.2 Sinhronizacija z MSF

Če hočemo razumeti kako poteka sinhronizacija, moramo prvo definirati nekatere pojme. Na diagramu *slika 8* so predstavljeni členi sinhronizacijske verige.



Slika 8: Sinhronizacijski tok

Sinhronizacija vedno poteka med dvema podatkovnima viroma oziroma podatkovnima skladiščema. Tem podatkovnim skladiščem pravimo *replike*. Za vsako repliko potrebujemo *sinhronizacijskega ponudnika* (sync provider). Naloga sinhronizacijskega ponudnika je sprejemanje in apliciranje sprememb na repliki, za katero je zadolžen. Sinhronizacijski ponudnik je specifičen za vsak tip replike. Tipi replik so razloženi v naslednjih poglavjih.
Posamezne replike sinhronizirajo svoje podatke z vzpostavitvijo *sinhronizacijske seje* (sync session). Oba sinhronizacijska ponudnika sta povezana z *sinhronizacijskim agentom* (sync agent). Sinhronizacijski agent je zadolžen za vzpostavitev in upravljanje sinhronizacijske seje.
Sinhronizacija vedno poteka samo v eno smer – od *izvorne replike* (source replica) k *ponorni repliki* (destination replica). To lastnost sinhronizacije imenujemo sinhronizacijski tok (synchronization flow). Za primer vzemimo dve repliki A in B. Sinhronizacija lahko poteka v smeri od replike A k repliki B ali od replike B k repliki A. Nikoli ne pride do situacije, kjer bi sinhronizirali v obe smeri naenkrat. Ob vzpostavitvi seje s strani sinhronizacijskega agenta si repliki izmenjata svoje *znanje* (knowledge). Na podlagi znanja je možno določiti razlike med posameznima replikama in na pošiljanje pripraviti samo tiste podatke, ki v ciljni repliki niso vsebovani. Seveda lahko pri sinhronizaciji pride tudi do *konfliktov* (conflicts), ki jih je potrebno rešiti. Več informacij o razreševanju sinhronizacijskih konfliktov sledi v nadaljnjih poglavjih.

Na diagramu *slika 8* sta vidna še dva gradnika, ki ju do sedaj nisem omenil. Prvi je *MSF runtime*, ki predstavlja izvajalno okolje za sinhronizacijsko ogrodje. Naprava, ki želi sinhronizirati podatke z uporabo MSF sinhronizacijskega ogrodja, mora imeti nameščene runtime knjižnice.
Drugi neomenjen gradnik pa predstavlja *skladišče metapodatkov* oziroma *metapodatkovno skladišče* (metadata store). V tem skladišču so shranjeni vsi podatki o trenutnem stanju replike. Vsaka replika ima svoje metapodatke. Brez teh metapodatkov sinhronizacija ne bi bila mogoča. Preostanek tega poglavja govori o metapodatkih in njihovi uporabi v MSF.

Kaj pravzaprav so metapodatki? Metapodatki so po definiciji podatki o podatkih. Često je namreč potrebno hraniti podatke, ki sami po sebi niso iskane informacije, a vendar dodatno obogatijo oziroma klasificirajo željen podatek. Preprost primer je recimo digitalna fotografija. Metapodatki o tej fotografiji pa so lahko datum fotografiranja, odprtost zaslonke, tip in model objektiva oziroma fotoaparata, ipd. Podobno velja tudi za hrambo podatkov in izmenjavo le-teh. MSF deluje na podlagi metamodela, ki predstavlja ogrodje za sinhronizacijo podatkov [14].

4.3 Metapodatkovne komponente

4.3.1 Razčlemba metapodatkov

Zaradi širokega spektra podatkovnih skladišč, se poraja vprašanje o združljivosti. Če želimo sinhronizirati podatke nas običajno zanimajo naslednje postavke:

- Tip podatka/podatkov
- Tip podatkovnega skladišča
- Prenosni protokoli
- Topologija omrežja

Praviloma so zgornja vprašanja na mestu. Ob uporabi sinhronizacijskega ogrodja, v našem primeru MSF, pa so ta vprašanja odveč. Ogrodja so grajena na principu skupnega metamodela, ki omogoča sinhronizacijo popolnoma različnih podatkovnih skladišč.
Cilj tovrstnih sinhronizacijskih ogrodij je:

- Povečati interoperabilnost (angl. Interoperability)
- Zmanjšati količino informacij, ki se prenašajo ob sinhronizaciji
- Neodvisnost od topologije omrežja, podatkovnih tipov, podatkovnih skladišč in prenosnih protokolov.

MSF metapodatke deli v dve kategoriji:

1. Metapodatki o elementu (item metadata)
2. Metapodatki o repliki (replica metadata)

Metapodatki o repliki se nanašajo na samo podatkovno skladišče. Najbolj preprost primer metapodatka o podatkovnem skladišču je njegovo unikatno ime. Običajno uporabljamo globalno enoličen identifikator ali GUID (angl. Globally Unique IDentifier), ki omogoča razlikovanje med sinhronizacijskimi izvori in ponori.

Metapodatki o elementih so metapodatki o zapisih v dejanskem skladišču. Primer takšnega metapodatka je datum in čas zadnje spremembe določenega zapisa. Z uporabo ravnokar zapisanega je moč ugotoviti, da je zelo pomembna granularnost metapodatkov oziroma nivo podrobnosti beleženja sprememb. V relacijskih podatkovnih bazah je torej *element* tabela ali zapis v tabeli. Če se pokaže potreba po beleženju sprememb na nivoju atributov v zapisu pa je možno implementirati tudi takšno metapodatkovno skladišče.
Kvaliteta in nasploh prisotnost metapodatkov je ključnega pomena pri razreševanju konfliktov ob sinhronizaciji.

Do sedaj smo že velikokrat omenili metapodatke, nikoli pa nismo natančno definirali, kje se ti podatki nahajajo. V kontekstu MSF se podatki lahko nahajajo praktično kjerkoli:

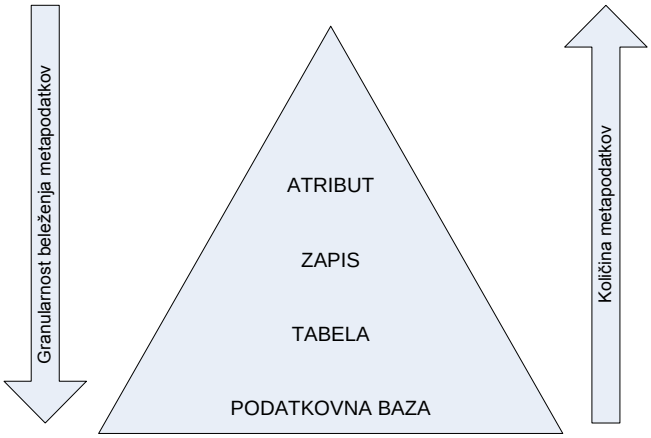
- V datoteki,
- V ločeni podatkovni bazi,
- V repliki.

Paziti je potrebno le na to, da je metapodatkovno skladišče programsko dosegljivo in da omogoča branje, spreminjanje (novi zapisi in spreminjanje obstoječih) in brisanje.
MSF je namreč zasnovan tako, da metapodatkovno skladišče upravlja oziroma posodablja preko sinhronizacijskih ponudnikov. Vmesniki za sinhronizacijske ponudnike so vnaprej določeni, zato je njihovo strukturo potrebno upoštevati, ko načrtujemo metapodatkovno skladišče. Urejanje in posodabljanje metapodatkov je običajno dolžnost sinhronizacijskega ponudnika. Kljub temu MSF poleg popolnoma prilagodljivega sistema za hrambo metapodatkov ponuja tudi svojo implementacijo, ki je realizirana znotraj SQL Server Compact Edition (SQL CE). Ta “vgrajena” implementacija močno olajša in pohitri razvoj.
Odločitev o tem, kakšen tip metapodatkovnega skladišča bo uporabljen, leži na plečih razvijalcev. Običajno se upošteva pravilo spremenljivosti. Če je podatkovno skladišče, ki je vključeno v sinhronizacijo, obsežno oziroma je verjetnost da se shema podatkovnega skladišča spremeni, potem implementiramo lastno metapodatkovno skladišče, ker ga je lažje programsko spreminjati. Če pa je shema podatkovnega skladišča relativno “zabetonirana”, lahko uporabimo metapodatkovno skladišče, ki je vgrajeno v SQL CE.

Kot je bilo v tem razdelku razloženo, se metapodatki delijo na metapodatke o elementih ter metapodatke o replikah. Komponenta, ki beleži podatke o elementih se imenuje v*erzija* (angl. version). Komponenta, ki beleži podatke o repliki se imenuje *Znanje* (angl. knowledge). Za delovanje MSF je potrebno hraniti še nekatere druge metapodatke, katerih vsebina je razložena v naslednjih razdelkih [14].

4.3.2 Verzija

Verzije beležijo kdaj in kje je bil nek zapis generiran oziroma spremenjen. Vsak zapis beleži svoje metapodatke o verzijah. Na tem mestu velja opozoriti, da se podatki o verziji beležijo v odvisnosti od granularnosti. To pomeni, da če sinhroniziramo na nivoju atributov, za vsak atribut beležimo metapodatke o verziji. Običajno temu ni tako, saj je potrebno upoštevati dejstvo da se na vsak shranjen atribut shranjuje precej metapodatkov. Vsebino teh metapodatkov bomo razdelali v nadaljevanju.



Slika 9: Primerjava granularnosti beleženja metapodatkov

Vzemimo za primer, da sinhroniziramo dve relacijski podatkovni bazi. Imenujmo ju replika A in replika B. V vsaki izmed replik se nahaja tabela *Stranka*. Shema tabele *Stranka* je prikazana na sliki Slika 10. Za namene tega primera bomo privzeli, da je element replike zapis (vrstica) v tabeli. Brez problemov lahko granularnost povečamo in kot element replike proglasimo kar celotno tabelo ali morda kar celotno podatkovno bazo (vse tabele). Granularnost bi lahko tudi zmanjšali ter tako kot element replike obravnavali atribut (stolpec). Granularnost je ponazorjena na sliki 9.

Column Name	Data Type	Allow Nulls
Stranka_ID	bigint	☐
Stranka_Naziv	varchar(150)	☐
Stranka_Naslov	varchar(150)	☑
Stranka_Telefon	varchar(30)	☑

Slika 10: Shema tabele *Stranka*

Denimo, da na začetku v tabeli *Stranka* v repliki A in v repliki B ni nobenih podatkov. Odločimo se, da bomo vnesli dve novi stranki. V tabeli *Stranka* v repliki A so po vnosu novih strank naslednji podatki (glej sliko 11).

	Stranka_ID	Stranka_Naziv	Stranka_Naslov	Stranka_Telefon
1	1	Stranka 1	Naslov 1	+ 386 1 234 56 78
2	2	Stranka 2	Naslov 2	+ 386 41 200 300

Slika 11: Stanje tabele *Stranka* v replik A po začetnem vnosu

V tabelo *Stranka* v repliki B vnesemo še eno stranko. Vsebina tabele je prikazana spodaj (glej sliko 12).

	Stranka_ID	Stranka_Naziv	Stranka_Naslov	Stranka_Telefon
1	10	Stranka 3	Naslov 3	+386 2 111 22 33

Slika 12: Stanje tabele *Stranka* v repliki B po začetnem vnosu

Na tem mestu je potrebno omeniti še razločevanje posameznih elementov v podatkovni bazi. Ker smo se za namene tega primera odločili, da bo element sinhronizacije predstavljal zapis v tabeli, moramo na nivoju zapisa tudi zagotoviti enolično identifikacijo posameznih

zapisov. To pomeni, da morajo imeti vsi zapisi v tabeli svoj enolični identifikator. V našem primeru je to polje Stranka_ID. V realnosti oziroma v produkcijskih okoljih pa je lahko enolični identifikator zapisa v tabeli sestavljen iz več različnih atributov (recimo ob povezavah M:N v podatkovni bazi). Ne glede na to, kakšen enolični identifikator uporablja podatkovna baza, moramo za namene sinhronizacije tudi uporabiti enolični identifikator. Ta identifikator imenujemo *ID metapodatkovnega elementa* (angl. Item ID). V našem primeru bomo zaradi enostavnosti privzeli, da je ta vrednost kar enaka enoličnemu identifikatorju. Tabela 2 prikazuje trenutne vrednosti verzij metapodatkov za tabelo *Stranka* v repliki A, tabela 3 pa trenutne vrednosti verzij metapodatkov za tabelo *Stranka* v repliki B.

Tabela 2: Trenutne vrednosti verzij za tabelo *Stranka* (replika A)

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
1	1	1	A	A
2	2	2	A	A

Tabela 3: Trenutne vrednosti verzij za tabelo Stranka (replika B)

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
10	3	3	B	B

Kot je razvidno iz zgornjih tabel, verzije beležijo kdaj in kje je bil nek zapis spremenjen oziroma ustvarjen. Verzija si "zapomni" tudi ID metapodatkovnega elementa, zato da ob sinhronizaciji vemo kateri zapis je prazaprav (ob morebitnih spremembah) potrebno prenašati.

Verzije kategoriziramo v dve smeri:

- *Verzija nastanka* (angl. creation version)
- *Verzija posodobitve* (angl. update version)

Prva beleži kdaj je bil element ustvarjen ,druga pa kdaj je bil nazadnje osvežen. Opazimo tudi, da sta čas nastanka in čas zadnje posodobitve ob nastanku zapisa enaka. Čas nastanka se kasneje ne spreminja več, čas zadnje posdobitve pa se osveži ob vsaki spremembi zapisa.

Obe verziji sestojita iz dveh podatkov:

- *Števca* (ang. tick count)
- *ID replike* (angl. replica ID)

Števec je pravzaprav število, ki zabeleži čas, kdaj je bil element nazadnje posodobljen oziroma ustvarjen. Ob vsaki spremembi elementa se to polje posodobi in s tem omogoča sledenje spremembam.

Naslednji korak tega primera je posodobitev enega zapisa v repliki A. Spremenili bomo vrednost atributa *Stranka_Naslov* za stranko z nazivom "Stranka 1". Nove vrednosti zapisov se nahajajo na sliki 13.

	Stranka_ID	Stranka_Naziv	Stranka_Naslov	Stranka_Telefon
1	1	Stranka 1	Nov naslov	+ 386 1 234 56 78
2	2	Stranka 2	Naslov 2	+ 386 41 200 300

Slika 13: Stanje tabele Stranka v repliki A po posodobitvi zapisa

Zaradi spremembe atributa v zapisu je potrebno zabeležiti spremembe tudi v metapodatkovnem skladišču ter posodobiti metapodatke za ta zapis. Tabela 4 prikazuje novo stanje verzij metapodatkov za isto tabelo.

Tabela 4: Nove vrednosti verzij za tabelo *Stranka* (replika A)

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
1	1	4	A	A
2	2	2	A	A

Proces spremljanja sprememb ter njihovo evidentiranje v metapodatkovnem skladišču imenujemo *spremljanje posodobitev* (angl. change tracking). Tega procesa ne smemo zamenjati z *dnevnikom sprememb* (angl. change log). Dnevnik sprememb namreč omogoča pogled v zgodovino sprememb. Z drugo, morda bolj domačo besedno zvezo, ta dnevnik imenujemo tudi revizijska sled. Pri spremljanju posodobitev ne beležimo zgodovine sprememb, temveč le trenutno veljavno vrednost. Spodnji primer razlaga razliko med spremljanjem posodobitev ter zgodovino sprememb.

PRIMER

V podatkovni bazi X v ustvarimo tabelo T. Tabela ima dva atributa:

- ID int; predstavlja primarni ključ tabele
- Naziv varchar(150)

Predpostavimo, da metapodatke beležimo avtomatsko na nivoju podatkovne baze.

V tabelo T vnesemo tri zapise z vrednostmi (ID, Naziv):

- 1, "naziv 1"
- 2, "naziv 2"
- 3, "naziv 3"

Posodobimo atribut Naziv tam, kjer je atribut ID enak 2 in vnesemo vrednost "Nova vrednost".

Pri spremljanju posodobitev nas zanimjo le časi nastankov ter časi posodobitev zapisov.

Končni metapodatki za tabelo T so prikazani v tabeli 5.

Tabela 5: Končni metapodatki za tabelo T

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
1	1	1	X	X
2	2	4	X	X
3	3	3	X	X

Dnevnik sprememb je prikazan v tabeli 6.

Tabela 6: Dnevnik sprememb za tabelo T

ID spremembe	Vrsta spremembe	Vrednost ID	Vrednost Naziv
1	Vnos (insert)	1	Naziv 1
2	Vnos (insert)	2	Naziv 2
3	Vnos (insert)	3	Naziv 3
4	Posodobitev (update)	2	Nova vrednost

Pri spremljanju posodobitev je pomembno, da sinhronizacijski ponudnik posodobi podatke takoj ko se spremembe pojavijo. V nasprotnem primeru lahko pride do neljubih situacij, kjer se sinhronizacijski tok ne odvije tako kot je bilo načrtovano ter pride do konfliktov.

MSF pozna dva načina za beleženje metapodatkov – *takojšnje* oziroma *sinhrono* (angl. inline tracking) ter*zakasnjeno* oziroma *asinhrono* (angl. asynchronous tracking).
Pri takojšnjembeleženju se metapodatki posodobijo takoj, ko se v repliki zgodi sprememba. Običajno se uporablja takrat, ko so metode, ki dejansko spreminjajo podatke, implementirane na način, ki omogoča relativno preprosto nadgradljive s postopkom za posodobitev metapodatkov. Primer takšnega scenarija je sinhronizacija podatkovnih baz. Metapodatke lahko namreč osvežimo s pomočjo prožilcev.
Zakasnjeno beleženje, kot pove že ime, beleži oziroma posodablja metapodatke *kasneje*. Kaj to pravzaprav pomeni? Z uporabo zunanje storitve ali procesa lahko preiščemo repliko in najdemo spremembe v repliki. Na podlagi rezultatov iskanja sprememb ustrezno modificiramo metapodatke replike. Takšen pristop je lahko zelo uporaben pri sistemih, kjer bi bila uvedba takojšnjega beleženja sprememb *predraga*. Uvedba sistema za takojšnje beleženje metapodatkov je predraga takrat, kadar je implementacija preveč kompleksna oziroma časovno potratna ali ko imamo opravka z aplikacijami ki so procesorsko zahtevne ali se spopadamo s časovno kritičnimi aplikacijami.

Bistvena lastnost obeh metod je granularnost beleženja. Le-ta mora biti vsaj na nivoju metapodatkovnega elementa. Za lažje razumevanje glej sliko 9, kjer je bilo že govora o granularnosti [14].

4.3.3 Znanje

Znanje je kompaktna oblika verzije oziroma sprememb, ki jih določena replika pozna. Nekatere spremembe na zapisih so lahko poznane samo določenim replikam, ostale replike pa se teh sprememb ne zavedajo. Do takšnih stanj prihaja pogosto, rešujemo pa jih s sinhronizacijo na podlagi znanja.
Za razliko od verzij, ki se nanaša na metapodatkovne elemente, je znanje metapodatek o repliki. Znanje vsebuje spremembe:

- ki so nastale v repliki (neposredne),
- ki so nastale preko sinhronizacije (posredne).

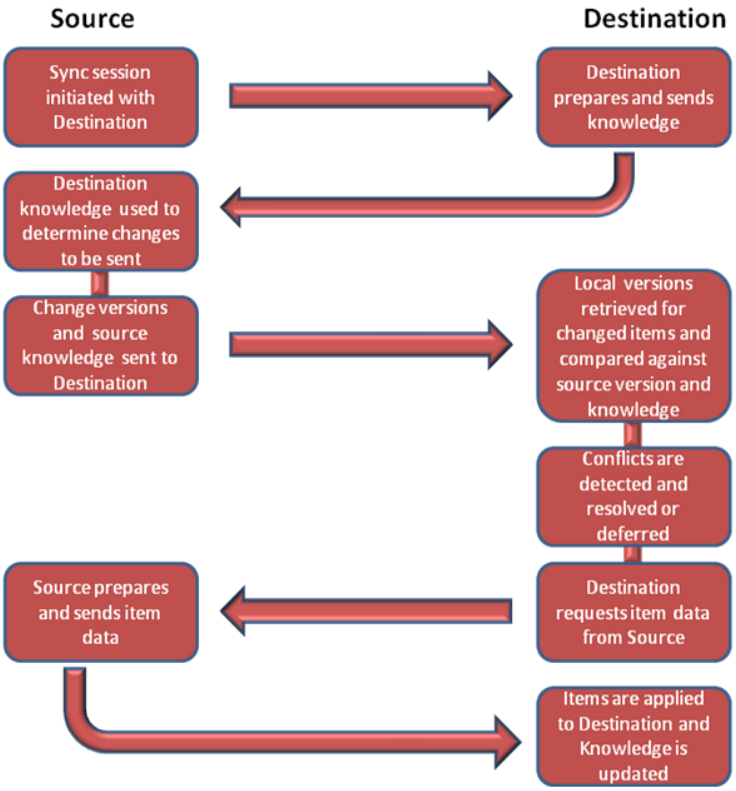
Ponudniki sinhronizacije uporabljajo znanje za enumeracijo sprememb ter zaznavanje konfliktov. Več o enumeraciji sprememb in zaznavanju konfliktov bomo govorili v poglavju 3, za sedaj pa le na kratko definirajmo da:

- je enumeracija sprememb določi spremembe, ki so se zgodile na repliki, za katere ostale replike ne vedo,
- zaznavanje konfliktov določi operacije, ki so se zgodile na repliki, za katere ostale replike ne vedo.

Če pogledamo tabele v prejšnjem razdelku lahko za repliki A in B določimo njuno znanje. Za repliko A rečemo da je njeno znanje po vnosu zapisov enako A2, ko pa na teh zapisih izvršimo spremembe znanje dobi vrednost A4. Replika B ima po vnosu podatkov znanje B3, ki ostane enako tudi po posodobitvi v repliki A, ker ni prišlo do sinhronizacije.
Znanje replike torej sestoji iz unikatnega imena replike (ID replike) ter zadnje vrednosti števca (tick count) [14].

4.3.4 Teoretični potek sinhronizacije

Potek sinhronizacije je najlažje razložiti na primeru. Repliki A in B, ki smo ju uporabili v prejšnjem razdelku bosta služili kot izvor in ponor sinhronizacije. Privzemimo, da želimo sinhronizacijo sprožiti od replike A k repliki B. Na sliki 14 je grafični prikaz sinhronizacijskega toka.



Slika 14: Potek sinhronizacije

Postopek sinhronizacije je sledeč:

1. Ponorna replika (replika B) pošlje svoje znanje izvorni repliki (replika A).
2. Izvorna replika z uporabo pridobljenega znanja ponorne replike poišče tiste zapise oziroma spremembe, ki se jih ponorna replika ne zaveda. Temu postopku pravimo *enumeracija sprememb* (angl. change enumeration).
3. Sinhronizacijski ponudnik izvorne replike na podlagi enumeracije sprememb zgradi *paket sprememb* (angl. change batch) ter ga pošlje ponorni repliki. To *niso* dejanski podatki temveč metapodatki (verzije). Primer paketa sprememb je v tabeli 7.

Tabela 7: Paket sprememb v smeri od replike A k repliki B

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
1	1	4	A	A
2	2	2	A	A

4. Po prejemu paketa sprememb ponorna replika izračuna ali je prišlo do konflikta. Ta postopek se imenuje *detekcija konfliktov* (angl. conflict detection). Do konflikta pride takrat, kadar različni repliki spreminjata zapis z istim identifikatorjem (ID elementa).
5. Ponorna replika zahteva dejanske podatke, ki jih izvorna replika pošlje. Poslani so podatki o Stranki 1 in Stranki 2. Ponorna replika podatke shrani in osveži svoje znanje. Stanje replike B po zaključenem koraku 5 je prikazano na sliki 15.

	Stranka_ID	Stranka_Naziv	Stranka_Naslov	Stranka_Telefon
1	10	Stranka 3	Naslov 3	+ 386 2 111 22 33
2	1	Stranka 1	Nov naslov	+ 386 1 234 56 78
3	2	Stranka 2	Naslov 2	+ 386 41 200 300

Slika 15: Stanje v tabeli *Stranka* v repliki B na polovici sinhronizacije

Stanje metapodatkovnega skladišča v repliki B po koraku 5 je prikazano v tabeli 8.

Tabela 8: Stanje metapodatkovnega skladišča v repliki B na polovici sinhronizacije

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
10	3	3	B	B
1	1	4	A	A
2	2	2	A	A

Na podlagi metakodatkov (tabela 8) lahko MSF izračuna znanje replike B, katerega vrednost je sedaj A4B3.

Po končanem koraku številka 5 je sinhronizacija opravljena polovično. Potrebno je sinhronizirati tudi podatke iz smeri replike B v smeri replike A. Sedaj ponorna replika postane replika A, izvorna pa replika B. Po končani sinhronizaciji imasta repliki enake podatke in enako znanje. Ko imata repliki enako znanje pravimo, da sta *sinhronizirani* (angl. in sync with each other). Na sliki 16 je stanje v tabeli *Stranka* po končani sinhronizaciji.

	Stranka_ID	Stranka_Naziv	Stranka_Naslov	Stranka_Telefon
1	1	Stranka 1	Nov naslov	+ 386 1 234 56 78
2	2	Stranka 2	Naslov 2	+ 386 41 200 300
3	10	Stranka 3	Naslov 3	+ 386 2 111 22 33

Slika 16: Stanje v tabeli *Stranka* v repliki A po končani sinhronizaciji

Tabela 9 prikazuje stanje v metapodatkovnem skladišču replike A po končani sinhronizaciji.

Tabela 9: Stanje metapodatkovnega skladišča v repliki A ob koncu sinhronizacije

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
1	1	4	A	A
2	2	2	A	A
10	3	3	B	B

Med samo sinhronizacijo seveda lahko pride tudi do konfliktov, kot je opisano zgoraj v koraku številka 4. V nadaljevanju primera bomo simulirali nastanek konflikta. V obeh

replikah spremenimo vrednost atributa *Stranka_Naziv* za zapisa, ki imata vrednost atributa *Stranka_ID* enako 1. Najprej spremenimo vrednost v repliki A, kasneje pa še v repliki B. Stanje tabel *Stranka* v obeh replikah je prikazano na slikah 17 in 18.

	Stranka_ID	Stranka_Naziv	Stranka_Naslov	Stranka_Telefon
1	1	Stranka 1-Replika A	Nov naslov	+ 386 1 234 56 78
2	2	Stranka 2	Naslov 2	+ 386 41 200 300
3	10	Stranka 3	Naslov 3	+ 386 2 111 22 33

Slika 17: Stanje v tabeli *Stranka* v repliki Apo spremembi

	Stranka_ID	Stranka_Naziv	Stranka_Naslov	Stranka_Telefon
1	10	Stranka 3	Naslov 3	+ 386 2 111 22 33
2	1	Stranka 1-Replika B	Nov naslov	+ 386 1 234 56 78
3	2	Stranka 2	Naslov 2	+ 386 41 200 300

Slika 18: Stanje v tabeli *Stranka* v repliki B po spremembi

Stanje metapodatkovnih skladišč v obeh replikah prikazujeta spodnji tabeli.

Tabela 10: Stanje metapodatkovnega skladišča za tabelo *Stranka* v repliki A

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
1	1	5	A	A
2	2	2	A	A
10	3	3	B	B

Tabela 11: Stanje metapodatkovnega skladišča za tabelo *Stranka* v repliki B

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
10	3	3	B	B
1	1	6	A	B
2	2	2	A	A

Znanje replike A je v tem trenutku A5B3, znanje replike B pa je A4B6. V postopku sinhronizacije replika A ugotovi, da replika B ne pozna spremembe, ki se je zgodila ob števcu 5 v repliki A (A5). V tabeli 12 se nahaja paket sprememb, ki ga replika A pošlje repliki B.

Tabela 12: Paket sprememb replike A

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v

1	1	5	A	A

Ko replika B prejme paket sprememb tudi sama enumerira spremembe, ki jih bo poslala repliki A. V tabeli 13 se nahaja paket sprememb, ki jih replika B enumerira za repliko A.
Ob tej enumeraciji replika B ugotovi, da je bil zapis 1 (ID elementa = 1) spremenjen v obeh replikah, kar povzroči konflikt.

Tabela 13: Paket sprememb replike B

ID elementa	Kdaj		Kje	
	Čas nastanka	Čas zadnje posodobitve	Nastalo v repliki	Nazadnje posodobljeno v
1	1	6	A	B

Ponudnik sinhronizacije je odgovoren za razrešitev oziroma obravnavo konfliktov. Obstaja več različnih možnosti razreševanja:

- *Izvor prevlada* (angl. source wins): ohrani se sprememba, ki je bila vnešena v izvorni repliki
- *Ponor prevlada* (angl. destination wins): ohrani se sprememba, ki je bila vnešena v ponorni repliki
- *Preskoči spremembi* (angl. skip change): konflikt se ignorira in se ga razrešuje kasneje
- *Združi spremembi* (angl. merge change): obe spremembi se združita
- *Shrani konflikt* (angl. save conflict): konflikt se shrani za kasnejšo obravnavo [14]

4.3.5 Obvezni metapodatki

MSF za svoje delovanje potrebuje metapodatke o elementu in repliki. Poleg vrste metapodatkov, MSF zahteva upoštevanje tudi nekaterih pravil, ki se nanašajo na dejansko hrambo podatkov. Kot povzetek prešnjih treh razdelkov naštejmo obvezne metapodatke za replike in metapodatkovne elemente [14].

4.3.5.1 Replika

Metapodatki o repliki, katerih hrambo zapoveduje MSF so:

1. *ID replike* (angl. replica ID)
2. *Trenutna vrednost števca* (angl. current tick count)
3. *Mapiranje replike* (angl. replica key map): ID replike se znotraj dejanske implementacije metapodatkovnega skladišča pojavi večkrat. Zaradi smotrnosti pri porabi prostora, ID replike po metapodatkovnem skladišču “prenašamo” s pomočjo mapiranja. ID replike shranimo v posebno tabelo, le-to pa v preostalih delih metapodatkovnega skladišča naslavljamo s 4-bitnim ključem.
4. *Trenutno znanje* (angl. current knowledge)
5. *Pozabljeno znanje* (angl. forgotten knowledge): Tu hranimo podatke o izbrisanih elementih, za katere druge replike ne vedo.

6. *Dnevnik konfliktov* (angl. conflict log): Tu shranimo podatke o nastalih konfliktih. Replike imajo "proste roke" pri zasnovi hrambe teh podatkov, edina zahteva glede hrambe teh podatkov je, da MSF lahko do njih dostopa ter jih ustrezno enumerira.
7. *Dnevnik nagrobnikov* (angl. tombstone log): Tu hranimo podatke o izbrisanih elementih, za katere vedo tudi druge replike [14].

4.3.5.2 Metapodatkovni element

Metapodatki o elementih, katerih hrambo zapoveduje MSF so:

1. *Globalni enolični identifikator* (angl. global item ID): S pomočjo tega metapodatka lahko nek zapis enolično identificiramo.
2. *Trenutna verzija zapisa* (angl. current version)
3. *Verzija nastanka* (angl. creation version) [14]

4.4 Microsoft Storage Service – MSS

V prejšnjih podpoglavjih smo že omenili, da se MSF močno zanaša na skupni podatkovni metamodel. To v praksi pomeni, da morajo vse strani, vpletene v sinhronizacijo, upoštevati nekatera pravila, ki jih ta metamodel zapoveduje. Kot je že razvidno iz slike 8, je replika tesno povezana z izvajalnim okoljem in sinhronizacijskim ponudnikom. Podobno je z izvajalnim okoljem in sinhronizacijskim ponudnikom povezano metapodatkovno skladišče. MSF s svojim metamodelom predpisuje le programski vmesnik (angl. application programming interface – API). Pri dejanski implementaciji metapodatkovnega skladišča pa razvijalcem pušča popolnoma proste roke.

MSS razvijalcem pomaga upoštevati zahtevan metamodel in je načrtovan tako, da poskrbi za večino podrobnosti povezanih s hrambo, pridobivanjem in urejanjem metapodatkov. Model je zasnovan tako, da se razvijalci osredotočajo na dejanski problem – vsebinsko sinhronizacijo podatkov – brez, da bi se v ozadju "borili" še s problemi, ki jih prinašajo hramba, pridobivanje in urejanje metapodatkov.
Poleg vpeljave čim večje mere abstrakcije na nivoju manipulacije z metapodatki, MSS nudi tudi jasno ločnico med metapodatkovnim skladiščem in programskimi vmesniki, ki iz prej omenjenega metapodatkovnega skladišča pridobivajo podatke. Ta lastnost je pomembna, saj se nemalokrat zgodi, da se razvijalci odločijo za menjavo tipa metapodatkovnega skladišča med samim razvojem oziroma ko je aplikacija že vpeljana (npr. zaradi pohitritve). Metapodatkovno skladišče se običajno razvija "na roke" takrat, ko replika sama ni zmožna beležiti sprememb. Tu so predvsem mišljeni tekstovni podatkovni viri kot so prosti tekst, XML in CSV datotke [14].

4.4.1 Metapodatkovno skladišče vgrajeno v SQL Server CE

Velikokrat je vpeljava lastnega metapodatkovnega skladišča potrata časa. Že v prejšnjih poglavjih smo omenili implementacijo metapodatkovnega skladišča znotraj SQL Server CE. Vsi podatki in metapodatki so tako shranjeni na enem mestu – v MS SQL Server CE

podatkovni bazi. MS SQL Server CE je enostavna podatkovna baza, ki je idealna za razvoj *občasno povezanih aplikacij* (angl. Occasionally Connected Application/s – OCA/OCAs) in samostojnih aplikacij, ki za svoje delovanje potrebujejo manjšo količino relacijskih podatkov [5].

Podatkovna baza SQL Server CE ima naslednje prednosti oziroma lastnosti:

- Je zastonj.
- Je relacijska.
- Jo je lahko razpečevati.
- Podpira tako namizne računalnike kot mobilne naprave.
- Je *vgradljiva* (angl. embeddable) – primerno za samostojne aplikacije.
- Je kategorizirana kot *lahka* (angl. lightweight), saj za svoje delovanje ne potrebuje veliko sredstev – ima majhen *odtis* (angl. footprint).
- Ne zahteva strokovne administracije s strani IT profesionalcev.
- Celotna baza je v eni sami datoteki.
- Podpira različne opcije namestitve (ClickOnce, Xcopy, MSI, CAB,...).
- Podpira vse Microsoftove namizne, strežniške in mobilne operacijske sisteme.
- Podpira precejšenj odstotek nabora ukazov Transact-SQL (T-SQL).
- Podpira ADO.NET, LINQ to SQL, LINQ to Entities in ADO.NET Entity Framework.
- Podpira več hkratnih povezav.
- Je integrirana v razvojno orodje MS Visual Studio 2008 in MS Visual Studio 2010.
- Ima vgrajen varnostni mehanizem.
- Je razširljiva – v primeru dodajanja novih atributov oziroma tabel [15, 16].

4.4.2 Lokacija metapodatkovnega skladišča

MSF omogoča delo brez povezave ali skupinsko delo (angl. collaboration). Vpeljava sinhronizacije je mogoča tako rekoč za vsako vrsto aplikacije, naprave in tipa skladišča metapodatkov. Ko izberemo način implementacije metapodatkovnega skladišča (skladišče po meri ali MS SQL Server CE), moramo sprejeti še odločitev o tem, kje se bo to metapodatkovno skladišče dejansko nahajalo. Običajno drži, da se metapodatki nahajajo *blizu* dejanskih podatkov. Pod pojmom *blizu* imamo v mislih vsaj isto napravo, če ne celo točno določeno mesto na trdem disku. Takšno razmišljanje je precej običajno in zdrži večino scenarijev, ne pa vseh. Metapodatki se namreč lahko nahajajo praktično kjerkoli. Odločitev o tem, kje se bodo metapodatki nahajali, je seveda odvisna od razvijalcev, a je hkrati pogojena tudi s tipom naprave, ki sinhronizacijo opravlja. MSF kategorizira naprave v tri različne skupine:

a. *Polno sodelujoče* (angl. full participants)
b. *Delno sodelujoče* (angl. partial participants)
c. *Preproste sodelujoče* (angl. simple participants)

4.4.2.1 Polno sodelujoče naprave

Polno sodelujoče naprave so naprave, ki lahko ustvarjajo nova podatkovna skladišča in izvajajo aplikacije na napravi sami. Primeri takšnih naprav so osebni računalniki, strežniki, prenosni računalniki, tablični računalniki in pametni telefoni. Polno sodelujoče naprave lahko

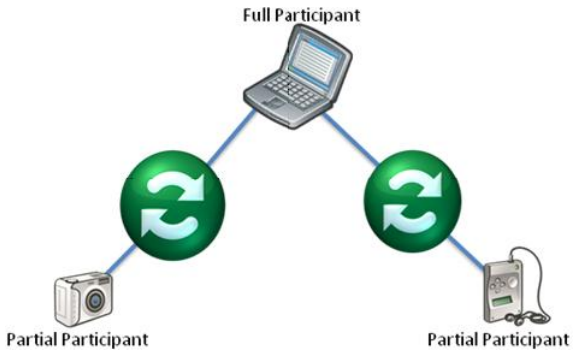
hranijo metapodatke. Slika 19 predstavlja zmožnost sinhronizacije med polno sodelujočimi napravami.

Slika 19: Sinhronizacija med polno sodelujočimi napravami

Katerakoli izmed polno sodelujočih naprav lahko hrani metapodatke tudi za drugo napravo. To je na primer koristno ob sinhronizaciji strežnika in mobilne naprave, saj na mobilni napravi morda zaradi kapacitivnih omejitev nočemo hraniti preveč podatkov.

4.4.2.2 Delno sodelujoče naprave

Podobno kot polno sodelujoče naprave, tudi delno sodelujoče naprave lahko ustvarjajo nova podatkovna skladišča, za razliko od polno sodelujočih pa ne morejo izvajati poljubnih aplikacij. Primeri takšnih naprav so recimo USB ključi, digitalni fotoaparati ali MP3 predvajalniki.



Slika 20: Sinhronizacija med polno sodelujočimi in delno sodelujočimi napravami

Delno sodelujoče naprave lahko svoje podatke sinhronizirajo samo z polno sodelujočimi napravami. Metapodatkovno skladišče se sicer lahko nahaja na delno sodelujočih napravah, vendar zaradi nezmožnosti manipuliranja z metapodatki (ker ne morejo izvajati poljubnih aplikacij) običajno metapodatke hranimo na polno sodelujočih napravah, ker so hitreje dostopni.

4.4.2.3 Preproste sodelujoče naprave

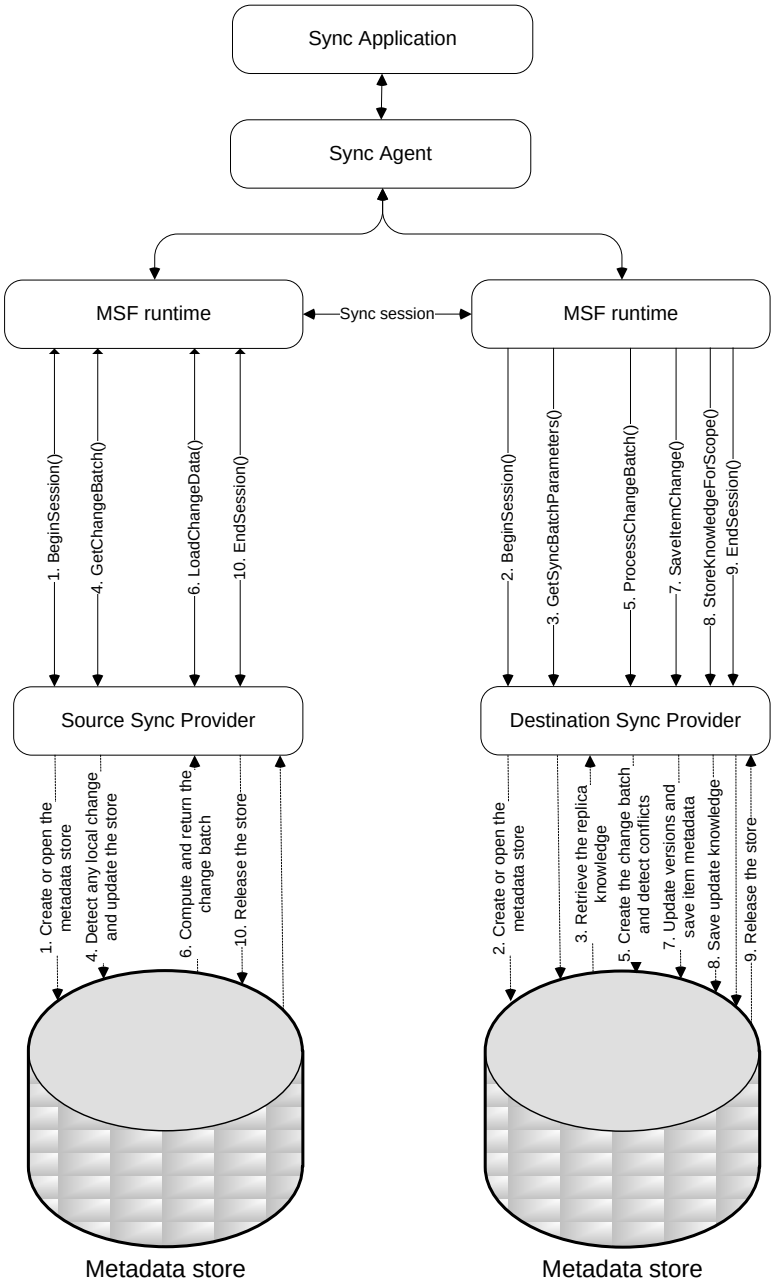
Preproste sodelujoče naprave so tiste naprave, ki ne morejo ustvarjati novih podatkovnih skladišč in ne morejo izvajati poljubnih aplikacij. Običajno samo ponujajo informacije na zahtevo. Primer preproste sodelujoče naprave je Really Simple Syndication (RSS).

Slika 21: Sinhronizacija med preprosto sodelujočo napravo in polno sodelujočo napravo

Preproste sodelujoče naprave lahko podatke dobivajo le preko sinhronizacije s polno sodelujočo napravo. Razlog tiči v tem, da so metapodatki vedno shranjeni na polno sodelujoči napravi [8].

4.4.3 Komunikacija med sinhronizacijskim ponudnikom in MSS

Slika 22 predstavlja način komunikacije med MSF, sinhronizacijskim ponudnikom in metapodatkovnim skladiščem. Gradniki med seboj komunicirajo po točno določenem vzorcu z uporabo API-jev, ki smo jih omenili že v poglavju 4.3.4 (slika 14).



Slika 22: Komunikacija med sinhronizacijskim ponudnikom in MSS

Sinhronizacijska aplikacija sproži proces sinhronizacije s klicem metode *Synchronize()*, ki se nahaja v sinhronizacijskem agentu. Med replikama se vzpostavi sinhronizacijska seja in sinhronizacija se prične.
Naslednji koraki opisujejo tok metapodatkov v sinhronizacijski seji:

1. Izvajalno okolje kliče metodo *BeginSession()* na izvornem sinhronizacijskem ponudniku (to je ponudnik, ki pripada izvorni repliki). MSS ob klicu te metode zagotovi, da metapodatkovno skladišče obstaja in je inicializirano.
2. Ta korak je enak koraku 1, le da se izvaja na ponornem sinhronizacijskem ponudniku.
3. Izvajalno okolje kliče metodo *GetSyncBatchParameters()* na ponornem sinhronizacijskem ponudniku. MSS ponorne replike pridobi podatek o svojem znanju ter ga posreduje izvornemu sinhronizacijskemu ponudniku.
4. Na izvorni repliki izvajalno okolje kliče metodo *GetChangeBatch()*. Ta metoda zazna lokalne spremembe in v primeru le-teh osveži metapodatke. Ponorni repliki se pošlje razlika med lokalnim znanjem in prejetim znanjem. To so spremembe, ki so se zgodile v izvorni repliki in jih ponorna replika ne pozna.
5. V ponorni repliki izvajalno okolje kliče metodo *ProcessChangeBatch()*. MSS na podlagi ravnokar prejetega znanja izvorne replike in sprememb le-te (iz koraka 4) poišče lokalne spremembe v ponorni repliki ter zazna morebitne konflikte. Ko so konflikti razrešeni, ponorni sinhronizacijski ponudnik pošlje zahtevo za pošiljanje dejanskih podatkov iz izvorne replike.
6. Izvajalno okolje na izvorni repliki prejme zahtevo za pošiljanje podatkov in kliče metodo *LoadChangeData()*. Ta metoda pripravi lokalne podatke in jih pošlje ponorni repliki.
7. Izvajalno okolje na ponorni repliki kliče metodo *SaveItemChange()*, ki shrani ravnokar prejete podatke. MSS tudi osveži verzije v metapodatkovnem skladišču in vanj shrani morebitne nove zapise.
8. Izvajalno okolje kliče metodo *StoreKnowledgeForScope()* na ponornem sinhronizacijskem ponudniku, ki shrani svoje novo znanje v metapodatkovno skladišče.
9. Izvajalno okolje ponorne replike kliče metodo *EndSession()*, ki sprosti ponorno metapodatkovno skladišče.
10. Izvajalno okolje izvorne replike kliče metodo *EndSession(),* ki sprosti izvorno metapodatkovno skladišče.

Na tem mestu je seveda potrebno poudariti, da ima lahko vsaka replika svoj tip metapodatkovnega skladišča [8, 14].

4.5 Sinhronizacijski ponudniki – Sync providers

V prejšnjih poglavjih, so bili sinhronizacijski ponudniki večkrat omenjeni. Na tej točki vemo, da predstavljajo pomemben člen pri poteku same sinhronizacije, ne vemo pa podrobnosti o njihovih nalogah in zadolžitvah. V tem podpoglavju bomo opisali kaj pravzaprav sinhronizacijski ponudnik je. Razložena bo tudi uporaba sinhronizacijskih ponudnikov, ki so že vgrajeni v MSF. MSF je, tako kot pri metapodatkih in metapodatkovnih skladiščih, tudi pri sinhronizacijskih ponudnikih zelo odprt in omogoča lastno implementacijo. Za običajne scenarije, ko je sinhronizacija predvidena že od faze analize oziroma načrtovanja projekta, običajno zadostujejo vgrajeni sinhronizacijski ponudniki. V MSF najdemo tri vgrajene sinhronizacijske ponudnike:

- Sinhronizacijski ponudnik za podatkovna skladišča, ki so podprta v ADO.NET;
- Sinhronizacijski ponudnik za datoteke in mape
- Sinhronizacijski ponudnik za RSS in Atom

Kot je bilo že razloženo in prikazano na sliki 8, mora imeti vsaka imeti svojega sinhronizacijskega ponudnika, ki skrbi za urejanje metapodatkov, zapisovanje in posredovanje dejanskih podatkov ter komunikacijo z sinhronizacijskim agentom.

V naslednjih podpoglavjih bomo obravnavali sinhronizacijski ponudnike za datoteke in mape ter sinhronizacijske ponudnike za podatkovna skladišča, ki so podprta v ADO.NET [14].

4.5.1 Sinhronizacija datotek in map

4.5.1.1 Splošno

Vgrajen sinhronizacijski ponudnik za datoteke in mape deluje na NT in FAT datotečnih sistemih. Iz napisanega lahko sklepamo, da se tovrstna sinhronizacija lahko uporablja za sinhronizacijo datotek in map na lokalnem računalniku oziroma omrežju. Tipičen primer uporabe je npr. ustvarjanje varnostnih kopij. Sinhronizacijski ponudnik lahko uporabljamo tudi za sinhronizacijo izmenljivih medijev in USB ključev.

Sinhronizacijski ponudnik za datoteke in mape omogoča:

- Inkrementalno (delno) sinhronizacijo – Sinhronizacijsko ogrodje zazna spremembe v samih datotekah in mapah, tako da se vedno sinhronizirajo samo spremenjene datoteke in mape. Ta funkcionalnost je mogoča zato, ker ogrodje spremlja časovne žige in atribute datotek in map. Možno je tudi uporabljati *hash* datoteke, v primerih ko časovno žigosanje in atributi niso zanesljivi.
- Zaznavanje konfliktov.
- Filtriranje datotek in map, ki sodelujejo v sinhronizaciji na podlagi imena ali končnice.
- „*Predogled*“ oziroma „*Preview mode*“, s katerim lahko vidimo kako bi izgledal rezultat sinhronizacije, brez da se spremembe shranijo na disk.

V razredu ***SyncOrchestrator*** (sinhronizacijski agent) se nahaja atribut ***Direction***, ki določa smer sinhronizacije. Agent pozna štiri načine delovanja: ***Upload, Download, UploadAndDownload, DownloadAndUpload***, ki so dostopni preko *enumeracije* ***SyncDirectionOrder***.

Iz sinhronizacijskega postopka so avtomatično izpuščene datoteke *Thumbs.db*, *Desktop.ini* in *Metapodatkovne datoteke* [14, 17].

4.5.1.2 Zaznavanje sprememb na datotečnem sistemu

Pred samo sinhronizacijsko sejo, mora sinhronizacijski ponudnik pridobiti informacije o spremembah od zadnje sinhronizacije. Ta postopek se imenuje *zaznavanje sprememb* (angl. Change detection).
Sinhronizacijski ponudnik shrani metapodatke, ki opisujejo stanje vseh datotek in map. Spremembe so zaznane s primerjavo trenutnega stanja in zadnjega shranjenega stanja v metapodatkih. Postopek se sicer požene avtomatično, če pa ga želimo izvesti ročno, moramo poklicati metodo ***DetectChanges()*** v razredu ***FileSyncProvider***.

S pomočjo hevrističnega algoritma se išče preimenovanja datotek. Če je bila datoteka samo preimenovana se bo na ponorni repliki izvršilo samo preimenovanje. S tem se izognemo nepotrebnemu prenašanju datoteke.
V primeru, da je bila premaknjena mapa, se to obravnava kot brisanje in ponovno ustvarjanje mape. Datoteke znotraj mape se bodo, kljub drugi lokaciji, samo »preimenovale«.

Privzeto je, da sinhronizacijski ponudnik zazna spremembe pred vsako sinhronizacijsko sejo. V primeru, da je datotek veliko, je to lahko časovno potratno, še posebej če izvajamo iterativno sinhronizacijo med npr. desetimi terminali. Za takšne časovno zahtevne primere je možno sprožiti zaznavanje sprememb ročno, tako da se delo ne podvaja.

Med samo sinhronizacijo lahko prihaja do konfliktov (glej poglavje 4.5.1.5). V procesu razreševanja le-teh je možno, da je potrebno kakšno datoteko zbrisati. Takšna dejanja so permanentna. Sinhronizacijski ponudnik omogoča, da se izbrisane datoteke prestavijo v *koš (*angl. *recycle bin*) namesto da se permanentno zbrišejo. Takšno obnašanje sinhronizacijskega ponudnika dosežemo z uporabo razreda ***FileSyncOptions*** [14, 18].

4.5.1.3 Poročanje o napredku in način predogleda

Sinhronizacijski ponudnik omogoča poročanje o napredku med samo sinhronizacijo. Tako lahko interaktivne aplikacije implementirajo *indikator poteka* (angl. progress bar). Poročanje je realizirano preko *dogodkov* (angl. events). Z ročnim obravnavanjem dogodkov lahko razvijalec končnim uporabnikom omogoči izbiro o tem, ali bo neka datoteka oziroma mapa predmet sinhronizacije ali ne.

V primeru, da hočemo uporabiti funkcionalnost poročanja o poteku sinhronizacije, se moramo povezati na dogodke ***ApplyingChange, AppliedChange, CopyingFile*** in ***SkippedChange*** v razredu ***FileSyncProvider***.

Prva dva uporabimo za obveščanje o tem katera datoteka se sinhronizira in za kakšen tip sinhronizacije gre. Dogodek ApplyingChange se proži pred dejanskim prenosom. Na tem mestu se lahko odločimo in sinhronizacijo dotične datoteke preskočimo oziroma zavrnemo z uporabo atributa ***SkipChange*** v razredu ***ApplyingChangeEventArgs***.
Tretji dogodek (CopyingFile) se nanaša na datoteko, ki je trenutno v postopku sinhronizacije. Eden izmed parametrov dogodka je razred tipa ***CopyingFileEventArgs***. V njem najdemo atribut ***PercentCopied***, preko katerega lahko za dotično datoteko izvemo, koliko procentov prenosa je že opravljenega.
Četrti dogodek (SkippedChange) se proži, kadar pride do napake (premalo prostora na disku, onemogočen dostop,...) oziroma kadar je neka datoteka namenoma izpuščena iz sinhronizacije.[2]
Poznamo več različnih tipov sprememb. Do njih dostopamo preko enumeracije (angl. enumeration) ***ChangeType***. Na voljo imamo ustvarjanje, brisanje, posodabljanje, preimenovanje (angl. create, delete,update, rename). Ta podatek je bistvenega pomena zaradi optimalnega prenosa podatkov, koristen pa je tudi pri javljanju napredka uporabnikom.

Predogled sinhronizacije je nekakšen pogled na stanje, ki bi nastalo, če bi se podatki sinhronizirali. Gre za način delovanja, kjer se zažene le algoritem zaznave sprememb, katerega rezultate lahko prikažemo končnim uporabnikom. Predogled sinhronizacije lahko izvedemo z eno samo vrstico kode. Atribut ***PreviewMode*** v sinhronizacijskem ponudniku (razred ***FileSyncProvider***) nastavimo na ***true*** [14, 18].

4.5.1.4 Filtriranje datotek

Privzeto obnašanje sinhronizacijskega ponudnika je, da sinhronizira *vse*[3]. Preko posebnih atributov sinhronizacijskega ponudnika je možno nadzorovati, kaj se dejansko sinhronizira. To funkcionalnost podpira razred ***FileSyncScopeFilter***. Izbira datotek je možna na nivoju imen, map in podmap ter atributov datotek. Dovoljena je tudi uporaba *nadomestnih znakov (*angl. wildcards). V sinhronizaciji sodelujejo samo datoteke in mape, ki ustrezajo *vsem* filtrom. V spodnji tabeli so prikazani atributi prej omenjenega razreda za filtriranje datotek [14, 18].

Tabela 14: Atributi za filtriranje

Tip filtra	Privzeta vrednost	Primer
FilenameExcludes	Izključena ni nobena datoteka	Sinhroniziraj vse razen datotek: test.txt, *.jpg, dokument*.doc
FilenameIncludes	Izključene so vse datoteke	Sinhroniziraj datoteke (ostalo igrnoriraj): *.png, pomembno.doc
AttributeExcludeMask	Nič ni izključeno na podlagi	Ne sinhroniziraj sistemskih in

[2] To ne velja za datoteke, ki so bile izločene ob filtriranju. Te datoteke niso del sinhronizacijskega procesa in potemtakem iz njega ne morejo biti izločene.
[3] Sinhronizirajo se vse datoteke, razen Thumbs.db, Desktop.ini in sinhronizacijskih metapodatkov.

	atributov	skritih datotek
SubdirectoryExcludes	Izključena ni nobena mapa	Ne sinhroniziraj mape (in njenih podmap): c:\Podatki\Temp

4.5.1.5 Obravnava konfliktov

Večkrat pride do situacij, ko uporabniki spreminjajo isto datoteko na različnih sistemih. Ob sinhronizaciji teh datotek pride do konfliktov, saj v resnici ne vemo katera vsebina oziroma odločitev je prava. V splošnem MSF omogoča razreševanje konfliktov po meri. V primeru sinhronizacijskega ponudnika za datoteke in mape temu žal ni tako. Fiksni algoritem podpira naslednjo politiko razreševanja konfliktov: [14]

1.) **Konflikt zaradi vzajemne posodobitve (concurrent update conflict)**
Če je obstoječa datoteka posodobljena na obeh replikah se ohrani tista verzija, ki ima novejši čas posodobitve. V primeru enakega časa zadnje posodobitve se uporabi prvega. V primeru uporabe opcije ***Recycle*** v razredu ***FileSyncOptions***, se datoteka "poraženca" premakne v koš, namesto da bi se permanentno odstranila.

2.) **Konflikt zaradi brisanja in posodobitve (concurrent update-delete conflict)**
Če se obstoječa datoteka na eni repliki posodobi, na drugi pa je odstranjena, sinhronizacijski ponudnik vedno favorizira posodobitev pred brisanjem.

3.) **Konflikt zaradi hkratnega ustvarjanja in brisanja očeta (concurrent child create-parent delete conflict)**
V primeru, da je na prvi repliki ustvarjena datoteka, na drugi repliki pa je odstranjena mapa, ki bi morala to datoteko vsebovati, se uporabi enaka politika kot v točki 2.

4.) **Konflikt zaradi enakih imen map (name collision conflict for folders)**
Včasih se zgodi, da se pojavi enako ime mape na različnih replikah. Uporabnik v tem primeru obdrži ime mape, vsebina mape pa se združi ob naslednji sinhronizaciji.[4]

5.) **Konflikt zaradi enakih imen datotek (name collision conflict for files)**
Zgoraj omenjeni scenarij se lahko pripeti tudi pri datotekah. Zanje je postopek reševanje konfliktov malce drugačen.

   a. **Konflikt Ustvari-ustvari (create-create conflict)**
   V primeru, da se na obeh replikah ustvari datoteka z enakim imenom, se ime datoteke obdrži, vsebina datoteke pa je enaka tisti, ki ima novejši čas posodobitve (s pomočjo časovnega žigosanja). V primeru enakega časovnega žiga, se smatra, da gre za enako datoteko in se deterministično izbere "zmagovalca". Če je omogočena opcija ***Recycle***, je datoteka "poraženca" prestavljena v koš. V nasprotnem primeru je permanentno izbrisana.

[4] Upoštevati je potrebno, da so lahko znotraj obeh map datoteke in mape z enakimi imeni. V takšnih primerih se konflikte rešuje popolnoma enako, le da je npr. za vsako podmapo potrebno ponovno sinhronizirati.

b. Konflikt Ustvari-preimenuj (create-rename conflict)
V primeru, da se na na eni repliki ustvari datoteka, na drugi repliki pa neka obstoječa datoteka v postopku preimenovanja dobi enako ime, je razrešitev vedno takšna, da se obdržita obe datoteki, pri čemer se ena deterministično preimenuje.

4.5.1.6 Primer uporabe

Spodnji primer nakazuje uporabo sinhronizacijskega ponudnika za datoteke in mape. Sinhronizirati želimo dve mapi. Iz mape »*C:\temp\source*« bomo prenesli datoteke v mapo »*C:\temp\dest*« [14, 18].

```
        public void Sinhroniziraj()
        {
            // izvorna in ponorna replika (mapa)
            string sourcePath = @"C:\temp\source";
            string destPath = @"C:\temp\dest";

            // kreiranje razreda za filtriranje
            FileSyncScopeFilter filter = new FileSyncScopeFilter();

            // izključi datoteke s končnico .avi in .mp3
            filter.FileNameExcludes.Add("*.avi");
            filter.FileNameExcludes.Add("*.mp3");

            // izključi datoteke ki so označene samo za branje in sistemske datoteke
            filter.AttributeExcludeMask = FileAttributes.ReadOnly |
                                          FileAttributes.System;

            // datoteke, ki so "poražene" v postopku sinhronizacije premakni v koš
            (recycle bin)
            FileSyncOptions options = FileSyncOptions.RecycleConflictLoserFiles |
                                      FileSyncOptions.RecycleDeletedFiles |
                                      FileSyncOptions.RecyclePreviousFileOnUpdates;

            // kreiranje sinhronizacijskih ponudnikov z uporabo filtrov in opcij
            FileSyncProvider fspSource = new FileSyncProvider(sourcePath,
                                                              filter,
                                                              options);
            FileSyncProvider fspDest = new FileSyncProvider(destPath,
                                                            filter,
                                                            options);

            // povežemo se na dogodke in izkoristimo možnost poročanja
            fspDest.AppliedChange += new
EventHandler<AppliedChangeEventArgs>(fspDest_AppliedChange);
            fspDest.CopyingFile += new
EventHandler<CopyingFileEventArgs>(fspDest_CopyingFile);
            fspDest.DetectedChanges += new
EventHandler<DetectedChangesEventArgs>(fspDest_DetectedChanges);

            // kreiranje sinhronizacijskega agenta
            SyncOrchestrator agent = new SyncOrchestrator();

            // agenta povežemo z izvorno in ponorno repliko
            agent.LocalProvider = fspSource;
            agent.RemoteProvider = fspDest;

            // smer sinhroniziranja nastavimo na Upload (enosmerno)
            agent.Direction = SyncDirectionOrder.Upload;

              // sproži sinhronizacijo in shrani rezultate sinhronizacije v
              spremenljivko stats
            SyncOperationStatistics stats = agent.Synchronize();
        }

        void fspDest_DetectedChanges(object sender, DetectedChangesEventArgs e)
```

```
        {
            string s = "Zaznane spremembe" + Environment.NewLine +
                       "Število map: " + e.TotalDirectoriesFound.ToString() +
Environment.NewLine + "Število datotek: " + e.TotalFilesFound.ToString() +
Environment.NewLine + "Skupna velikost datotek: " + e.TotalFileSize.ToString();

            Console.WriteLine(s);
        }

        void fspDest_AppliedChange(object sender, AppliedChangeEventArgs e)
        {
            string izpis = string.Empty;

            switch (e.ChangeType)
            {
                case ChangeType.Create:
                    izpis = "Ustvarjena je bila datoteka " + e.NewFilePath;
                    break;
                case ChangeType.Delete:
                    izpis = "Zbrisana je bila datoteka " + e.OldFilePath;
                    break;
                case ChangeType.Rename:
                    izpis = "Datoteka " + e.OldFilePath + " je bila preimenovana v " +
                            e.NewFilePath;
                    break;
                case ChangeType.Update:
                    izpis = "Datoteka " + e.OldFilePath + " je bila posodobljena";
                    break;
            }
            Console.WriteLine(izpis);
        }

        void fspDest_CopyingFile(object sender, CopyingFileEventArgs e)
        {
            string s = "Prenaša se datoteka " + e.FilePath + " (" +
                       e.PercentCopied.ToString() + " %)";
            Console.WriteLine(s);
        }
```

4.5.2 Sinhronizacija podatkovnih baz

4.5.2.1 Splošno

V poglavju 4.1 je prikazana arhitektura MSF. Do sedaj smo že omenili sinhronizacijske ponudnike za datoteke in mape ter spletne ponudnike za RSS in Atom. V poglavjih 4.3 in 4.4 smo se dotaknili tudi jedrnih komponent za lasten razvoj, ko smo razložili metapodatkovni model in Microsoft Storage Service – MSS. V tem poglavju bodo razloženi sinhronizacijski ponudniki za podatkovne baze.

Razvoj samostojnih sinhronizacijskih ponudnikov in metapodatkovnih skladišč ni več v obsegu tega dela. Za implementacijo lastnega sinhronizacijskega ponudnika je namreč potrebno kar nekaj truda. Nadalje je potrebno poznati podrobnosti o, s strani MSF, nepodprtem podatkovnem viru in predvsem konkreten problem sinhronizacije. Brez teh informacij je težko razložiti kako in zakaj je bila neka konkretna stvar implementirana.

Kot že rečeno, MSF omogoča sinhronizacijo različnih podatkovnih virov. Nekateri izmed njih so že podprti v MSF in jih praktično ni potrebno razširjati in prilagajati. V tem poglavju bomo obravnavali zgolj sinhronizacijo tistih podatkovnih virov, ki so že podprti s strani MSF. Microsoft je leta nazaj predstavil ADO.NET (*ActiveX Data Object for .NET*), nabor komponent za komunikacijo s podatkovnimi viri za .NET platformo. Na tem mestu naj bo dovolj vedeti le to, da se ta tehnologija v mnogih aplikacijah uporablja še danes in da sinhronizacijski ponudniki za podatkovne baze, ki so del MSF, temeljijo na njej. V naslednjih poglavjih torej sledi opis sinhronizacijskih ponudnikov za podatkovne baze. Microsoft je sklop knjižnic, ki vsebujejo sinhronizacijske ponudnike za podatkovne baze, poimenoval *Sync Services for ADO.NET* ter ga razdelil na dva dela:

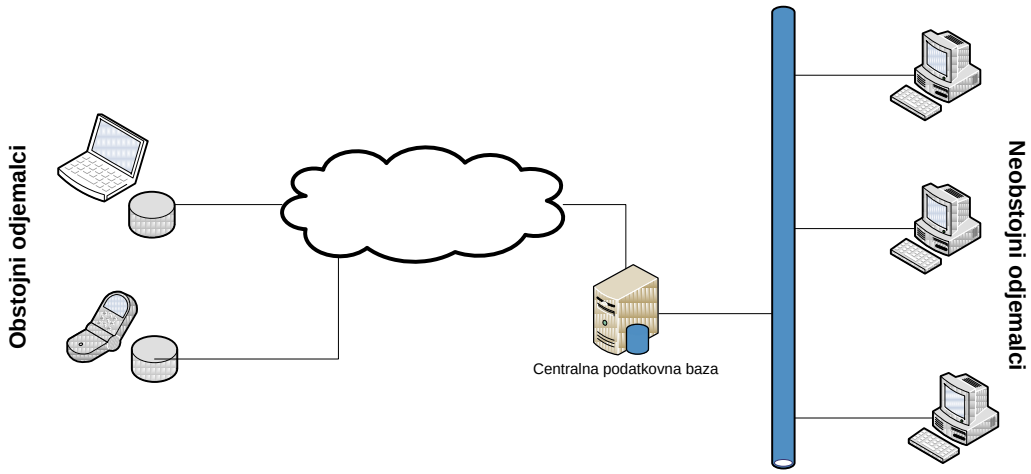
- Nepovezani scenariji – *Offline scenarios* in
- Vsak z vsakim – *Peer to peer* ali z bolj poznano kratico *P2P* [14, 19].

V svojem diplomskem delu sem se osredotočil samo na prvi scenarij in ga bom predstavil v naslednjih poglavjih.

4.5.2.2 Nepovezani scenariji (Sync Services for ADO.NET)

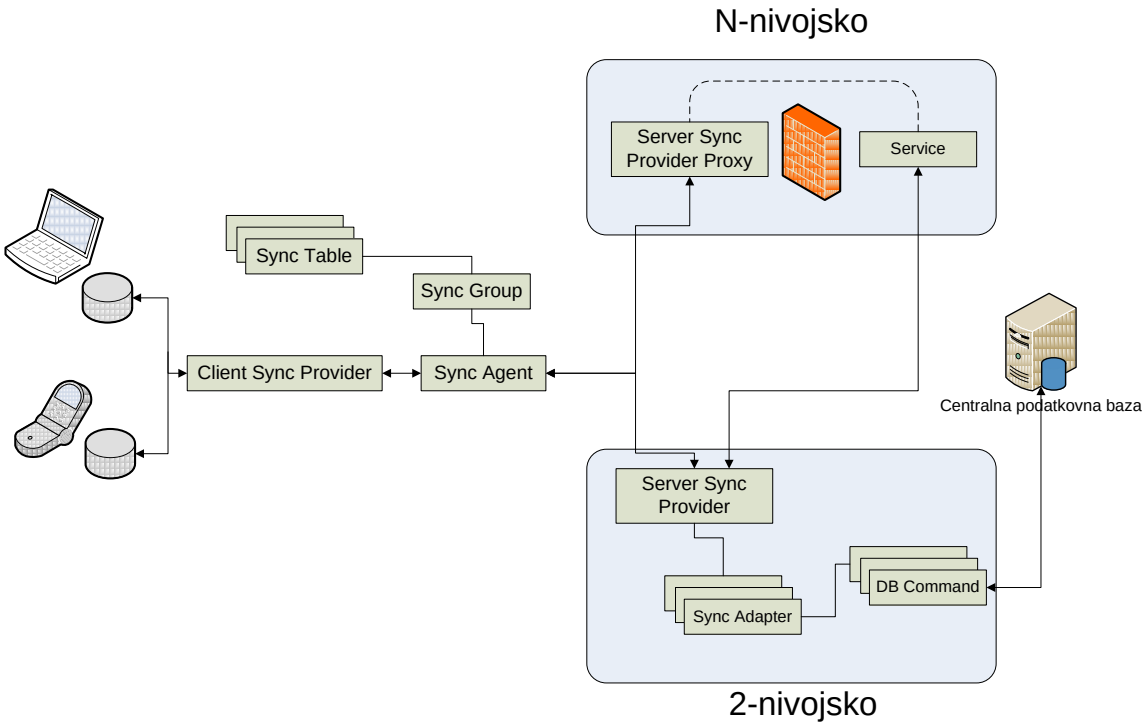
Na sliki 23 je prikazana shema delovanja občasno povezanih naprav. Na tem mestu je potrebno poudariti, da so oblak in povezave obstojnih odjemalcev na sliki lahko rahlo zavajajoče. Odjemalci namreč ne potrebujejo žične povezave pač pa le omrežje ki “vidi” centralni strežnik. To v praksi pomeni, da imamo lahko v odjemalcu vgrajena oba sinhronizacijska ponudnika (za izvorno in ponorno repliko). Takšna rešitev je uporabna, če se sinhronizacija venomer vrši znotraj lokalnega omrežja (npr. v podjetju). Seveda je takšnih primerov, ko se sinhronizacija vrši izven lokalne domene precej več. Tako lahko odjemalci z obstojnimi podatkovnimi bazami na drugem koncu sveta preko medmrežja sinhronizirajo svoje podatke s centralnim podatkovnim strežnikom. Veliko je bilo že napisanega na temo inkrementalnih posodobitev replik. V primeru sinhronizacije podatkovnih baz je ta zahteva ena izmed najpomembnejših, saj bi bila sinhronizacija, ki ne izvaja inkrementalnih sprememb na replikah prevelik performančni “udarec” za transakcijske podatkovne baze (OLTP).

Scenarija, ko se sinhronizacija vrši lokalno, ne bomo opisovali. Takšno sinhronizacijio namreč lahko obravnavamo kot podproblem splošne sinhronizacije.



Slika 23: Shema delovanja občasno povezanih naprav

Po krajšem premisleku ugotovimo, da je sinhronizacija preko svetovnega spleta v primerjavi z lokalno rahlo trši oreh. Problem nastane zaradi varnostnih politik. Logično (ali pa vsaj močno zaželjeno) je, da podjetje svojih podatkovnih baz ne pušča nezaščitenih. Dostop nepooblaščenim osebam bi moral biti vedno onemogočen. Ker je zahteva po varnosti prioritizirana višje kot sinhronizacija terenskih podatkov, moramo sinhronizacijo prilagoditi tako, da bo sinhronizacija izvedljiva, hkrati pa podatki še vedno varni. Na diagramu 24 je bolj podrobno prikazana arhitektura sinhronizacije občasno povezanih naprav [14, 20].



Slika 24: Podrobna arhitektura sinhronizacije

Za uspešno sinhronizacijo je potrebno podatke "pripeljati" iz zaščitenega lokalnega omrežja v manj zaščiteno področje – spletni strežnik. Implementirati je potrebno storitev, ki bo "simulirala" delovanje strežniških sinhronizacijskih komponent v spletnem kontekstu. MSF je produkt podjetja Microsoft in kot tak deluje le v Windows okolju. Če privzamemo, da je izvajalno okolje Windows, lahko kot spletni strežnik uporabimo Internet Information Services (IIS).. Dodatno je potrebno namestiti tudi ASP.NET, ker se naša bodoča storitev zanaša nanj. Odločiti se moramo še, kakšno storitev bomo implementirali. Najbolj očitni možni izbiri sta:

- Spletna storitev (*Web services*) – ASMX
- *Windows Communication Foundation* storitev *(WCF)* – SVC

Prednosti in slabosti posamezne izbire niso v obsegu tega dela, zato se na tem mestu z njimi ne bomo ukvarjali. WCF je v primerjavi z ASMX spletnimi storitvami konceptualno novejši, bolj ustreza standardom in je celostno bolje zasnovan, zato sem za demonstracijo izbral WCF storitev, katere implementacijo bom v naslednjih poglavjih tudi opisal.

Na sliki 24 je vidnih precej gradnikov. Vsi so bili neposredno ali posredno že omenjeni. Sledi razlaga vsakega izmed njih.

Sync Agent: Predstavlja sinhronizacijskega agenta. Ta gradnik se poveže in "sodeluje" z sinhronizacijskimi ponudniki. Njegova naloga je uvedba sprememb na izvorni in ponorni repliki. Odgovoren je tudi za informacije povezane s sinhronizacijsko sejo.

Sync Table/Sync Group: Sinhronizacijska tabela je najmanjša enota dela na odjemalcu. Metapodatki o spremembah se v demonstracijski implementaciji sicer beležijo na nivoju zapisov v tabeli v podatkovni bazi. Metapodatke bi lahko beležili tudi na nivoju atributa posameznega zapisa, vendar bi to precej povečalo količino metapodatkov. Takšna granularnost beleženja metapodatkov bi se izplačala le, če bi imela podatkovna tabela za posamezen zapis res veliko atributov, katerih vrednost se redko spreminja. To seveda zavisi od problemske domene in poslovnih pravil, tako da na tem mestu ni smiselno razglabljati katera izbira je boljša. Sinhronizacijske tabele beležijo podatke o tem katere podatkovne tabele je potrebno sinhronizirati, smer sinhronizacije in **način nastanka tabele**. O načinu nastanka tabele bomo govorili v naslednjih poglavjih, ko bomo opisovali postopek implementacije. Sinhronizacijska tabela se nahaja znotraj sinhronizacijske grupe. Sinhronizacijska grupa predstavlja nekakšno transakcijsko mejo med sinhronizacijskimi tabelami. S pomočjo grup na primer lahko zagotavljamo referencialno integriteto. V primeru, da odjemalci na terenu lahko spreminjajo šifrante, lahko to privede do padlih transakcij. Zato lahko vse (spremenljive) tabele šifrantov uvrstimo v sinhronizacijsko grupo "šifranti", vse ostale sinhronizacijske tabele pa v grupo "podatki". Sinhronizacijo tako prvo vršimo na grupi "šifranti", potem pa še na grupi "podatki". V nadaljevanju bomo videli, da je vrstni red dodajanja posamezne sinhronizacijske tabele v sinhronizacijsko zbirko pomemben.

Client Sync Provider: Sinhronizacijski ponudnik na odjemalcu je zadolžen za komunikacijo z odjemalcem in vsebuje podrobnosti o sinhronizaciji. Hkrati je zadolžen še za zaznavanje in uvedbo sprememb na odjemalcu.

Server Sync Provider: Naloga sinhronizacijskega ponudnika na strežniku je podobna kot naloga sinhronizacijskega ponudnika na odjemalcu.

Server Sync Provider Proxy: V N-nivojski sinhronizaciji je posredovalni strežnik (*angl. proxy)* zadolžen za posredovanje funkcionalnosti sinhronizacijskega ponudnika na strežniku. Običajno so podatkovne baze zaščitene s požarnimi zidovi. Posredovalni strežnik je idealen za premostitev te prepreke.

Sync Adapter: Sinhronizacijski adapter se uporablja za komunikacijo s strežniško podatkovno bazo. V nadaljevanju bomo opisali implementacijo in uporabo le-tega.

Dodatno je na tem mestu smiselno razložiti še tri pojme.

Sync Anchor: Sinhronizacijsko "sidro" nam pomaga določiti okno sinhroniziranja. Veliko je bilo že napisanega o inkrementalnih spremembah. Sinhronizacijsko sidro je mehanizem, ki jih omogoča.

Sync Session: Sinhronizacijska seja vsebuje podatke o posamezni sinhronizaciji in skrbi za stanja in transakcije med samo sinhronizacijo. Znotraj seje se nahajajo tudi inicializirane spremenljivke, ki se tičejo izključno dotične seje [21].

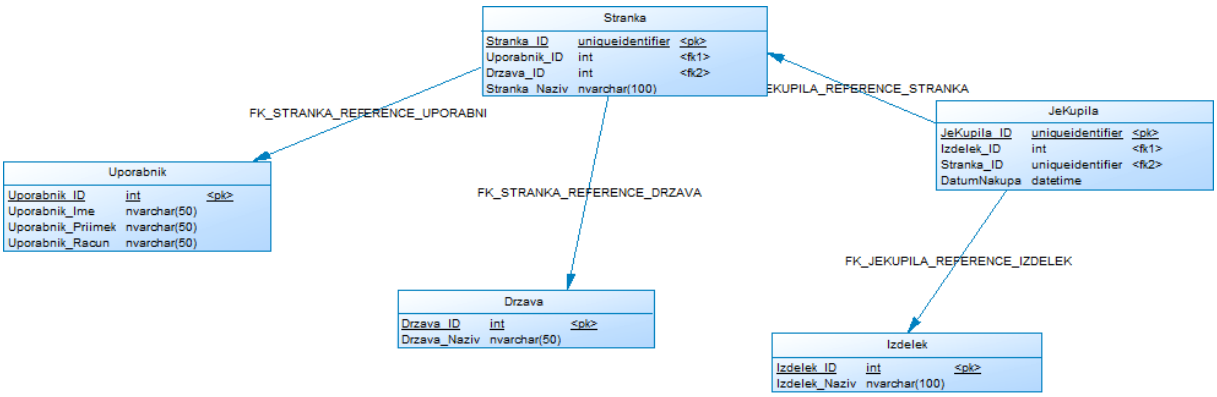
Sync Session Statistics: Vsaka sinhronizacija ima svojo sled. Ta sled je shranjena v tem razredu [14, 20].

Implementacijo N-nivojske sinhronizacije bomo razdelili na več zaključenih celot:

- Podatkovna baza
- Sinhronizacijski ponudnik na strežniku in bazne procedure
- Posredovalni strežnik (proxy)
- Sinhronizacijski ponudnik na odjemalcu
- Obravnava konfliktov

4.5.2.3 Podatkovna baza

Za namene demonstracije bomo uporabljali preprosto podatkovno bazo. Logični podatkovni model preproste podatkovne baze za MS SQL Server 2008 R2 se nahaja spodaj.



Slika 25: Osnovni logični podatkovni model

Razlaga logičnega podatkovnega modela je sledeča. Vsak (terenski) uporabnik je zadolžen za določene stranke. Vsaka stranka pripada neki državi. Vsaka stranka kupuje izdelke.
Uporabnik: Vsebuje vse terenske uporabnike. Tabela ne podpira brisanja in spreminjanja na terenu.

Tabela 15: Podatki o posameznih tabelah

Tabela	Opis	Sinhronizacija
Stranka	Vsebuje stranke.	Podpira dodajanje in urejanje na terenu. Terenski uporabnik vidi samo stranke, ki so mu dodeljene preko tujega ključa "Uporabnik_ID". Na teren se pošiljajo inkrementalni podatki o dodanih, spremenjenih in odstranjenih strankah.
Uporabnik	Vsebuje vse terenske uporabnike.	Ne podpira dodajanja, brisanja in urejanja na terenu. Na teren se pošiljajo inkrementalni podatki o dodanih in spremenjenih uporabnikih. Običajno se uporabnikov ne odstranjuje zaradi tujih ključev, zato tudi mi ne bomo implementirali funkcionalnosti brisanja uporabnikov.
Drzava	Drzava iz katere prihaja stranka.	Ne podpira brisanjam, dodajanja in urejanja na terenu. Na teren se pošiljajo podatki o dodanih, spremenjenih in odstranjenih državah.
Izdelek	Šifrant izdelkov.	Ne podpira brisanja, dodajanja in urejanja na terenu. Na teren se pošiljajo podatki o dodanih, spremenjenih in odstranjenih izdelkih.
JeKupila	Vsaka stranka kupi nič ali več izdelkov.	Podpira dodajanje, urejanje in odstranjevanje nakupov. Na teren se pošiljajo podatki o dodanih, spremenjenih ih odstranjenih nakupih.

Če želimo uporabljati sinhronizacijo, moramo naš osnovni podatkovni model razširiti. Večkrat je bilo že poudarjeno, da se delovanje MSF temelji na metapodatkih. Nekje moramo beležiti kdaj in kdo je nek zapis v tabeli urejal oziroma ustvaril. Že na tem mestu imamo dve možnosti:

- Uporaba Nagrobnih (angl. *Tombstone*) tabel in prožilcev (angl. *Triggers*) ali
- Uporaba avtomatskega beleženja sprememb.

Pri prvi možnosti moramo na posamezno tabelo, ki jo želimo sinhronizirati, dodati štiri atribute[5]:

- Datum nastanka,
- Datum spremembe,
- Kdo je zapis ustvaril,
- Kdo je zapis nazadnje spreminjal.

Vse te vrednosti je potrebno "vzdrževati" s pomočjo prožilcev in/ali s pomočjo privzetih vrednosti (angl. *Defaults*).
Dodatno je potrebno ustvariti nagrobno tabelo. Če bi recimo želeli ustvariti nagrobno tabelo za tabelo "Stranka", bi jo poimenovali "Stranka_Tombstone". V tej tabeli se namreč hrani informacija o tem, kdaj in kateri zapis iz tabele "Stranka" je bil izbrisan. Obvezno moramo v njej beležiti datum izbrisa in celoten primarni ključ osnovne tabele. Opcijsko pa lahko v njej beležimo tudi vrednosti vseh atributov v času, ko je bil zapis odstranjen in morebitne druge informacije.

Po drugi strani se avtomatsko beleženje sprememb sliši super, saj se s tem znebimo precej dodatnega dela, ki smo ga opisali zgoraj. Na žalost je ta možnost izvedljiva le na MS SQL Server 2005 in novejših omogoča pa beleženje na nivoju zapisov in na nivoju stolpcev. Na tem mestu pa vendarle lahko v izogib mistifikaciji povemo, da MS SQL Server za vsako tabelo v podatkovni bazi, za katero to želimo, vodi evidenco sprememb v sistemskih tabelah. Podatki so zelo enostavno dostopni s pomočjo T-SQL DML (*Data Manipulation Language)* ukaza "SELECT". V praksi to pomeni, da moramo povsod, kjer želimo uporabiti informacijo o nastanku oziroma spremembi zapisa, narediti združitev tabel (angl. *join, inner join, left join,..*). Vsi ostali deli baznih procedur ostanejo enaki, ne glede na izbiro implementacije spremljanja sprememb. Avtomatska implementacija je torej podobna tisti z nagrobniki, le da je malce bolj kompaktna [22].

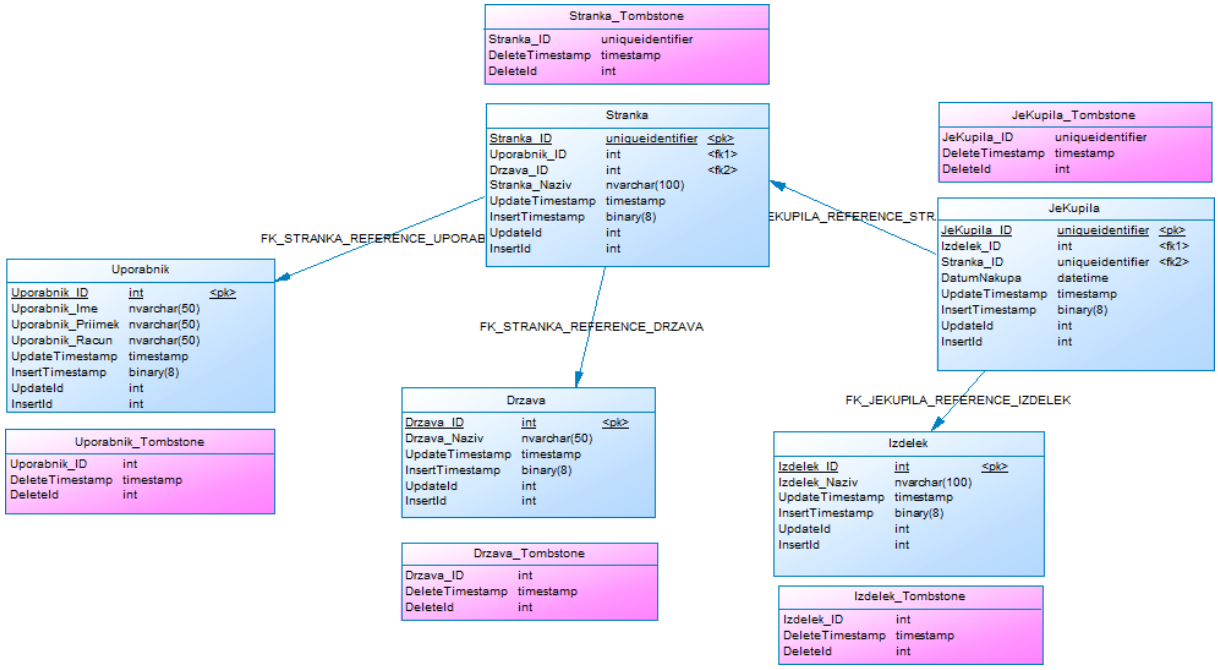
Prva možnost je bolj generična in deluje tudi na konkurenčnem Oracle podatkovnem strežniku, zato bomo opisali implementacijo le-te. Mimogrede, Oracle je ena izmed mnogih podatkovnih baz, ki so podprte s strani ADO.NET.

Kot zanimivost velja na tem mestu omeniti tudi dejstvo, da Visual Studio od verzije 2005 naprej omogoča konfiguracijo lokalne podatkovne baze s pomočjo čarovnika. Čarovnik je

[5] Tu že predpostavljamo, da bomo spremembe beležili na nivoju zapisov. Če bi hoteli spremembe beležiti na nivoju atributov, bi potrebovali bistveno več atributov. Če smo natančni, bi jih potrebovali 2 + N×2, pri čemer je N število stolpcev, katere želimo spremljati. Dva stolpca v začetku formule predstavljata InsertId in InsertTimestamp. Ravno zaradi te nerodne lastnosti, MS SQL Server omogoča avtomatsko beleženje sprememb v sistemskih tabelah, ki so grajene tako, da dopuščajo praktično neomejeno število stolpcev, brez dodatnih razširitev.

zmožen uporabiti obstoječe atribute v tabeli za namene sinhronizacije oziroma jih doda avtomatsko. Ravno tako doda prožilce na tabele v podatkovni bazi na strežniku [23].

Spodaj je dodelan logični podatkovni model. Le-ta sedaj vsebuje nagrobne tabele ter dodatne sinhronizacijske atribute v posameznih tabelah [14].



Slika 26: Logični podatkovni model z dodano sinhronizacijo

Skripta za generiranje podatkovne baze se nahaja v prilogi A.

4.5.2.4 *Sinhronizacijski ponudnik na strežniku*

Sinhronizacijo na najnižjem nivoju v kontekstu MSF vršita sinhronizacijska ponudnika. Ponudnik na strežniku je odgovoren za ustvarjanje sinhronizacijskih tabel in ukazov na strežniški podatkovni bazi.

Za vsako tabelo v strežniški podatkovni bazi, ki jo želimo prenesti na odjemalca, moramo ustvariti sinhronizacijski adapter. Sinhronizacijski adapter vsebuje ukaze za manipuliranje s podatki (DML). Dobra praksa narekuje, da DML ukaze iz sinhronizacijskih adapterjev kličemo s pomočjo baznih procedur (angl. *stored procedures*), če je to možno. V primeru dodajanja oziroma spreminjanja atributov na strežniški podatkovni bazi se tako na ta način izognemo ponovnemu izgradu izvodne kode na strežniški strani, vse skupaj pa je lepše in lažje obvladljivo ter nastavljivo.

Sinhronizacijsko sidro

Zgoraj smo že omenili pojem *Sync Anchor.* Uporabljamo ga za pridobitev sinhronizacijskega okna – obsega sinhronizacije. Implementacija je enostavna. Ustvariti moramo bazno proceduro (dajmo ji ime **sp_NewAnchor**) z naslednjo vsebino:

```
CREATE PROCEDURE [dbo].[sp_NewAnchor]
(
      @sync_new_received_anchor timestamp out
)
AS

SELECT @sync_new_received_anchor = @@DBTS
```

Procedura za delovanje potrebuje en parameter tipa timestamp, ki mora biti označen kot izhoden. Če je atribut označen kot izhoden, to pomeni, da procedura/funkcija vanj v času izvajanja zabeleži neko vrednost. V našem primeru je v izhodni atribut **@sync_new_received_anchor** zabeležena vrednost **@@DBTS**, ki predstavlja trenutni časovni žig podatkovne baze. Časovni žig podatkovne baze (gre za globalno spremenljivko) se spremeni ob vsaki spremembi kateregakoli zapisa v podatkovni bazi.

Originator (izvršitelj) spremembe

Poleg sinhronizacijskega sidra potrebujemo tudi ukaz za pridobitev IDja uporabnika, ki je spremembo izvršil. To storimo s pomočjo ***ServerClientIdCommand*** na strežniškem sinhronizacijskem ponudniku. Ta ukaz kliče bazno proceduro ***sp_GetOriginatorId***, ki s pomočjo parametra ***@sync_client_account*** pridobi ID uporabnika, ki se nato uporablja v vseh sinhronizacijskih ukazih.

```
CREATE PROCEDURE [dbo].[sp_GetOriginatorId]
      @sync_client_account nvarchar(50),
      @sync_originator_id int out
AS
BEGIN
      SET NOCOUNT ON;

      SELECT @sync_originator_id = Uporabnik_ID FROM Uporabnik WHERE
Uporabnik_Racun = @sync_client_account
END
```

Bazna procedura sp_GetOriginatorId preprosto vzame vrednost uporabniškega računa (angl. *username*) ter vrne odgovarjajoči ID iz tabele *Uporabnik*. Polje *Uporabnik_Racun* je na podatkovni bazi označeno kot unikatno, kar zagotavlja enoličnost.

Sinhronizacijski adapterji

Sinhronizacijski adapter se uporablja za komunikacijo s strežniško podatkovno bazo. Za vsako tabelo, ki jo želimo sinhronizirati, potrebujemo svoj sinhronizacijski adapter. Sinhronizacijski adapter za svoje delovanje potrebuje DML ukaze za inkrementalne spremembe (inkrementalo dodajanje, urejanje in brisanje) ter "običajne" DML ukaze za dodajanje, urejanje in brisanje. Katere ukaze bomo dejansko implementirali, je odvisno od končnih želja. Nekatere tabele se recimo posodabljajo samo na centralni podatkovni bazi, kar pomeni, da ukazov za dodajanje, urejanje in brisanje ne potrebujemo. Po drugi strani pa je obvezna implementacija ukazov za inkrementalne spremembe. Spodaj se nahaja primer sinhronizacijskega adapterja za tabelo *Izdelek*. Kot je razvidno, tabela izdelek potrebuje le

DML ukaze za inkrementalne spremembe, saj je dodajanje, odstranjevanje in spreminjanje izdelkov na terenu prepovedano oziroma onemogočeno.

```
private void InitializeAdapter()
{
    this.TableName = "Izdelek";

    #region INCREMENTAL UPDATES

    // zajemi UPDATE iz strežnika
    SqlCommand itemIncrUpdates = new SqlCommand();
    itemIncrUpdates.CommandText = "sp_Izdelek_Incremental_Updates";
    itemIncrUpdates.CommandType = CommandType.StoredProcedure;

    itemIncrUpdates.Parameters.Add("@" + SyncSession.SyncLastReceivedAnchor,
                                   SqlDbType.Timestamp);
    itemIncrUpdates.Parameters.Add("@" + SyncSession.SyncNewReceivedAnchor,
                                   SqlDbType.Timestamp);
    itemIncrUpdates.Parameters.Add("@" + SyncSession.SyncClientId,
                                   SqlDbType.UniqueIdentifier);
    itemIncrUpdates.Connection = Connection;

    this.SelectIncrementalUpdatesCommand = itemIncrUpdates;

    #endregion

    #region INCREMENTAL INSERTS

    // zajemi INSERT iz strežnika
    SqlCommand itemIncrInserts = new SqlCommand();
    itemIncrInserts.CommandText = "sp_Izdelek_Incremental_Inserts";
    itemIncrInserts.CommandType = CommandType.StoredProcedure;

    itemIncrInserts.Parameters.Add("@" + SyncSession.SyncLastReceivedAnchor,
                                   SqlDbType.Timestamp);
    itemIncrInserts.Parameters.Add("@" + SyncSession.SyncNewReceivedAnchor,
                                   SqlDbType.Timestamp);
    itemIncrInserts.Parameters.Add("@" + SyncSession.SyncClientId,
                                    SqlDbType.UniqueIdentifier);

    itemIncrInserts.Connection = Connection;
    this.SelectIncrementalInsertsCommand = itemIncrInserts;

    #endregion

    #region INCREMENTAL DELETES

    // zajemi DELETE iz strežnika
    SqlCommand itemIncrDeletes = new SqlCommand();
    itemIncrDeletes.CommandText = "sp_Izdelek_Incremental_Deletes";
    itemIncrDeletes.CommandType = CommandType.StoredProcedure;

    itemIncrDeletes.Parameters.Add("@" + SyncSession.SyncInitialized,
                                   SqlDbType.Bit);
    itemIncrDeletes.Parameters.Add("@" + SyncSession.SyncLastReceivedAnchor,
                                   SqlDbType.Timestamp);
    itemIncrDeletes.Parameters.Add("@" + SyncSession.SyncNewReceivedAnchor,
                                   SqlDbType.Timestamp);
    itemIncrDeletes.Parameters.Add("@" + SyncSession.SyncClientId,
                                   SqlDbType.UniqueIdentifier);
    itemIncrDeletes.Connection = Connection;
```

```
            this.SelectIncrementalDeletesCommand = itemIncrDeletes;

            #endregion
        }
```

Večina programske kode je precej neposredne in ne potrebuje dodatne razlage. Velja pa omeniti parametre, ki jih je potrebno podati baznim proceduram. Te parametre uporablja MSF med samo sinhronizacijo. Razred *Sync Session* je bil omenjen že zgoraj. Tu ga uporabimo kot pomoč pri deklaraciji imen parametrov za bazno proceduro. V našem primeru moramo ustvariti pet sinhronizacijskih adapterjev: *IzdelekSyncAdapter*, *StrankaSyncAdapter*, *DrzavaSyncAdapter*, *JeKupilaSyncAdapter* in *UporabnikSyncAdapter*.

Spodaj bomo opisali delovanje šestih baznih procedur, ki jih sinhronizacijski adapterji uporabljajo. Osredotočili se bomo na tabelo *Stranka*, ker zaradi dvosmerne sinhronizacije uporablja vseh šest procedur.

- sp_Stranka_Incremental_Inserts – zajame dodane zapise (insert)
- sp_Stranka_Incremental_Updates – zajame spremenjene zapise (update)
- sp_Stranka_Incremental_Deletes – zajame odstranjene zapise (delete)
- sp_Stranka_Inserts – doda zapise na strežnik (insert)
- sp_Stranka_Updates – spremeni zapise na strežniku (update)
- sp_Stranka_Deletes - odstrani zapise na strežniku (delete)

sp_Stranka_Incremental_Inserts

```
CREATE procedure [dbo].[sp_Stranka_Incremental_Inserts]
(
    @sync_last_received_anchor timestamp,
    @sync_new_received_anchor timestamp,
    @sync_originator_id int
) as

begin
    SELECT Stranka_ID, Uporabnik_ID, Drzava_ID,Stranka_Naziv
    FROM Stranka
    WHERE (
    InsertTimestamp >= @sync_last_received_anchor
    AND InsertTimestamp <= @sync_new_received_anchor
    AND InsertId <> @sync_originator_id)
End
```

Kot vhodne parametre procedura sprejme @sync_last_received_anchor in @sync_new_received_anchor ter @sync_originator_id. Na podlagi teh parametrov iz tabele *Stranka* izbere zapise, ki so bili dodani. Najbolj zanimiv je WHERE pogoj v SELECT ukazu. Pogoj zahteva, da je zapis nastal kasneje kot je bila zadnja sinhronizacija (@sync_last_received_anchor) ter oseba, ki ga je dodala, ni trenutni uporabnik. Če ne bi dodali zadnje vrstice (AND InsertId <> @sync_originator_id) bi prišlo do tako imenovanega odboja (angl. *echoing*). Zapisi, ki smo jih ravnokar dodali na našem odjemalcu, so se prenesli na strežnik, nato pa smo jih ponovno zajeli. Takšno obnašanje je seveda nezaželjeno, zato v izogib takšnim nevšečnostim vedno dodamo prej omenjeni pogoj in se tako znebimo odbojev. [14]

sp_Stranka_Incremental_Updates

```
CREATE procedure [dbo].[sp_Stranka_Incremental_Updates]
(
    @sync_last_received_anchor timestamp,
    @sync_new_received_anchor timestamp,
    @sync_originator_id int
) as

begin
    SELECT Stranka_ID, Uporabnik_ID, Drzava_ID, Stranka_Naziv
    FROM Stranka
    WHERE (
    UpdateTimestamp >= @sync_last_received_anchor
    AND UpdateTimestamp <= @sync_new_received_anchor
    AND UpdateId <> @sync_originator_id
    AND NOT (InsertTimestamp > @sync_last_received_anchor
    AND InsertId <> @sync_originator_id))
end
```

Inkrementalno spreminjanje kot vhodne parametre sprejme enake parametre kot inkrementalno dodajanje. SELECT ukaz iz tabele *Stranka* izbere tiste zapise, ki so bili spremenjeni kasneje od zadnje sinhronizacije. Hkrati pogoj preprečuje odboje in zahteva, da oseba, ki je zapis spreminjala, ni trenutni odjemalec. Na tem mestu bi nepozoren razvijalec svoj pogoj zaključil ter si s tem nakopal nemalo težav. Zadnji dve vrstici WHERE pogoja namreč preprečujeta zajem tistih vrstic, ki so dejansko nastale kasneje od zadnje sinhronizacije. Ob dodajanju zapisa v tabelo se atributa UpdateTimestamp in InsertTimestamp nič ne razlikujeta (sta identična). Zadnji dve vrstici pogoja preprečujeta zajem takšnih vrstic. Te vrstice bodo svoje mesto našle v rezultatu procedure za inkrementalno dodajanje.

sp_Stranka_Incremental_Deletes

```
CREATE procedure [dbo].[sp_Stranka_Incremental_Deletes]
(
    @sync_initialized bit,
    @sync_last_received_anchor timestamp,
    @sync_new_received_anchor timestamp,
    @sync_originator_id int
) as

begin
    SELECT Stranka_ID
    FROM Stranka_Tombstone
    WHERE (
    @sync_initialized = 1
    AND DeleteTimestamp > @sync_last_received_anchor
    AND DeleteTimestamp <= @sync_new_received_anchor
    AND DeleteId <> @sync_originator_id)
end
```

Inkrementalno odstranjevanje sprejme en atribut več kot ostale inkrementalne operacije - @sync_initialized, ki ima vrednost 0, če gre za prvo sinhronizacijo ter vrednost 1 za vse nadaljne. Procedura iz tabele *Stranka_Tombstone* zajame zapise, ki so bili odstranjeni

kasneje od naše zadnje sinhronizacije ter hkrati zahteva, da jih ni odstranil ravno trenutni odjemalec. Razlog za dodaten parameter je precej preprost. Zapise, ki so bili na strežniku odstranjeni in zajeti s pomočjo zgornje procedure, MSF takoj odstrani tudi na lokalni repliki. Če replika ni bila sinhronizirana (=inicializirana), ni zapisov. Zato odstranjevanje zapisov v takšnih primerih ni smiselno.

sp_Stranka_Inserts

```
CREATE procedure [dbo].[sp_Stranka_Inserts]
    @Stranka_ID uniqueidentifier,
    @Uporabnik_ID int,
    @Drzava_ID int,
    @Stranka_Naziv nvarchar(100),

    @sync_originator_id int,
    @sync_row_count int
AS
BEGIN

INSERT INTO Stranka(Stranka_ID, Uporabnik_ID, Drzava_ID,
Stranka_Naziv, InsertId, UpdateId)
VALUES (@Stranka_ID, @Uporabnik_ID, @Drzava_ID, @Stranka_Naziv,
@sync_originator_id, @sync_originator_id)

SET @sync_row_count = @@rowcount
end
```

Procedura za dodajanje zapisov na strežnik za svoje delovanje potrebuje dva atributa povezana s samo sinhronizacijo ter N atributov povezanih z dejanskim zapisom. Število N je odvisno od tega, koliko atributov nek zapis ima. V našem primeru je N = 4.

Procedura doda zapis v tabelo *Stranka*. Vrednosti atributov novega zapisa so podane v parametrih procedure. Poleg podatkovnih vrednosti v tabelo *Stranka* dodamo tudi sinhronizacijski vrednosti UpdateId in InsertId. Atributa InsertTimestamp in UpdateTimestamp zapolni SQL Strežnik sam, ker so v nastavitvah tabele nastavljene privzete vrednosti.

sp_Stranka_Updates

```
CREATE procedure [dbo].[sp_Stranka_Updates]
    @Stranka_ID uniqueidentifier,
    @Uporabnik_ID int,
    @Drzava_ID int,
    @Stranka_Naziv nvarchar(100),

    @sync_force_write bit,
    @sync_last_received_anchor timestamp,
    @sync_originator_id int,
    @sync_row_count int
 as

begin
```

```
    UPDATE Stranka SET
    Stranka_ID = @Stranka_ID,
    Uporabnik_ID = @Uporabnik_ID,
    Drzava_ID = @Drzava_ID,
    Stranka_Naziv = @Stranka_Naziv,
    UpdateId = @sync_originator_id

    WHERE
    Stranka_ID = @Stranka_ID AND
    (@sync_force_write = 1
    OR (UpdateTimestamp <= @sync_last_received_anchor
    OR UpdateId = @sync_originator_id))

    SET @sync_row_count = @@rowcount
end
```

Procedura sprejme enake podatkovne parametre kot zgornja. Sinhronizacijski parametri pa so:

- @sync_force_write – ta parameter nam pove ali naj spremembe zapišemo “na silo”.
- @sync_last_received_anchor – časovni žig zadnje sinhronizacije
- @sync_originator_id – id odjemalca
- @sync_row_count – število vrstic, ki so bile spremenjene na podlagi ravnokar izvedene bazne procedure.

sp_Stranka_Deletes

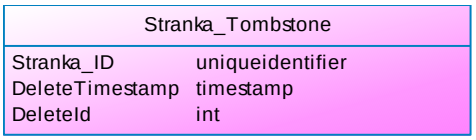
```
CREATE procedure [dbo].[sp_Stranka_Deletes]
      @Stranka_ID uniqueidentifier,
      @sync_force_write bit,
      @sync_last_received_anchor timestamp,
      @sync_originator_id int,
      @sync_row_count int
as

DELETE FROM Stranka
WHERE (Stranka_ID = @Stranka_ID)
AND (@sync_force_write = 1
OR (UpdateTimestamp <= @sync_last_received_anchor
OR UpdateId = @sync_originator_id))

SET @sync_row_count = @@rowcount
IF (@sync_row_count > 0)
BEGIN
      UPDATE Stranka_Tombstone
      SET DeleteId = @sync_originator_id
      WHERE (Stranka_ID = @Stranka_ID)
END
```

Procedura sprejme že znane parametre. Dodan je le parameter @Stranka_ID, ki predstavlja stranko, ki jo želim odstraniti. Procedura stranko prvo odstrani iz tabele *Stranka* nato pa v tabeli *Stranka_Tombstone* posodobi ID osebe, ki je stranko odstranila. Na tem mestu je nujno potrebno upoštevati zaporedje brisanja. Če pogledamo naš podatkovni model vidimo, da je tabela *JeKupila* odvisna od tabele *Stranka.*Če torej odstranjujemo zapis iz tabele *Stranka* je potrebno odstraniti tudi zapise iz tabele *JeKupila* in to preden odstranjujemo stranko. To je klasičen problem zaradi referencialne integritete in kot tak predstavlja željeno

oziroma pričakovano delovanje podatkovne baze. Rešitev le-tega je klic neke bazne procedure, ki bi odstranila vse zapise povezane s stranko v tabeli *JeKupila* (npr. exec sp_JeKupila_DeleteById(@Stranka_ID)).

Nagrobne tabele

Za tabele, za katere želimo beležiti odstranitve zapisov, je potrebno ustvariti nagrobno tabelo ter bazni prožilec. Spodaj se nahaja primer nagrobne tabele za tabelo *Stranka.*



Slika 27: Nagrobna tabela

Kot je opisano zgoraj, se v nagrobni tabeli nahajajo najmanj primarni ključ tabele *Stranka* (Stranka_ID), časovni žig odstranitve zapisa ter ID uporabnika, ki je zapis odstranil. Tu bi lahko dodali tudi ostale atribute tabele *Stranka* ter tudi kakšne druge, nepovezane. Primer takega atributa bi lahko bila recimo lokacija računalnika, na katerem je bil zapis odstranjen. Takšna informacija je zanimiva na primer pri aplikacijah s trgovskimi potniki, kjer mora potnik vedno podatke vnašati na licu mesta.

Spodaj se nahaja prožilec, ki skrbi za polnjenje tabele *Stranka_Tombstone*.

```
CREATE TRIGGER [dbo].[Stranka_DeleteTrigger]
ON [dbo].[Stranka] FOR DELETE
AS
BEGIN
SET NOCOUNT ON
      DELETE FROM Stranka_Tombstone
      WHERE EXISTS (SELECT * FROM DELETED WHERE Stranka_ID =
Stranka_Tombstone.Stranka_ID)

      INSERT INTO Stranka_Tombstone (Stranka_ID)
      SELECT Stranka_ID
      FROM deleted
SET NOCOUNT OFF
END
```

Prožilec je razdeljen na dva dela. Prvo iz tabele *Stranka_Tombstone* pobriše vse zapise, ki imajo *Stranka_ID* enak trenutnemu parametru *Stranka_ID,* nato pa informacijo doda Možno je[6], da se v tabeli že nahaja informacija o odstranitvi zapisa. Takšna informacija vsebuje star časovni žig. Zato je potrebno to informacijo odstraniti ter dodati novo, zato da jo bodo inkrementalne bazne procedure zajele.

Sinhronizacijski ponudnik

[6] V našem primeru, ko uporabljamo GUID, je to skoraj nemogoče.

Vse sinhronizacijske adapterje je potrebno nekako povezati v neko smiselno celoto. V ta namen je potrebno ustvariti sinhronizacijski ponudnik. Le-ta je zadolžen za inicializacijo sinhronizacijskih adapterjev. K temu delu je priložena izvorna koda celotne demonstracijske aplikacije, v kateri si bralec lahko bolj podrobno pogleda dejansko implementacijo.

Ker je naš cilj N-nivojska sinhronizacija, je potrebno že v začetni fazi misliti na prenosljivost same sinhronizacije. V ta namen sem izdelal posebno knjižnico *SyncLib*, v kateri so implementirani vsi zgoraj našteti sinhronizacijski adapterji ter sinhronizacijski ponudnik (*ServerSyncProvider.cs*).

4.5.2.5 Posredovalni strežnik (proxy)

Če želimo strežniški del sinhronizacije posredovati s pomočjo neke storitve, moramo sprejeti odločitev o prenosnem protokolu samih podatkov ter o gostiteljskem strežniku. Že v uvodu je bilo napisano, da bomo implementirali WCF storitev in jo bomo gostili v ASP.NET na IIS oziroma ASP.NET razvojnem strežniku.
Ustvariti je potrebno prazno spletno stran. Ta spletna stran vsebuje datoteko **web.config** v kateri podamo navodila ASP.NET strežniku (reference na knjižnice, varnost, avtentikacija,....).
Ustvariti je potrebno datoteko ***ServerSyncProvider.svc*** v kateri se nahaja samo ena vrstica kode:

```
<%@ServiceHost Service="SyncLib.ServerSyncProvider"%>
```

Ta vrstica kode spletnemu strežniku pove, da se storitev nahaja v knjižnici *SyncLib* v razredu *ServerSyncProvider*. Na tem mestu spletni strežnik ne ve kje se SyncLib pravzaprav najaha zato moramo dodati referenco na našo knjižnico. S tem dosežemo, da se *SyncLib.dll* prenese na spletni strežnik.
V datoteko web.config moramo dodati še nastavitve, ki bodo omogočile odjemalcem vpogled v metapodatke. Te nastavitve naredijo storitev “vidno“ zunanjemu svetu. Primer web.config datoteke se nahaja spodaj:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true"/>
  </system.webServer>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="MyBehavior">
          <serviceMetadata httpGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="false" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
   <services>
       <service name="SyncLib.ServerSyncProvider"
             behaviorConfiguration="MyBehavior">
        <endpoint address="" binding="wsHttpBinding"
```

```
                contract="SyncLib.IServerSyncProvider"/>
            <endpoint contract="IMetadataExchange"
               binding="mexHttpBinding" address="mex"/>
          </service>
        </services>
      </system.serviceModel>
    </configuration>
```

Najbolj pomemben del nastavitev se nahaja v sklopu <services>. Vsako storitev, ki jo želimo narediti vidno zunanjemu svetu, moramo navesti v tem sklopu. Poleg dejanskega imena (SyncLib.ServerSyncProvider), ki mora biti enak kot v .svc datoteki, moramo navesti še na kakšen način se bo odjemalec lahko povezoval (*endpoint binding=»wsHttpBinding«*).
WCF uvaja princip »pogodb« (angl. *contract*), ki jo v praksi implementiramo kot vmesnik (angl. *interface*). Vsaki metodi v vmesniku, ki jo želimo odkriti zunanjemu svetu, moramo dodati atribut **[OperationContract]**, samemu vmesniku pa atribut **[ServiceContract]**. Na ta način spletnemu strežniku povemo katere metode razreda SyncLib.ServerSyncProvider so vidne zunanjemu svetu. WCF storitev zahteva informacijo o tem vmesniku, kar je vidno v naslednji nastavitvi (atribut contract):

```
<endpoint address="" binding="wsHttpBinding" contract="SyncLib.IServerSyncProvider"/>
```

Storitev je sedaj pripravljena za posredovanje.

Če na kratko povzamemo, imamo v danem trenutku implementiran sinhronizacijski ponudnik za strežniško stran (*ServerSyncProvider.cs*), ki se nahaja v knjižnici *SyncLib*. Naš strežniški sinhronizacijski ponudnik implementira vmesnik *IServerSyncProvider.*cs, ki nekatere metode odkriva javnosti s pomočjo atributov [ServiceContract] in [OperationContract]. To je naša WCF storitev. Nadalje smo implementirali spletno stran, s pomočjo katere na novo pridobljeno WCF storitev objavljamo na spletu. Vsa našteta funkcionalnost je v obliki ustreznih tabel in baznih procedur ter prožilcev podprta v podatkovni bazi [24].

V nadaljevanju bomo opisali, kako to storitev lahko uporabljajo odjemalci.

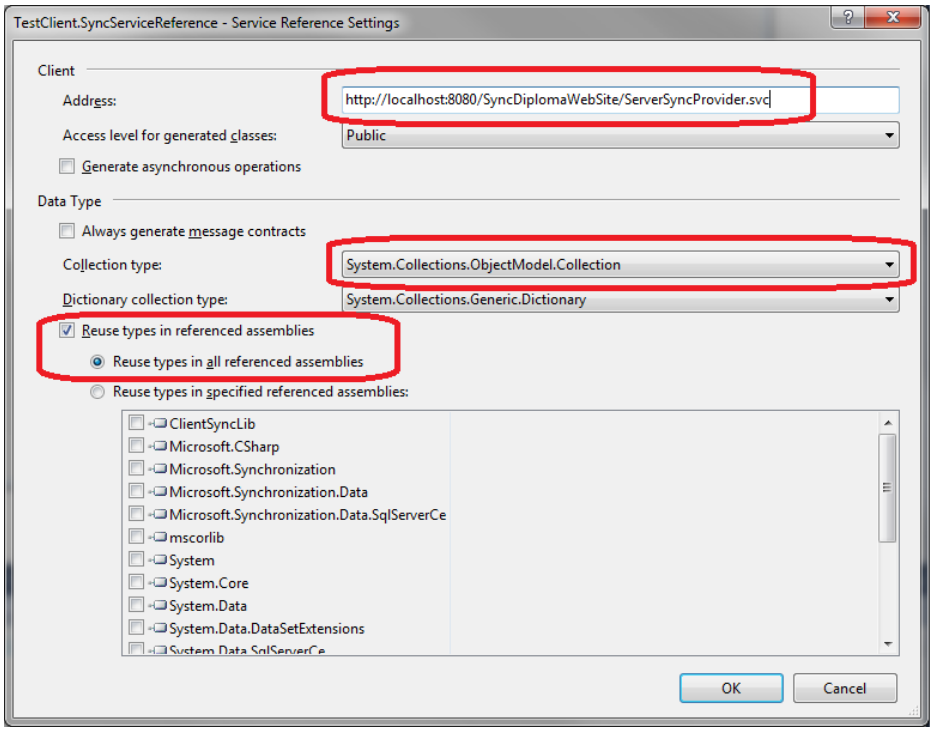
4.5.2.6 Sinhronizacijski ponudnik na odjemalcu

Zaradi lažje uporabe bomo tudi izvorno kodo sinhronizacije za odjemalce implementirali v obliki knjižnice (*ClientSyncLib*), sinhronizacijo pa bomo testirali v Windows Forms aplikaciji (*TestClient*).

Referenciranje WCF storitve

Prvi korak pri tem opravilu je dodajanje reference na WCF storitev, ki smo jo ustvarili v prejšnjem koraku. Referenco moramo dodati v projektu *TestClient.* To dosežemo z desnim klikom na samem projektu ter izberemo “Add Service Reference”. Pri konfiguraciji reference je pomembno, da označimo nastavitve tako kot kaže slika 28. Če tega ne storimo, bo Visual Studio sam ustvaril takoimenovane proxy razrede. S pomočjo metapodatkov (ki smo jih omogočili na WCF storitvi v prejšnjem poglavju) Visual Studio najde in doda referenco na

storitev, hkrati pa ustvari vse podatkovne strukture (razrede), ki jih med metapodatki najde, ter jih označi kot da so tipa "Proxy". Ko vršimo sinhronizacijo le-ta ne bo delovala, ker se podatkovne strukture tipsko ne ujemajo med seboj. V primeru, da navodila na sliki 28 upoštevamo, Visual Studio preskoči generiranje proxy razredov in uporabi že poznane definicije (v našem primeru iz knjižnic Microsoft.Synchronization.dll, Microsoft.Synchronization.Data.dll in Microsoft.Synchronization.SqlServerCe.dll. Če v Visual Studiu v "Solution Explorerju" kliknemu na tipko "Show all" bomo prikazali tudi skrite datoteke. Ena izmed skritih datotek je datoteka *References.cs*. V njej so shranjeni podatki o referenci na WCF storitev.[7]



Slika 28: Referenca na WCF storitev

Sinhronizacijske tabele – Sync Tables

Drugi korak pri implementaciji sinhronizacijskega ponudnika odjemalca je izgrad sinhronizacijskih tabel. Implementacijo tabel naredimo v projektu *ClientSyncLib.* Za vsako tabelo v centralni podatkovni bazi, ki bi jo radi sinhronizirali, je potrebno ustvariti sinhronizacijsko tabelo (angl. *SyncTable*).
Vsaki izmed teh tabel je potrebno določiti tri atribute:

- Ime tabele,
- Smer sinhronizacije,
- Način nastanka tabele na odjemalcu.

Smer sinhronizacije nam pove v kateri smeri se podatki sinhronizirajo. Dostopna je preko enumeracije ***SyncDirection***. Na voljo imamo štiri možnosti:

[7] Informacije o referenci na WCF storitev so shranjene še v nekaterih drugih datotekah, ki pa niso tako pomembne za naš primer.

- DownloadOnly: Podatki se prenašajo samo v smeri od strežnika k odjemalcu.
- UploadOnly: Podatki se prenašajo od odjemalca na strežnik.
- Bidirectional: Podatki se prenašajo v obe smeri (DownloadOnly + UploadOnly)
- Snapshot: Ob vsaki sinhronizaciji se podakti na odjemalcu popolnoma osvežijo. Poteka v smeri od strežnika k odjemalcu.

Ime tabele se mora ujemati z imenom v centralni podatkovni bazi. Načinov nastanka tabele na odjemalcu je več. Ravno tako kot smer sinhronizacije, so tudi načini nastanka tabele dostopni preko enumeracije (***TableCreationOption)***.
Na voljo so naslednje možnosti:

- CreateTableOrFail: Ustvari tabelo če ne obstaja. Če obstaja, vrni napako.
- UseExistingTableOrFail: Uporabi obstoječo tabelo. Če tabela ne obstaja, vrni napako.
- TruncateExistingOrCreateNewTable: Ustvari če tabela ne obstaja. Če obstaja, pobriši vse podatke iz nje in jo uporabi.
- DropExistingOrCreateNewTable: Ustvari novo tabelo, če ta ne obstaja. Če obstaja jo pobriši in ponovno ustvari.
- UploadExistingOrCreateNewTable: Če tabela ne obstaja, jo ustvari. Če tabela obstaja, ob prvi sinhronizaciji pošlji vsebino te tabele na centralni strežnik.

Izvorna koda spodaj prikazuje sinhronizacijsko tabelo za tabelo *Stranka*.

```
class StrankaSyncTable : Microsoft.Synchronization.Data.SyncTable
{
    public StrankaSyncTable()
    {
        this.TableName = "Stranka";

        this.CreationOption = TableCreationOption.DropExistingOrCreateNewTable;
        this.SyncDirection = SyncDirection.Bidirectional;
    }
}
```

Kot je razvidno iz zgornje kode je stvar zelo preprosta. V implementaciji demonstracijske aplikacije sem se zaradi preglednosti in lažje berljivosti odločil, da bom za vsako sinhronizacijsko tabelo dedoval od razreda Microsoft.Synchronization.Data.SyncTable, kar pa glede na količino kode v praksi niti ni potrebno. Vsako izmed sinhronizacijskih tabel moramo dodati v seznam (angl. *collection*) sinhronizacijskih tabel na sinhronizacijskem agentu [14].

Sinhronizacijski agent – Sync Agent

Na tem mestu imamo zopet možnost neposredne uporabe razreda Microsoft.Synchronization.SyncAgent ali pa od le-tega dedujemo ter razvijalcu aplikacije za odjemalca ustvarimo bolj prijazno izkušnjo. Ker smo se lotili implementacije knjižnice, je smiselno dedovanje.
V sinhronizacijskem agentu je potrebno inicializirati izvorni in ponorni sinhronizacijski ponudnik, vse sinhronizacijske tabele, morebitne sinhronizacijske grupe ter dodatne sinhronizacijske parametre, ki bi jih morda potrebovali. V našem primeru ob vsaki sinhronizaciji MSF podamo še vrednost atributa ***@sync_client_account***. Ta parameter vsebuje uporabniško ime (atribut *Uporabnik_Racun* v tabeli *Uporabnik*). V poglavju o

sinhronizacijskem ponudniku na strežniku smo že razložili pomen tega atributa, zato o njem ne bomo razpravljali. Sinhronizacijski agent sem poimenoval *DiplomaSyncAgent* [14].

Podatkovna baza na odjemalcu

Do popolne implementacije sinhronizacije na naši demonstracijski podatkovni bazi nas loči le še sinhronizacijski ponudnik na odjemalcu.
Prva stvar za katero se moramo odločiti je tip podatkovne baze na odjemalcu. Zaradi lažje implementacije in demonstracije vgrajenih zmogljivosti, sem izbral MS SQL Server CE podatkovno bazo. V poglavju 4.4 (Microsoft Storage Service) smo to podatkovno bazo že omenili. Gre za eno-datotečno podatkovno bazo, ki se ob branju v celoti naloži v pomnilnik naprave. Priročna je tudi zato, ker za svoje delovanje ne potrebuje strežnika. Glavna prednost te podatkovne baze v kontekstu sinhronizacije pa se skriva v dejstvu, da je zmožna samostojno beležiti spremembe. Tako kot MS SQL Server tudi MS SQL Server CE za beleženje sprememb uporablja sistemske tabele, v katerih drobovje se ne bomo spuščali [16].

Podatkovno bazo MS SQL Server CE ustvarimo z naslednjo metodo:

```
private void InitializeClientDatabase()
{
     string clientConn = Properties.Settings.Default.ClientDBConnectionString;

     using (SqlCeConnection sqlconn = new SqlCeConnection(clientConn))
     {
         if (!File.Exists(sqlconn.Database))
         {
             SqlCeEngine sqlCeEngine = new SqlCeEngine(clientConn);
             sqlCeEngine.CreateDatabase();

             Synchronize();
         }
     }
 }
```

Razred *SqlCeEngine* se nahaja v knjižnici System.Data.SqlServerCe. Metoda iz nastavitev prebere podrobnosti za povezavo (angl. *Connection string*). V nastavitvah demonstracijske aplikacije je vrednost te nastavitve "Data Source='ClientDB.sdf'". Ta nastavitev razredu SqlCeEngine pove, naj na tistem mestu kjer se nahaja aplikacija, ki to metodo izvaja, ustvari MS SQL Server CE podatkovno bazo z imenom *ClientDB.sdf*. Ker smo podatkovno bazo ravnokar ustvarili, je logično da izvršimo prvo sinhronizacijo, kar izvedemo s klicem metode *Synchronize()*. Vsebina te metode na tem mestu še ni pomembna.

Uspešno smo implementirali knjižnico *ClientSyncLib*. To knjižnico je sedaj potrebno uporabiti v dejanski aplikaciji na odjemalcu.

4.5.2.7 Implementacija odjemalca

V prejšnjem poglavju smo čisto na začetku omenili, da potrebujemo dva projekta:

- *ClientSyncLib* (knjižnica) in
- *TestClient* (aplikacija)

V projekt *TestClient* smo dodali referenco na našo WCF storitev. Sedaj moramo dodati še referenco na knjižnico *ClientSyncLib*. V njej smo namreč implementirali vso sinhronizacijsko logiko za odjemalca. Spodaj se nahaja metoda *Synchronize()*, ki smo jo tudi že omenili na koncu prejšnjega poglavja.

```
private void Synchronize()
{
    var userName = "martinb";
    DiplomaSyncAgent syncAgent = new
    DiplomaSyncAgent(Properties.Settings.Default.ClientDBConnectionString,
                     userName);

    syncAgent.AddServiceReference(new
                         SyncServiceReference.ServerSyncProviderClient());

    try
    {
        var statistics = syncAgent.Synchronize();

        StringBuilder sb = new StringBuilder();
        sb.AppendLine("Download Changes Applied: " +
             statistics.DownloadChangesApplied);
        sb.AppendLine("Download Changes Failed: " +
             statistics.DownloadChangesFailed);
        sb.AppendLine("Total Changes Downloaded: " +
             statistics.TotalChangesDownloaded);
        sb.AppendLine("Total Changes Uploaded: " +
             statistics.TotalChangesUploaded);
        sb.AppendLine("Upload Changes Applied: " +
             statistics.UploadChangesApplied);
        sb.AppendLine("Upload Changes Failed: " +
             statistics.UploadChangesFailed);

        MessageBox.Show(sb.ToString(), "Sync Results");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
}
```

Metoda naredi več stvari. Prvo inicializira dva objekta (*userName* in *syncAgent*). Spremenljivka *userName* vsebuje uporabniško ime odjemalca. Spremenljivka *syncAgent* je sinhronizacijski agent, ki smo ga razvili v prejšnjem poglavju. Konstruktor tega objekta sprejme tri spremenljivke: pot do odjemalčeve podatkovne baze, zato da lahko inicializira sinhronizacijski ponudnik za odjemalca, uporabniško ime, ki ga uporablja za pridobitev odjemalčevega IDja, ki sodeluje v večini sinhronizacijskih baznih procedur ter referenco na WCF storitev.

Na tem mestu je sinhronizacija implementirana in vse kar nam preostane je klic metode *Synchronize()* na objektu *syncAgent.* Metoda vrne statistiko sinhronizacije, ki jo tudi izpišemo. Vsa ostala izvorna koda v projektu *TestClient* služi le kot demonstracija in ni del sinhronizacijskega procesa. Vsebuje ukaze za spreminjanje, dodajanje in odstranjevanje zapisov na tako na odjemalcu, kot na strežniku.

4.5.2.8 Obravnava konfliktov

Do sedaj smo podrobno opisali razvoj podatkovne baze, sinhronizacijskih ponudnikov in sinhronizacijskega agenta. Sinhronizacijo smo implementirali ter jo z uporabo WCF storitve uspešno premostili iz lokalnega omrežja v svetovni splet. N-nivojski scenariji predvidevajo mnogo odjemalcev, ki svoje podatke sinhronizirajo. Predvidene so tudi "in-house" spremembe podatkovne baze – spremembe znotraj podjetja, na težkih odjemalcih. Slej kot prej pride do situacije, ko dva odjemalca (eden od njiju je lahko težki odjemalec) posodobita ali kako drugače obdelata isti zapis v podatkovni bazi.[8] Takrat ob sinhronizaciji pride do konflikta. Tako kot pri sinhronizacijskem ponudniku za datoteke in mape tudi tu obstajajo različne vrste konfliktov. Konflikti so vsebovani v enumeraciji ***ConflictType***.

Poznamo naslednje vrste konfliktov:

- **ClientInsertServerInsert** – Do tega konflikta pride, ko odjemalec in strežnik oba v neko tabelo vstavita zapis. Oba zapisa lahko dobita enak primarni ključ Ob sinhronizaciji pride do kolizije teh dveh ključev in dobimo konflikt.
- **ClientUpdateServerUpdate** – Do tega konflikta pride, ko odjemalec in strežnik oba posodobita isti zapis v neki tabeli. Ob sinhronizaciji MSF s pomočjo časovnega žigosanja in izvora spremembe (UpdateId) zazna kolizijo in nastane konflikt.
- **ClientUpdateServerDelete** – Do tega konflikta pride, ko odjemalec odstrani nek zapis, strežnik ga pa posodobi. S pomočjo časovnih žigov MSF ugotovi, da bi odjemalec *moral* poznati nek zapis, vendar tega zapisa ni. Hkrati MSF zazna, da je na strežniku prišlo do posodobitve istega zapisa in tako nastane konflikt.
- **ClientDeleteServerUpdate** – Popolnoma enako kot pri zgornjem konfliktu, le da tokrat strežnik zapis odstrani ter ga odjemalec posodablja.
- **ErrorsOccured** – Do tega konflikta pride takrat, kadar neka napaka prepreči sinhronizacijo. Takšna napaka je recimo lahko uveljavljanje referencialne integritete pri sklicevanju na neobstoječ zapis v podatkovni bazi.
- **Unknown** – Včasih se zgodi, da strežnik ne more določiti za kakšno vrsto konflikta gre. V tem primeru MSF tak konflikt označi kot "nepoznan" konflikt.

Prvo opažanje v primeru sinhronizacijskega konflikta je to, da se konflikt lahko zgodi na dveh različnih koncih:

- na strežniku in/ali
- na odjemalcu.

Na odjemalcu pride do konflikta takrat, kadar se spremembe prenašajo od strežnika k odjemalcu, na strežniku pa takrat kadar se spremembe prenašajo v drugo smer. Reševanje konfliktov je potemtakem potrebno na obeh straneh sinhronizacijskega toka. Sinhronizacijska ponudnika na strežniku in odjemalcu ob konfliktih prožita dogodek ***ApplyChangeFailed***. Če hočemo reševati nastale konflikte, moramo ta dva dogodka zajeti in ju ustrezno obravnavati. Na obeh straneh sinhronizacije je možno videti vrstice, ki so v konfliktu. SQL CE sinhronizacijski ponudnik (tega uporabljamo v naši demonstracijski aplikaciji) je zmožen te vrstice poiskati sam. Na strežniški strani jih moramo poiskati sami. To naredimo s pomočjo dveh SqlCommand objektov na sinhronizacijskem adapterju za

[8] Oba odjemalca zapis posodabljata v svoji podatkovni bazi.

posamezno tabelo. Ta dva objekta sta **SelectConflictUpdatedRowsCommand** ter **SelectConflictDeletedRowsCommand**.

V MSF se konflikti dogajajo na nivoju zapisov (vrstic v tabelah Za vsako vrstico v konfliktu lahko določimo željeno obnašanje MSF (**ApplyAction**). MSF pozna tri akcije:

- **Continue** – Ignoriraj konflikt in nadaljuj s sinhronizacijo.
- **RetryApplyingRow** – Ponovno poskusi z dodajanjem spremembe. Če vsebine vrstice pred tem ne spremenimo, se bo konflikt ponovil in dobili bomo neskončno zanko (angl. *deadlock*).
- **RetryWithForceWrite** – ponovno poskusi z dodajanjem spremembe z zastavico @sync_force_write = 1. S to zastavico so spremembe na silo zapisane v podatkovno bazo. Seveda ima referencialna integriteta še vedno veljavo in bi v primeru napak tudi ta rešitev odpovedala.

SQL CE sinhronizacijski ponudnik ima poseben objekt, ki je v pomoč pri reševanju konfliktov. Objekt se imenuje **ConflictResolver**. Objekt ConflictResolver ima pet različnih nastavitev:

- ClientDeleteServerUpdateAction,
- ClientUpdateServerDeleteAction,
- ClientInsertServerInsertAction,
- ClientUpdateServerUpdateAction,
- StoreErrorAction.

Zgoraj naštete nastavitve so identične tistim v enumeraciji ConflictType. S pomočjo enumeracije **ResolveAction** lahko nastavimo vrednosti tem nastavitvam. Možne vrednosti so:

- **ClientWins** – enakovredno ApplyAction.Continue
- **ServerWins** – enakovredno ApplyAction.RetryWithForceWrite
- **FireEvent** – sproži privzeti ApplyChangeFailed dogodek

Načeloma ni nobene potrebe po tem, da razvijalec nastavlja te vrednosti, saj so konflikti rešljivi od zajetju dogodka ApplyChangeFailed. Kljub temu se razvijalec lahko odloči in te vrednosti nastavi ter si s tem rahlo olajša delo [14, 25].

5 Varnost

Pravilno je, da ob čim bolj bogati funkcionalnosti naše aplikacije, poskrbimo tudi za varnost. V grobem, bi lahko varnost v informatiki razdelili na fizično in računalniško/informacijsko. Glede obeh je bilo že ogromno napisanega in na tem mestu nima smisla ponavljati že izrečenih splošnih modrosti. Ker pa varnost (predvsem računalniška) ni vedno samo nek produkt ampak bolj ekosistem, je smiselno navesti nekaj okvirnih prijemov za zaščito naše demonstracijske aplikacije. Tudi pri varnosti je potrebno delo razdeliti na dva dela: na strežniški ter na odjemalski del.

Prvo pravilo varnosti je, da je sistem varen točno toliko, kot je varen njegov najšibkejši člen. Gledano torej iz striktno fizičnega vidika, potrebujemo za strežnike ustrezno opremljen in varovan prostor (kamere, dostopne kode, požarna varnost, izpad električnega toka, ipd.)

Varnost je kakopak odvisna od želja posameznega podjetja. Kljub temu obstajajo smernice (in celo zakoni), ki naj bi jim podjetja sledila. Prva pametna reč, ki jo podjetje lahko naredi, je da na podlagi izkušenj administratorjev in v okviru zakonodaje sprejme lastno varnostno politiko. Brez tega dandanes v večjih podjetjih ne gre.

5.1 Strežnik

Na strežniški strani je skoraj samoumevno, da imajo uporabniki onemogočen neposreden dostop do podatkov. Uporabnik mora podatke obdelovati samo preko točno določenih vmesnikov (aplikacije) nikakor pa ne neposredno. Glede na arhitekturo naše demonstracijske aplikacije na sliki 24, lahko varnost na strežniški strani ponovno delimo na:

- Varnost podatkovne baze in
- Varnost WCF storitve oziroma IIS strežnika.

5.1.1 Varnost podatkovne baze

Sama podatkovna baza v našem primeru omogoča izvrstno varovanje podatkov. S pomočjo uporabniških računov in shem, lahko zelo granularno določamo kaj oziroma česa nek uporabnik ne vidi. Ravno tako imamo možnost uporabniku dodeliti pravico nad izvrševanjem baznih objektov (recimo bazne procedure).
Glede na to, da je demonstracijska aplikacija N-nivojska, je smiselno pričakovati veliko odjemalcev. V idealnem svetu bi vsak uporabnik dobil svoje uporabniško ime. Temu zaradi stroškov in težav pri vzdrževanju v praksi ni tako. Namesto tega, administratorji raje dodeljujejo uporabniška imena vsaki *aplikaciji*, na razvijalcih pa je, da potem znotraj lastne podatkovne baze razvijejo logiko za avtentikacijo uporabnikov.
Hkrati je smiselno občutljive podatke v podatkovni bazi kriptirati s pomočjo vgrajene enkripcije (če uporabljamo podatkovni strežnik, ki to omogoča) [26].

5.1.2 Varnost WCF storitve oziroma IIS strežnika

Od problema je odvisno kakšno varnost lahko vršimo na spletnem strežniku. Če imamo recimo IP-je odjemalcev, lahko že na samem IIS strežniku onemogočimo dostop do WCF storitve vsem nepoklicanim osebam. To je uporabno v zaprtih korporativnih omrežjih, kjer imamo nek nivo zaupanja, a je vendarle potrebno predvideti nepooblaščene dostope in poskuse vdorov.
Poleg zavračanja nepoznanih IP naslovov se dandanes za varovanje kočljivih podatkov v spletnih storitvah in aplikacijah uporabljajo v glavnem certifikati. Implementacija je opisana na spletnem naslovu [27].

5.2 Odjemalec

Tako kot strežnike, je potrebno tudi odjemalce zaščititi. Žal se velikokrat zgodi, da nam je naš lahki odjemalec protipravno odtujen oziroma nehote pride v roke in uporabo nepooblaščeni osebi. Glede takšnih ”napadov” lahko žal le redko ukrepamo. Ravno zato pa je potrebno poskrbeti, da se na odjemalcu nahajajo samo tiste informacije, ki se tičejo točno določenega uporabnika. V našem primeru sinhronizacije, smo za to poskrbeli s pomočjo bazne procedure *sp_GetOriginatorId*, s katero smo pridobili ID uporabnika in potem v odvisnosti od tega ID-ja na odjemalca vračali zapise. Kljub temu je možno, da so tudi tako filtrirani podatki ”uporabni” v rokah nepridipravov.
V našem demonstracijskem primeru kot lokalno podatkovno bazo uporabljamo MS SQL Server CE. Le-ta ima zelo okrnjeno varovanje. Omogoča namreč centralno geslo ter kriptiranje podatkov. Če podrobno pogledamo naš povezovalni niz (angl. *Connection string*), opazimo sklop ”Password=diploma123”.

Povezovalni niz = ”Data Source='ClientDB.sdf'; Password=diploma123”

Ob prvem nastanku lokalne podatkovne baze MSF izvrševalno okolje na odjemalcu dojame, da podatkovna baza ne obstaja. Če v našem povezovalnem nizu navedemo geslo, bo ob ustvarjanju nove podatkovne baze MSF to bazo zaščitil z navedenim geslom [15].

6 Razhroščevanje sinhronizacije

Pravilno delovanje aplikacij je običajno cilj vseh razvijalcev. Kljub vloženemu trudu pa se včasih zgodi, da aplikacija ne deluje pravilno. V takih primerih se prične diagnostika za ugotavljanje mesta napake. Tej diagnostiki pravimo razhroščevanje (angl. *Debugging*). Največkrat se zgodi, da se zaradi napake ob sinhronizaciji enega samega zapisa napačno ali pa sploh ne sinhronizirajo tudi drugi zapisi. V takih primerih je potrebno poznati dejanske vrednosti v tabelah, ki so povzročile napako. Na tem mestu je pomembno vedeti, da tu večinoma ne gre sintaktične napake ampak za semantične. Takšne napake je bistveno težje odkrivati. V nadaljevanju bomo spoznali orodje za analizo ukazov na SQL strežniku MS SQL Server. Opisali bomo tudi postopek s katerim lahko natančno vidimo celoten postopek sinhronizacije (log sinhronizacije).

6.1 Dnevnik dogodkov

MSF ob sinhronizaciji izvaja mnogo operacij. Vse te operacije je možno zajeti s pomočjo dnevnika dogodkov (angl. *Trace log*). Dogodke je možno beležiti na dva načina:

- Z infrastrukturo, ki je osnovana na .NET platformi,
- S pomočjo razreda **SyncTracer**, ki se nahaja v sinhronizacijskih knjižnicah.

Oba načina sta opisana v spodnjih poglavjih.

6.1.1 Vgrajena infrastruktura v .NET

Tovrstno beleženje dogodkov lahko dosežemo s pomočjo konfiguracijskih datotek. Glede na to, da je naša WCF storitev nosilka strežniške sinhronizacije, je potrebno v konfiguracijski datoteki na spletnem strežniku (web.config) vnesti naslednje nastavitve:

```
  <system.diagnostics>
    <switches>
      <add name="SyncTracer" value="4" />
    </switches>

    <trace autoflush="true">
      <listeners>
        <add name="SyncListener" type="System.Diagnostics.TextWriterTraceListener"
initializeData="c:\Temp\SyncTrace.log"/>
      </listeners>
    </trace>
  </system.diagnostics>
```

Prvi sklop nastavitev (<switches>) uporabimo zato, da se “dokopljemo” do razreda SyncTracer. Temu razredu je potrebno podati nivo beleženja. Ogrodje .NET pozna naslednjih pet različnih nivojev beleženja:

- OFF (0) –brez sporočil
- ERROR (1) – samo napake
- WARN (2) – napake in opozorila

- INFO (3) – info sporočila, napake in opozorila
- VERBOSE (4) – vsa sporočila

Prvi nivo (OFF) beleženje ustavi, ostali štirje pa ujete dogodke pošiljajo na ponor. Ponora na tem mestu še nismo določili. Višji kot je nivo, bolj podrobni so podatki.

Drugi sklop nastavitev je “<trace>”. S to nastavitvijo .NET ogrodju povemo, da želimo beležiti podatke. V sklopu “<listeners>” navedemo ponore. V našem primeru bo to ponor tipa *System.Diagnostics.TextWriterTraceListener* – tekstovna datoteka. V atributu *initializeData* navedemo ponorno datoteko, v katero bi radi beležili podatke. Ponorov imamo lahko več (SQL Strežnik, XML datoteka, TXT datoteka, elektronska pošta,..) [28].

6.1.2 Razred SyncTracer

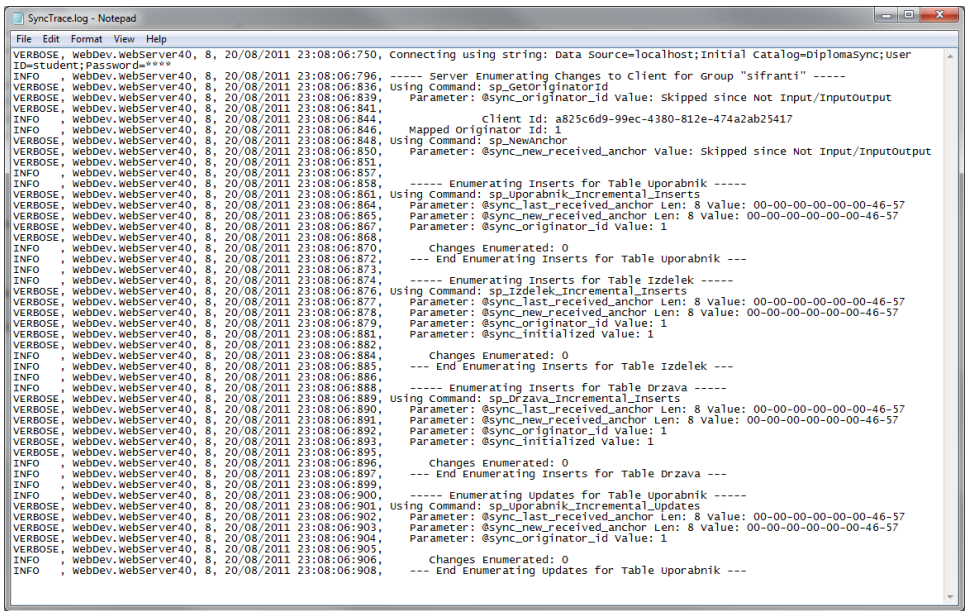
Podobno kot v zgornjem primeru, je tudi v tem primeru potrebno uporabiti razred SyncTracer. Edina razlika je, da smo v prejšnjem primeru uporabili ponor *System.Diagnostics.TextWriterTraceListener*. Možno je, da bi radi formirali svoj format beleženja. To nam omogoča ta način beleženja.

Preko programske kode lahko dostopamo do razreda SyncTracer, s katerim beležimo sporočila. Spodaj je primer zelo preproste metode, ki zabeleži zaznano napako v dnevnik napak.

```
private void Belezenje(string napaka)
{
    SyncTracer.Error(napaka);
}
```

Na tem mestu je potrebno poudariti, da je v nastavitvah še vedno potrebno nastaviti ponor/e (sklop <listeners>) [28].

Spodaj se nahaja izsek iz testnega dnevnika. Nivo beleženja je bil pri tem beleženju nastavljen na na 4 – verbose (vsa sporočila).



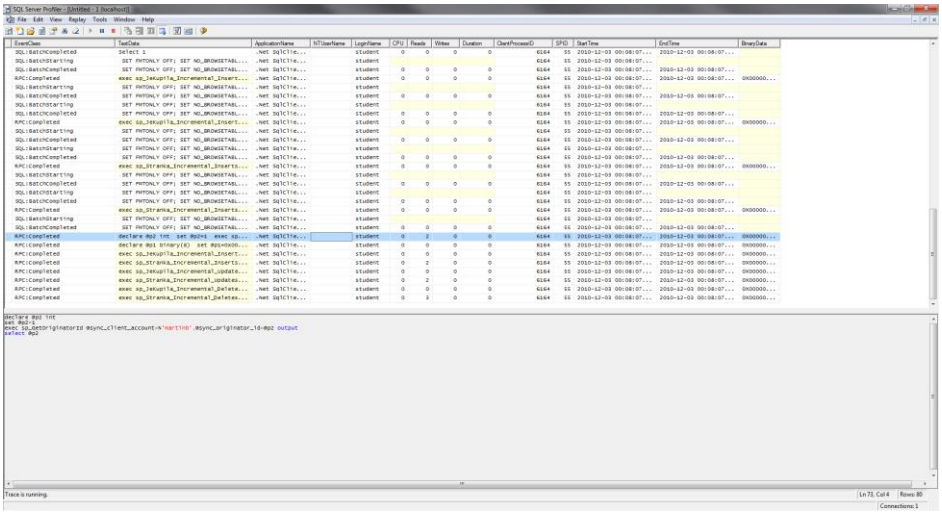
Slika 29: Dnevnik dogodkov

6.2 SQL Profiler

Beleženje dogodkov v prejšnjem poglavju je uporabna reč. Kljub temu se včasih znajdemo v situacijah, ko tovrstno beleženje dogodkov ne zadostuje. Ker v našem demonstracijskem primeru WCF storitev komunicira s SQL strežnikom preko baznih procedur, lahko uporabimo orodje SQL Profiler, ki je del MS SQL Server paketa. S tem orodjem lahko natančno vidimo podrobnosti o posameznem ukazu na SQL strežniku. Na spodnji sliki je prikazana uporaba SQL Profiler-ja.

Na voljo so nam podatki o tem kdaj in kdo je izvrševal posamezen ukaz na strežniku. Vidne so tudi vrednosti vhodnih spremenljivk v bazne procedure ter njihov rezultat.

SQL Profiler je v kombinaciji z beleženjem dogodkov zelo uporabno orodje in je praktično nepogrešljiv v sinhronizacijskih aplikacijah z N-nivojsko arhitekturo, kjer se del aplikacije nahaja na spletnem strežniku. SQL Profiler je uporaben tudi za detekcijo počasnih poizvedb [29].



Slika 30: SQL Profiler

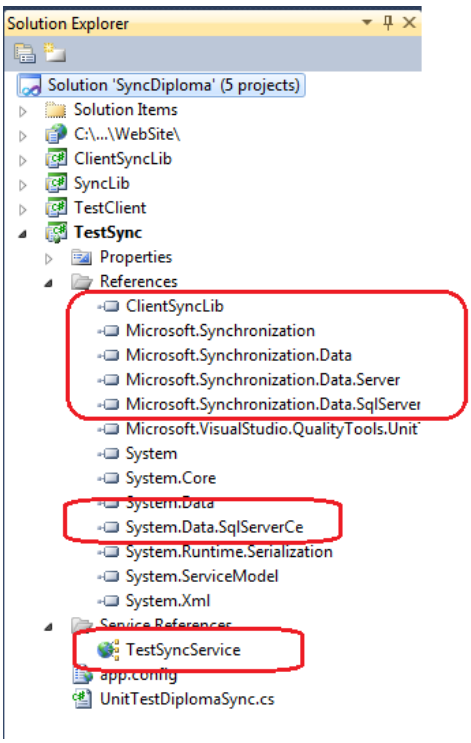
7 Testiranje sinhronizacije

Programsko opremo je možno razvijati na več različnih načinov. Ne glede na način razvoja, pa je potrebno slediti smernicam dobrih praks (angl. *Best practices*) ki narekujejo, da je potrebno vsako funkcionalnost testirati, preden pošljemo aplikacijo v produkcijsko okolje.

Testiranje enot (angl. *Unit testing*) je metoda, s katero testiramo delovanje različnih metod z namenom ugotavljanja ustreznosti in pravilnega delovanja. Enota je najmanjši sklop programske kode, ki ga je možno testirati. V proceduralnih jezikih je to lahko metoda, v objektnem programiranju pa je to največkrat kar cel vmesnik oziroma razred. Testi so vnaprej določeni. Potrebno je vedeti kaj se testira, kakšni so vhodni podatki, kakšni so pričakovani rezultati in podobno. Poznamo različne vrste testov (white box, black box,..). Od vrste testa je lahko odvisno kdo bo test *implementiral*. Pomembno je vedeti, da testiranje ni zgolj klikanje po zaslonskih maskah, ampak programska koda, s pomočjo katere testiramo neko drugo programsko kodo. Običajno razvijalci testirajo kar celotne knjižnice, ki vsebujejo na desetine razredov. Zelo pomembno je, da pisci testov niso seznanjeni z dejansko implementacijo knjižnic ki jih testirajo. Zlato pravilo testiranja je, da manj kot veš o implementaciji nečesa, bolje boš lahko testiral delovanje.
Testi morajo biti med seboj neodvisni. Torej, vsak test si pripravi svoje spremenljivke in jih po uporabi tudi zavrže. Na začetku smo omenili, da obstaja več načinov razvoja programske opreme. Eden izmed takšnih načinov je *Test Driven Development (TDD)*. Pri takšnem razvoju programske opreme razvijalci dejansko najprej razmišljajo o tem, kako bodo neko programsko kodo testirali in kakšni bodo vhodni podatki. Testi so do popolnosti razviti še preden se dejansko začne razvoj aplikacije.
Prednost testiranja programske opreme je velika. Nemalokrat se namreč zgodi, da v ekipi programerjev en sam programer spremeni delovanje neke knjižnice. Sprememba knjižnice lahko povzroči, da se unit testi ne izvršijo pravilno in to predstavlja neke vrste alarm. Unit testi so pri velikih projektih namreč avtomatizirani in se vršijo vsaj enkrat dnevno. Vodja testiranja tako nemudoma izve za napako. Ve tudi katerega testa programska oprema ni prenesla in o tem obvesti odgovorne ljudi.

Za namene demonstracije sem napisal dva preprosta testa sinhronizacije. Na tem mestu bi rad poudaril, da bi se nedvomno našel nekdo, ki bi trdil, da spodaj prikazani testi niso “pravi” unit testi, ker ne testirajo delovanja najmanjše enote. Dejstvo je, da je potrebno pri testiranju določiti obseg le-tega. Razvijalci MSF naj testirajo MSF in vse njegove komponente. Razvijalci, ki pa MSF uporabljajo, naj testirajo malce širše, tako da bodo lahko preverili pravilnost delovanja lastne programske opreme. MSF bi moral biti na tem mestu že testiran in ustrezen.

Spodaj je opisan postopek za implementacijo preproste testne knjižnice.
V naši rešitvi je potrebno ustvariti nov projekt z imenom *TestSync.* Projekt mora biti tipa „Test Project“. Znotraj tega projekta je potrebno referencirati vse potrebne sinhronizacijske knjižnice ter našo WCF storitev, kot je prikazano na spodnji sliki.



Slika 31: Reference v testnem projektu

Naknadno je potrebno implementirati posamezne teste. Teste uvrščamo v testne razrede. Primer testnega razreda je v našem primeru *UnitTestDiplomaSync.cs.* Vsako testno metodo in vsak testni razred je potrebno posebej označiti z atributom **[TestMethod]** oziroma **[TestClass]**. Tako .NET ve katere metode mora pognati ko zaganja testno knjižnico.

Poleg testnih metod .NET testno ogrodje pozna še dve metodi: inicializacijsko metodo (atribut **[TestInitialize]** in čistilno metodo **[TestCleanup]**). V inicializacijski metodi globalnim spremenljivkam nastavimo vrednost. Ta metoda se izvrši pred vsakim testom. V čistilni metodi implementiramo operacije, ki se morajo zgoditi po dejanskem testu. Ta metoda se izvrši po vsakem testu.

Spodaj se nahajajo demonstracijska inicializacijska metoda, čistilna metoda ter tri globalne spremenljivke:

```
        ClientSyncLib.DiplomaSyncAgent syncAgent;
        string serverConnectionString;
        string localConnectionString;

        [TestInitialize]
        public void Initialize()
        {
            serverConnectionString = "Data Source=localhost;Initial
Catalog=DiplomaSync;User ID=student;Password=student";
            localConnectionString = @"Data Source='c:\Temp\ClientDB.sdf';
Password=diploma123";

            syncAgent = new ClientSyncLib.DiplomaSyncAgent(localConnectionString,
"martinb", new TestSyncService.ServerSyncProviderClient());

            string clientConn = localConnectionString;
```

```
        using (SqlCeConnection sqlconn = new SqlCeConnection(clientConn))
        {
            if (!File.Exists(sqlconn.Database))
            {
                SqlCeEngine sqlCeEngine = new SqlCeEngine(clientConn);
                sqlCeEngine.CreateDatabase();
            }
        }
    }

    [TestCleanup]
    public void Cleanup()
    {
        syncAgent.Dispose();
    }
```

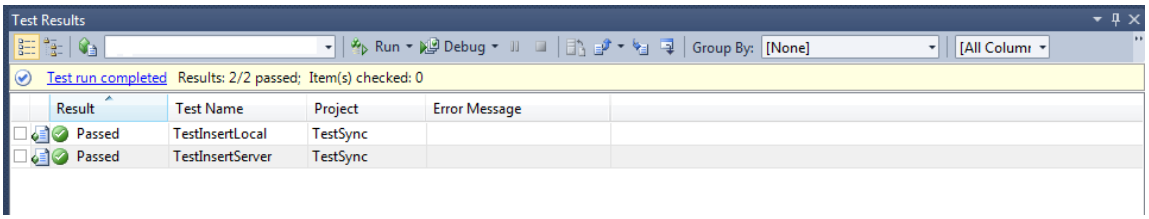
V inicializacijski metodi nastavimo poveve na strežniško in lokalno podatkovno bazo ter inicializiramo WCF storitev. Hkrati tudi preverimo, če na izbrani lokaciji morda še ne obstaja lokalna podatkovna baza.

Spodnja izvorna koda predstavlja en test. S tem testom želimo testirati ali se na strežniku dodane stranke ustrezno prenesejo na odjemalca. Ta test bi bilo smiselno nadgraditi še z preverjanjem posameznih atributov v tabelah, vendar ker gre za demonstracijsko aplikacijo tega nisem naredil.

```
    [TestMethod]
    public void TestInsertServer()
    {
        // sinhroniziraj - zato da bosta repliki enaki
        syncAgent.Synchronize();

        // dodaj 4 Stranke na strežnik
        var ids = InsertStranka(4, Destination.Server);

        // preveri če si dobil 4 IDje
        Assert.AreEqual(ids.Count, 4);

        // sinhroniziraj
        var stats = syncAgent.Synchronize();

        // preveri če so bile vse 4 stranke prenešene
        Assert.AreEqual(stats.DownloadChangesApplied, 4);

        // odstrani stranke
        DeleteStranka(ids, Destination.Server);

        // ponovno sinhroniziraj
        stats = syncAgent.Synchronize();

        Assert.AreEqual(stats.DownloadChangesApplied, 4);
    }
```

Pomembno je, da bralec opazi ukaze razreda **Assert**. V tem statičnem razredu se namreč nahajajo metode, ki na različne načine testirajo različne spremenljivke. V našem primeru uporabljamo samo metodo **AreEqual**, vendar je teh testnih metod še precej več.

Ko poženemo testno knjižnico, nam Visual Studio v oknu *Test Results* prikaže rezultate testov. Naša želja je seveda, da se vsi testi zaključijo uspešno. Na spodnji sliki je prikazan rezultat dveh testov, ki sta se zaključila uspešno. [30]



Slika 32: Rezultat testiranja

8 Pogled v prihodnost MSF

Trenutna verzija MSF je 4.0 CTP (Community Technological Preview). Le-ta je osnovana na verziji 2.1.[9]

8.1 Nov sinhronizacijski protokol - OData

Verzija 4.0 definira nov odprti sinhronizacijski protokol imenovan **OData**. S pomočjo OData in sinhronizacijskega ogrodja, bo sinhronizacija potekala malce drugače, kot smo opisovali v prejšnjih poglavjih. Vsa sinhronizacijska logika bo od sedaj naprej shranjena na strežniku. Odjemalci se bodo s pomočjo protokola OData povezovali na neko storitev, ki bo zmožna vseh za sinhronizacijo potrebnih enumeracij in operacij. To je zelo pomembno, saj v tem primeru odjemalci ne potrebujejo sinhronizacijskega ogrodja ampak samo zmožnost beleženja lokalnih podatkov. Microsoft je s to potezo naredil velik korak naprej. OData je namreč osnovan na podlagi HTTP in je kot tak uporaben praktično na vseh napravah, na katerih bi bila sinhronizacija podatkov smiselna.

8.2 Sinhronizacija v oblaku

Druga novost je podpora **SQL Azure**. SQL Azure je SQL Server v oblaku. Glede na trende in vložen trud in kapital s strani gigantov je pričakovati, da bo računalništvo v oblaku igralo pomembno vlogo v prihodnosti.

8.3 Sinhronizacijski prestrezniki

Tretja novost je ravno tako dobrodošla kot prvi dve. Veliko razvijalcev se je pritoževalo nad slabo podporo poslovni logiki v samem MSF. Vedeti je potrebno, da se zaradi hitrejšega in lažjega vzdrževanja poslovna logika seli vedno bolj „dol“ – stran od odjemalca in čim bližje podatkom oziroma podatkovni bazi. MSF omogoča klic baznih procedur ampak to velikokrat ni dovolj. Včasih je potrebno ob nastanku ali odstranitvi nekega zapisa v podatkovni bazi, narediti še kakšno drugo, s sinhronizacijo popolnoma nepovezano operacijo. Le-to bi bilo potemtakem na podatkovni bazi potrebno umestiti v bazne prožilce ali pa se posluževati časovno tempiranih nalog (angl. *scheduled jobs*). To je seveda prava mora za razvijalce, saj stvari postanejo zelo hitro zelo nepregledne, kar privede do napak v delovanju in v najslabšem primeru izpada dobička. Microsoft je ta problem elegantno rešil z **sinhronizacijskimi prestrezniki** (angl. *sync interceptors*).
Sinhronizacijski prestrezniki so pravzaprav mehanizem, ki razvijalcem omogoča izvajanje lastnih operacij med samimi fazami sinhronizacijskega postopka. Stvar je za uporabo razmeroma preprosta. Prestrezniki za implementirani kot metode na strežniški strani. MSF pozna tri vrste prestreznikov:

[9] Microsoft je zaradi združljivosti z MS SQL Server CE, ki je trenutni v verziji 3.5, namenoma dvignil verzijo MSF iz 2.1 na 4.0 namesto na 3.0. Poleg napisanega, je bila tudi znotraj samega MSF manjša zmeda z verzijami. Odločitev za skok na verzijo 4.0 tako ni bila neosnovana.

- **SyncRequestInterceptor** – Ta koda se izvrši ravno preden se vhodna zahteva prične izvajati.
- **SyncResponseInterceptor** – Ta koda se izvrši takoj zatem ko se vhodna zahteva izvrši in je potrebno rezultate vrniti na odjemalca.
- **SyncConflictInterceptor** – Ta koda se izvrši takrat, kadar pride do konfliktov.

Sintaksa za glavo prestrezne metode(prestreznike), je sledeča (v jeziku C#):

```
public void ImeMetode(SyncOperationContext operationContext)
```

Tik nad glavo metode je potrebno uporabiti atribut za označitev tipa prestreznika v katerem navedemo obseg (angl. *scope*) prestrezanja in smer prestrezanja. Prestrezniki lahko prestrežejo zahteve za Upload, Download in obojestranske zahteve (enumeracija **SyncOperations**).

Spodaj se nahaja izvorna koda za glavo metode prestrežnika za tabelo *Stranka.*

```
[SyncResponseInterceptor("Stranka", SyncOperations.Download)]
public void Stranka_Interceptor(SyncOperationContext operationContext)
```

Zaradi atributa *SyncResponseInterceptor* se bo ta metoda izvršila takrat, kadar bo prišla zahteva za zajem (SyncOperations.Download) podatkov iz strežnika za tabelo *Stranka*. V sam obseg prestrezanja lahko zavedemo več kot eno tabelo, vendar je tabele med seboj potrebno ločiti z vejico [31].

8.4 Podporna orodja

Microsoft je v verzijo 4.0 dodal tudi nekaj več podpore razvijalcem. Vključena so namreč tudi orodja za diagnostiko objavljenih (angl. *deployed*) sinhronizacijskih rešitev, orodje za grafično in konzolno generiranje in objavljanje sinhronizacijskih storitev (**SyncSvcUtil.exe** in **Tooling Wizard UI**) [32].

9 Sklepne ugotovitve

V tem diplomskem delu smo uporabili MSF za razvoj preproste sinhronizacijske rešitve. Nedvomno obstaja mnogo potencialnih scenarijev, kjer bi lahko uvedli tovrstne mobilne aplikacije. V kombinaciji z CRM, SCM, ERP, ipd. rešitvami je lahko sinhronizacija mobilnih odjemalcev z centralnim strežnikom zelo uporabna in učinkovita. Ko trgovski potnik pri stranki pobere naročilo, gre to naročilo lahko s pomočjo podpornega informacijskega sistema praktično nemudoma v nadaljno obdelavo. Storilnost in učinkovitost poslovanja se z uporabo sinhronizacijskih rešitev nedvomno poveča.

S stališča nekega večjega podjetja so sinhronizacijske platforme kot je MSF lahko strateško zelo pomembna reč, saj poleg že omenjenih prednosti omogočajo tudi konsolidacijo različnih podatkovnih virov iz različnih poslovnih funkcij v en sam centralni vir. Vedeti je potrebno, da se vedno več podjetij odloča za uvedbo podatkovnega skladišča (angl. *Data warehouse*). V podatkovnih skladiščih se nahajajo gore prečiščenih in statistično uporabnih podatkov. Več kot imamo podatkov, boljše in bolj inovativne statistike lahko izvajamo. Če podjetje statistično obdela vse ključne točke svojega poslovanja, je veliko lažje kontrolirati stroške in povečevati storilnost, h čemur pa stremi vsako podjetje.

MSF je sinhronizacijsko ogrodje, ki omogoča enostaven razvoj sinhronizacijskih rešitev. Kljub njegovi enostavnosti pa je ogrodje dovolj zmogljivo in razširljivo, da lahko podpre praktično vsak podatkovni vir. MSF omogoča tako enostavno kot tudi kompleksno razreševanje sinhronizacijskih konfliktov, kar minimizira morebitne napake v poslovanju. Hkrati s pomočjo sinhronizacijskih prestrežnikov omogoča zelo elegantno implementacijo poslovne logike in integracijo v ostale podporne informacijske sisteme. V prihodnosti bo ogrodje podpiralo vse naprave, ki se bodo zmožne povezati na splet. Ta novost, v kombinaciji z brezplačnim licenciranjem, daje MSF izreden potencial in konkurenčno prednost.

Že zdavnaj smo prečkali točko, ko si podjetja lahko privoščijo nekonkurenčnost. Prostora za slabe poslovne odločitve v danem trenutku ni. Iz napisanega torej lahko povzamemo, da je potrebno za uspešno poslovanje izbrati pravo informacijsko podporo,ki je hkrati ekonomsko upravičljiva. MSF je lep primer, kako lahko cenovno dostopna tehnologija neposredno pripomore k izboljšanju poslovanja podjetja.

Priloge

Vse priloge so dostopne na zgoščenki, ki je priložena k diplomskemu delu.

Priloga A – T-SQL skripta za podatkovno bazo in predpripravljena podatkovna baza MS SQL Server
Priloga B – Logični podatkovni model (PowerDesigner)
Priloga C – Izvorna koda v jeziku C# (.NET)
Priloga D – Namestitveni paketi za MSF

Literatura

[1] J. Kanjilal, R. Singh, *Pro Sync Framework*, Apress, 2009.

[2] (2011) Xchange Network, *XC Bridge*. Dostopno na: http://www.xcnetwork.com/xcbridge.jsp.

[3] (2011) eMarketer, *Two in Five Mobile Owners Use Internet on the Go*. Dostopno na: http://www.emarketer.com/Article.aspx?R=1008553.

[4] (2011) International Telecommunication Union, *Data and Statistics*. Dostopno na: http://www.itu.int/ITU-D/ict/statistics/material/excel/2010/Global_ICT_Dev_00-10.xls.

[5] (2011) Wikipedia, *Smartphone*. Dostopno na: http://en.wikipedia.org/wiki/Smartphone.

[6] (2011) Wikipedia, *ADO.NET*. Dostopno na: http://en.wikipedia.org/wiki/ADO.NET.

[7] (2008) V. Shukla, *ADO.NET Sync Services in N Tier Architecture*. Dostopno na: http://www.codeproject.com/KB/smart/sync_services.aspx.

[8] (2008) S. Nair, *Microsoft Sync Framework - A primer to the file sync provider*. Dostopno na: http://www.c-charpcorner.com/UploadFile/mail2sharad/FileSyncProvider01012008193150PM/FileSyncProvider.aspx.

[9] (2011) Microsoft MSDN, *How to: Use Session Variables*. Dostopno na: http://msdn.microsoft.com/en-us/library/bb902811.aspx.

[10] (2007) D. Mulder, *Hosting and Consuming WCF Services*. Dostopno na: http://msdn.microsoft.com/en-us/library/bb332338.aspx.

[11] (2008) Microsoft MSDN, *.NET Compact Framework 3.5*. Dostopno na: http://www.microsoft.com/download/en/details.aspx?id=65.

[12] (2011) Microsoft MSDN, *.NET Framework*. Dostopno na: http://msdn.microsoft.com/en-us/netframework/aa569263.

[13] (2009) Microsoft MSDN, *How to choose a data synchronization technology - offline and collaboration*. Dostopno na: http://msdn.microsoft.com/en-us/sync/cc470041.

[14] (2011) Microsoft MSDN, *How to: Configure Data Synchronization in an Application*. Dostopno na: http://msdn.microsoft.com/en-us/library/bb384585.aspx.

[15] (2011) Microsoft MSDN, *Introduction to Microsoft Sync Framework*. Dostopno na: http://msdn.microsoft.com/en-us/sync/bb821992.

[16] (2009) Microsoft MSDN, *Introduction to Microsoft Sync Framework File Synchronization Provider*. Dostopno na: http://msdn.microsoft.com/en-us/data/bb887623.

[17] (2010) Microsoft MSDN, *Microsoft Sync Framework SDK*. Dostopno na: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=23217.

[18] (2010) Microsoft MSDN, *Occasionally Connected Applications*. Dostopno na: http://msdn.microsoft.com/en-us/library/bb384572.aspx.

[19] (2008) Oracle, *Oracle Database Lite 10g*. Dostopno na: http://www.oracle.com/technetwork/database/database-lite/twp-lite-10gr3-129866.pdf.

[20] (2011) Microsoft MSDN, *SQL Server Change Tracking*. Dostopno na: http://msdn.microsoft.com/en-us/library/cc305322.aspx.

[21] (2011) Microsoft MSDN, *SQL Server Compact 3.5 and Visual Studio*. Dostopno na: http://msdn.microsoft.com/en-us/library/aa983341.aspx.

[22] (2011) Microsoft MSDN, *SQL Server Compact Edition*. Dostopno na: http://www.microsoft.com/sqlserver/en/us/editions/compact.aspx.

[23] (2010) Microsoft MSDN, *Visual Studio 2010 download*. http://msdn.microsoft.com/en-us/vstudio/bb984878.aspx.

[24] (2011) S. Epps Rotman, *Tablets Will Grow As Fast As MP3 Players*. Dostopno na: http://www.forrester.com/rb/Research/tablets_will_grow_as_fast_as_mp3/q/id/58409/t/2.

[25] (2008) Microsoft MSDN, *How to: Handle Data Conflicts and Errors*. Dostopno na: http://msdn.microsoft.com/en-us/library/bb725997.aspx.

[26] (2008) J.D. Meier, *How To – Use Certificate Authentication and Message Security in WCF*. Dostopno na: http://wcfsecurity.codeplex.com/wikipage?title=How%20To%20-%20Use%20Certificate%20Authentication%20and%20Message%20Security%20in%20WCF%20calling%20from%20Windows%20Forms.

[27] (2007) Microsoft Learning Center, *SQL Server 2008 Security Overview for Database Administrators*. Dostopno na: http://download.microsoft.com/download/a/c/d/acd8e043-d69b-4f09-bc9e-4168b65aaa71/SQL2008SecurityOverviewforAdmins.docx.

[28] (2010) Microsoft MSDN, *How To: Extend Business Logic on Server using Interceptors*. Dostopno na: http://msdn.microsoft.com/en-us/library/gg299068%28v=sql.110%29.aspx.

[29] (2010) Microsoft MSDN, *Microsoft Sync Framework 4.0 October 2010 CTP*. Dostopno na: http://www.microsoft.com/download/en/details.aspx?id=12012.

[30] (2011) Microsoft MSDN, *How to: Trace the Synchronization Process*. MSDN. Dostopno na: http://msdn.microsoft.com/en-us/library/cc807160.aspx.

[31] (2008) A. Jana, *Monitoring with SQL Profiler*. Dostopno na: http://www.codeproject.com/KB/dotnet/SQLServerProfiler.aspx.

[32] (2010) Microsoft MSDN, *Walkthrough: Creating and Running Unit Tests*. Dostopno na: http://msdn.microsoft.com/en-us/library/ms182532.aspx.