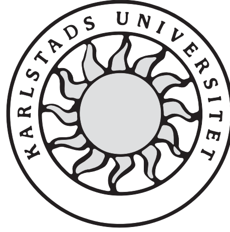


Datavetenskap

Magnus Malmgren och Assadullah Obaid

Replikering av databaser över Internet

Examensarbete, C-nivå

2003:07

Replikering av databaser över Internet

Magnus Malmgren och Assadullah Obaid

© Magnus Malmgren, Assadullah Obaid och Karlstads universitet

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Magnus Malmgren

Assadullah Obaid

Godkänd, 2003-06-03

Handledare: Katarina Asplund

Examinator: Stefan Lindskog

Sammanfattning

Replikering innebär att man skapar en miljö där man distribuerar flera kopior av samma informationsmängd till flera databaser på olika ställen. Distributionen kan göras periodiskt vilket leder till att de replikerade databaserna blir mer autonoma.

Denna rapport beskriver vårt arbete, vilket är ett examensarbete i datavetenskap på C-nivå. Rapporten innehåller dels våra undersökningar om befintlig teknik för att replikera databaser i Microsoft SQL Server, och dels beskriver den vår egen lösning på replikeringsproblemet. Vi börjar med att undersöka vad replikering egentligen är, och därefter beskriver vi Microsoft SQL Servers inbyggda replikeringsmetoder över Internet. Vi går vidare med att undersöka alternativa lösningar för att replikera databaser. Då uppdragsgivaren även är intresserad av att överföra andra filer än replikeringsfiler mellan sina servrar, tittar vi närmare på några olika metoder (FTP och HTTP) för att överföra filer.

Design och implementation av en exempel-applikation ingick också i uppdraget, så därför avslutar vi rapporten med att gå genom hur vi har designat och implementerat vår exempel-applikation.

Replication of databases over the Internet

Abstract

Replication means that you create an environment where you can distribute several copies of the same amount of information to several databases in different locations. The distribution could be made periodic, in which case the database becomes more autonomous.

This thesis describes our work, which is a C level Bachelor's Project in Computer Science. The thesis contains a survey about existing technology for replicating databases in Microsoft SQL server. It also describes our own solution to the replication problem.

We start by examining what replication actually is, and then we describe Microsoft SQL server's built-in replication methods over the Internet. We then go further and examine alternative solutions for replicating databases. Since our assigner is interested in transferring other files then replication files, we also investigate a number of different methods (FTP and HTTP) for file transfer.

Designing and implementing an example application was also a part of the project, so therefore we conclude the thesis with a description of how we have designed and implemented the example application.

Tack

Vi vill tacka Katarina Asplund för hennes engagemang och noggrannhet i samband med rapportskrivningen. Per Fasth på Miller Graphic som gett oss uppdraget har alltid funnits till hands och tagit sig tid för frågor och diskussioner. Slutligen vill vi tacka TRR (Trygghetsrådet) i Karlstad där vi har haft vår arbetsplats.

Innehållsförteckning

1 Inledning ... 1
1.1 Företaget ... 1
1.2 Uppdraget ... 1
1.3 Disposition ... 2
2 Våra förutsättningar ... 3
2.1 Önskemål och krav ... 3
2.2 Vårt val av utvecklingsmiljö ... 4
3 Replikering ... 5
3.1 Replikering och synkronisering ... 5
3.1.1 Vad är replikering
3.1.2 Vad är synkronisering
3.2 Replikering i Microsoft SQL Server ... 6
3.3 Replikeringsagenterna ... 6
3.3.1 Distributions-agenten
3.3.2 Loggläsar-agenten
3.3.3 Snapshot-agenten
3.3.4 Merge-agenten
3.4 Replikeringsmetoder ... 7
3.4.1 Snapshot-replikering
3.4.2 Transaktions-replikering
3.4.3 Merge-replikering
3.5 Hur man väljer distributionsmetod ... 12
3.5.1 När skall man använda replikering?
3.5.2 Vilken typ av replikering använde vi?
4 Filöverföring med FTP respektive HTTP ... 14
4.1 Vad är ett protokoll? ... 15
4.2 FTP (File Transfer Protocol) ... 16
4.3 HTTP (Hyper Text Transfer Protocol) ... 18
4.3.1 Hur HTTP fungerar
4.3.2 Olika versioner av HTTP och överföringskapacitet
4.3.3 Säkerhet
4.3.4 Fördelar med HTTP
4.3.5 Nackdelar med HTTP
4.4 FTP kontra HTTP ... 21

5 Verktyg och utvecklingsmiljö ... 23

5.1 Microsoft Visual Studio .NET ... 23

5.1.1 WinInet API

5.1.2 Windows-tjänster

6 Beskrivning av konstruktionslösningen ... 25

6.1 Loggläsaren ... 26

6.1.1 Läsa transaktioner från transaktionsloggen.

6.1.2 Lagring av transaktionsdata

6.2 Överföring av data ... 28

6.2.1 Implementationen av FTP-klienten

6.3 Överföring av data från XML-filer till databasen ... 32

6.3.1 Läsning av XML-filerna

6.3.2 Uppdatering av databasen

7 Uppbyggnad av programmet ... 34

7.1 Inledning ... 34

7.2 Logg-läsaren ... 34

7.3 FTP-klienten ... 35

7.3.1 Service1

7.3.2 ErrorOut

7.3.3 Creceipt

7.3.4 Errorlog

7.3.5 PublicData

7.3.6 WinInetMethods

7.4 XML-läsaren ... 37

8 Tester ... 39

9 Slutsatser ... 40

Referenser ... 42

A Bilaga ... 43

A.1 Modify ... 43

A.2 Insert. ... 44

A.3 Exempel på en XML-fil med en transaktion. ... 45

A.4 Tolkning av hexsträng ... 46

Figurförteckning

Figur 1. En ögonblicksbild av hela databasen sänds till abonnenterna 7
Figur 2. Snapshot-replikering 8
Figur 3 Transaktions-replikering 10
Figur 4. Merge-replikering. 11
Figur 5. Exempel på två olika protokoll 15
Figur 6. Bild över FTP-klient och FTP-server. 16
Figur 7. Bild över TCP-anslutningarna för FTP- protokollet. 17
Figur 8. Applikationernas samverkan 25
Figur 9. Bild över hur data skulle skickas i vår första design. 29
Figur 10. Flödesschema för den första designen av FTP-klienten. 29
Figur 11. Filöverföringen för den första designen. 30
Figur 12. Flödesschema för den modifierade FTP-klienten. 31
Figur 13 Exempel på element och data i en XML-fil. 33

1 Inledning

I den här uppsatsen presenterar vi vårt examensarbete. Vi börjar detta kapitel med en kort presentation av företaget och därefter beskriver vi vad vårt uppdrag har varit. Sist i kapitlet berättar vi hur uppsatsen är disponerad.

1.1 Företaget

Företaget vi jobbat åt heter Miller Graphics Scandinavia AB. Det ligger i Sunne och arbetar med utveckling, tillverkning och marknadsföring av klichéer, lasergraverade mönstervalsar, gummibelagda valsar, hylssystem och raklar. Företagen som använder dessa produkter finner man inom den grafiska industrin som förädlar plast och papper till bl.a. olika typer av förpackningar. Vid fabriken i Sunne finns nära 7000 m^2 produktionsyta och cirka 110 anställda med kunskap om grafisk teknik och den grafiska branschen. De erbjuder repro, tryckformar och gummivalsar inom affärsområdena grafisk och teknisk industri till kunder huvudsakligen på den skandinaviska marknaden. Det finns även en liten IT-avdelning på Miller. De har under några år utvecklat ett eget affärssystem som heter Bizcon. Systemet är nu så utvecklat att det har börjat säljas även till kunder utanför Miller. Det uppdrag vi fick kommer förhoppningsvis att bli en del av detta affärssystem. Mer om detta nedan.

1.2 Uppdraget

När Millers kunder beställde varor av Miller så kunde de hålla koll på sina beställningar genom att surfa in på en webbsida och se statusen på sin beställning. När de beställde flera produkter så var det inte säkert att alla produkter producerades på samma fabrik. Kunden kunde då behöva surfa till olika sidor beroende på vilka fabriker som producerade respektive produkt. För att förenkla för kunden ville man därför centralisera så att kunden bara behövde gå in på en sida och därifrån kontrollera statusen på sina beställningar oavsett var produkterna producerades. Det var här som vi kom in. För att underlätta för kunden ville Miller ha en databas som innehöll data om ordrar från flera fabriker samlade i en databas. Denna databas skulle alltså uppdateras från de andra databaserna. Att på detta sätt ha samma data lagrad på flera ställen kallas replikering. Uppgiften vi fick var att ta fram en metod för replikering av

fabrikernas databaser till en gemensam databas som kunderna kunde söka i. Dessutom skulle vi undersöka ett bra sätt för Miller att överföra filer mellan sina servrar.

1.3 Disposition

Rapporten är upplagd så att den skall vara pedagogisk och så att man på ett naturligt sätt skall kunna följa våra tankegångar fram till en färdig lösning. Först och främst går vi igenom våra förutsättningar, det vill säga vilka krav och önskemål som fanns. Vi tar även upp vårt val av utvecklingsmiljö och allt detta sker i kapitel 2. Därefter ställer vi oss i kapitel 3 frågan vad replikering är och vad replikering är bra till. Data som replikeras skall ju föras över mellan datorer och vi tar upp olika sätt som överföring kan ske på i kapitel 4. De verktyg och den utvecklingsmiljö som vi har använt tas upp till en viss del i kapitel 5. Kapitel 6 och 7 beskriver konstruktionen och designen av den exempelapplikation som vi har utvecklat. De tester vi har utfört beskrivs i kapitel 8. Slutsatserna tas upp i kapitel 9.

2 Våra förutsättningar

Vi hade nästan fria händer att designa lösningen. Miller hade dock några önskemål samt några krav som vi skulle ta hänsyn till. I detta kapitel beskriver vi vilka dessa var.

2.1 Önskemål och krav

Uppgiften var att på ett för användaren (Miller) enkelt sätt implementera replikering och synkronisering av databaser. Designen skulle inte ställa ytterligare krav på Millers IT-miljö än de nämnda nedan. Uppgiften var uppdelad i två delar. Dels skulle vi utreda olika sätt att utföra uppgiften på dels skulle vi göra ett program som utförde uppgiften. Beroende på vad vi kom överens med Miller om så kunde det bli så att lösningen inte avspeglade den enligt rapporten bästa metoden.

Krav på utvecklingsmiljö:

- Eftersom Miller uteslutande använder sig av Microsofts operativsystem Windows 2000 Server så skulle vi utveckla för detta operativsystem.
- Det skulle om möjligt inte medföra extra konfiguration att installera programvaran.
- Miller har brandväggar för att skydda sin IT-miljö och designen skulle inte medföra att de behöver konfigurera om dessa brandväggar.

Följande punkter är krav på själva programmet:

- Programmet skulle klara av att föra över filer som kunde vara upp till 20 megabyte stora.
- Programmet skulle kunna ta fram ändringar gjorda i en databas. Dessa ändringar skulle föras över till en annan dator utan att stoppas av brandväggar. De data som fördes över skulle sedan föras in i en databas.

Följande önskemål var också definierade:

- Lösningen behövde inte fungera skarpt i Millers verkliga miljö tillsammans med deras affärssystem utan skulle tjäna som grund för Miller att bygga vidare på och anpassa till deras egna behov.
- Då programmet skulle kunna anpassas av Miller för deras syfte borde de delar av programmet som de skulle kunna förändra vara skrivet i ett språk som Miller har kompetens i.
- Miller har för avsikt att uppgradera till .NET-miljö och har även för avsikt att utveckla sin programvara med hjälp av Visual Studio.Net(VS.Net). Ett önskemål var därför att vi använde nämnda programvara i utvecklingen.
- Miller använder SQL Server 2000 som databas för närvarande. Det var ett önskemål men inget krav att lösningen skulle vara oberoende av vilken databas som används. Det vill säga om de byter till en annan databas så var det ok om lösningen fungerar då också.

2.2 Vårt val av utvecklingsmiljö

Med hänsyn till ovanstående krav och önskemål kom vi fram till följande:

- Vi använde VS.Net och Windows 2000 Server som utvecklingsmiljö.
- Programmeringen skulle ske i Visual Basic.Net.
- Valet av operativsystem känns självklart och är onödigt att diskutera.
- Valet av VS.Net (se kapitel 5.1) har vi gjort då det är en miljö som underlättar utveckling. Då ingen av oss har erfarenhet av programmering i Windowsmiljö ger det även en viss hjälp för oss noviser. Det känns även som att det är ett ypperligt tillfälle att få fördjupa sig i en miljö som är relativt okänd för oss båda. Att skriva program i Windowsmiljö är lite annorlunda än att skriva C-kod i emacs.

Språket vi programmerade i är Visual Basic.NET (VB.Net). Språkvalet har vi gjort då det finns kompetens hos Miller att fortsätta utvecklingen av programmet samt att ingen av oss har programmerat i detta globalt sett mycket använda språk. Det är alltså ett bra tillfälle att prova på detta språk.

3 Replikering

I detta kapitel går vi igenom vad replikering och synkronisering är. Vi beskriver vad det finns för replikering i Microsoft SQL Server eftersom vi kommer att jobba med detta program. Efter det följer en genomgång av de agenter som användes vid replikering. Vi tar även upp vad det finns för olika sorters replikering samt vad dessa är bra respektive dåliga på.

3.1 Replikering och synkronisering

3.1.1 Vad är replikering

Replikering betyder i detta sammanhang att man gör kopior av en databas. Dessa kopior kan spridas och användas till exempel av folk som reser. Ett vardagligt exempel kan vara att man ofta lagrar telefonnummer i sin mobiltelefon. Hade man bara kunnat nå telefonnummer i en enda databas så skulle det kunna bli stora problem om till exempel servern kraschade. Replikering betyder inte att man tvunget måste kopiera hela databasen. I en mobiltelefon lagrar man ju bara en delmängd av de telefonnummer som finns. Ibland kanske någon byter telefonnummer och då kan man behöva uppdatera dessa uppgifter mellan de olika kopiorna. Då får man synkronisera kopiorna.

Två uttryck som man bör känna till i sammanhanget är autonomi och latens. Autonomi är frågan om hur oberoende repliken är. Man kan jämföra med exemplet ovan med telefonnummer i en mobiltelefon. Man behöver inte uppdatera dessa telefonnummer så ofta och därför har databaskopian i telefonen hög autonomi.

Latens handlar om hur lång tid det tar innan kopian är uppdaterad från det att originalet är ändrat. Om man har en fast uppkoppling så kan latensen bli liten då man kan uppdatera kopian omedelbart. Har man däremot kanske ingen uppkoppling alls så kanske man bara kan uppdatera kopian ibland och då blir latensen hög.

3.1.2 Vad är synkronisering

Synkronisering går ut på att man för över ändringar mellan kopiorna och uppdaterar dem så att alla har lika innehåll. Synkronisering kan även innebära att ändringar av databasens struktur, till exempel tabelländringar, förs över till kopiorna.

3.2 Replikering i Microsoft SQL Server

Microsoft SQL server tillhandahåller tre olika typer av replikering, nämligen snapshot-replikering, transaktions-replikering och merge-replikering. För att utföra dessa olika replikeringar så använder SQL Server sig av ett antal olika agenter. Det är dessa agenter som utför det arbete som de olika transaktionmetoderna kräver. Man kallar den server som äger den databas som skall replikeras för publicerare och den som tar emot data för abonnent. Vid vissa typer av replikering skickas data från abonnenten till publiceraren men det är fortfarande publiceraren som "äger" databasen. De agenter som anges nedan kan arbeta på två sätt. Antingen kan de skicka data från publiceraren till abonnenten, eller så ber de publiceraren skicka ändringarna. Dessa olika metoder kallas för push respektive pull [1].

3.3 Replikeringsagenterna

3.3.1 Distributions-agenten

Distributions-agenten för över data från publiceraren till abonnenten.

3.3.2 Loggläsar-agenten

Loggläsar-agenten flyttar på transaktionerna som är markerade för replikering från transaktions-loggen till distributionsdatabasen. Distributionsdatabasen är en databas där transaktioner som ännu inte har distribuerats till abonnenten lagras.

3.3.3 Snapshot-agenten

Snapshot-agenten skapar tabeller och data från den artikel som skall replikeras. Dessa lagras i distributionsserverns katalog.

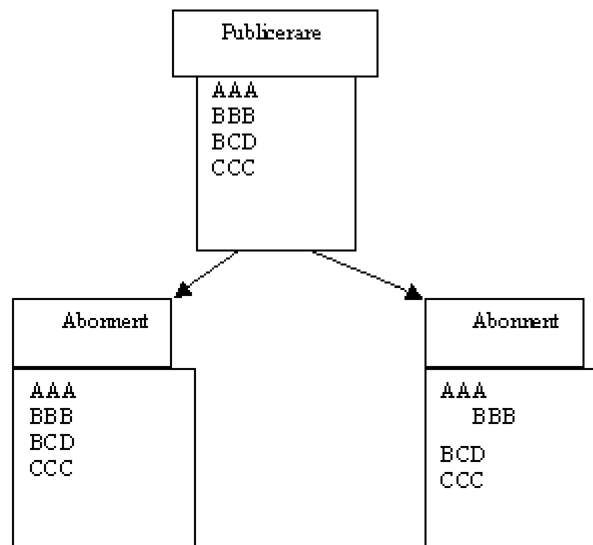
3.3.4 Merge-agenten

Merge-agenten hämtar ändringar både från publiceringsservrarna och abonnenterna. Den har även till uppgift att lösa konflikter som kan uppstå när data skall läggas in i tabeller.

3.4 Replikeringsmetoder

3.4.1 Snapshot-replikering

Vid snapshot-replikering tas en exakt kopia av databasen vid en specifik tidpunkt och sedan skickas kopian till abonnenten/abonnenterna. Snapshot-replikering är den lättaste replikeringsmetoden att konfigurera i Microsofts SQL-server. Då man använder snapshot-replikering betraktas den mottagna informationen av abonnenten som skrivskyddad. Enligt [2] beror detta på att data inte kommer att återsändas till publiceringsdatabasen och om ett nytt snapshot tas emot så raderas alla tidigare ändringar [2]. För en illustration av snapshot-replikering se Figur 1.

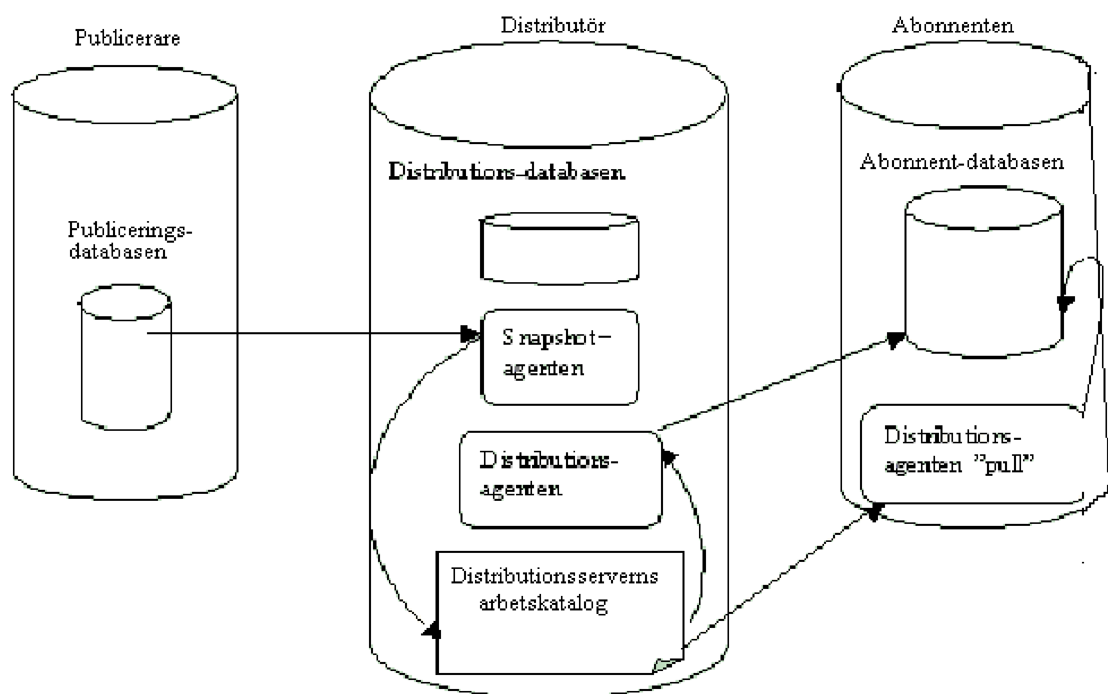


Figur 1. En ögonblicksbild av hela databasen sänds till abonnenterna

3.4.1.1 Agenter vid snapshot-replikering

Vid snapshot-replikering används två agenter för att hantera replikering och distribution av data, nämligen snapshot-agenten och distributions-agenten. Om det är en pull-replikering (abonnenten beger att få data) så befinner sig distributions-agenten hos abonnenten och om det är en push-replikering så befinner sig distributions-agenten hos publiceraren.

Snapshot-agentens uppgift är att läsa den publicerade informationen samt att skapa ett tabellschema till distributionsserverns arbetskatalog. Distribution-agentens uppgift är att läsa det skapade schemat och att bygga om tabellerna hos abonnenten. Den har också till uppgift att flytta på data till den tabell/tabeller som den nyligen skapade hos abonnenten. För illustration se Figur 2.



Figur 2. Snapshot-replikering

Fördelar med snapshot-replikering:

- Bra när ändringarna är omfattande.
- Bra när data är statiska och inte ändras så ofta.
- Bra när är det inte är så viktigt med omedelbara uppdateringar
- Bra när abonnentservern är av read-only typ.

Nackdelar med snapshot-replikering:

- Eftersom det kan hända att man skickar stora mängder av data i onödan, kan det leda till att man tar beslag på en stor del av bandbredden.
- Att skicka kopior av en databas kan medföra en säkerhetsrisk.

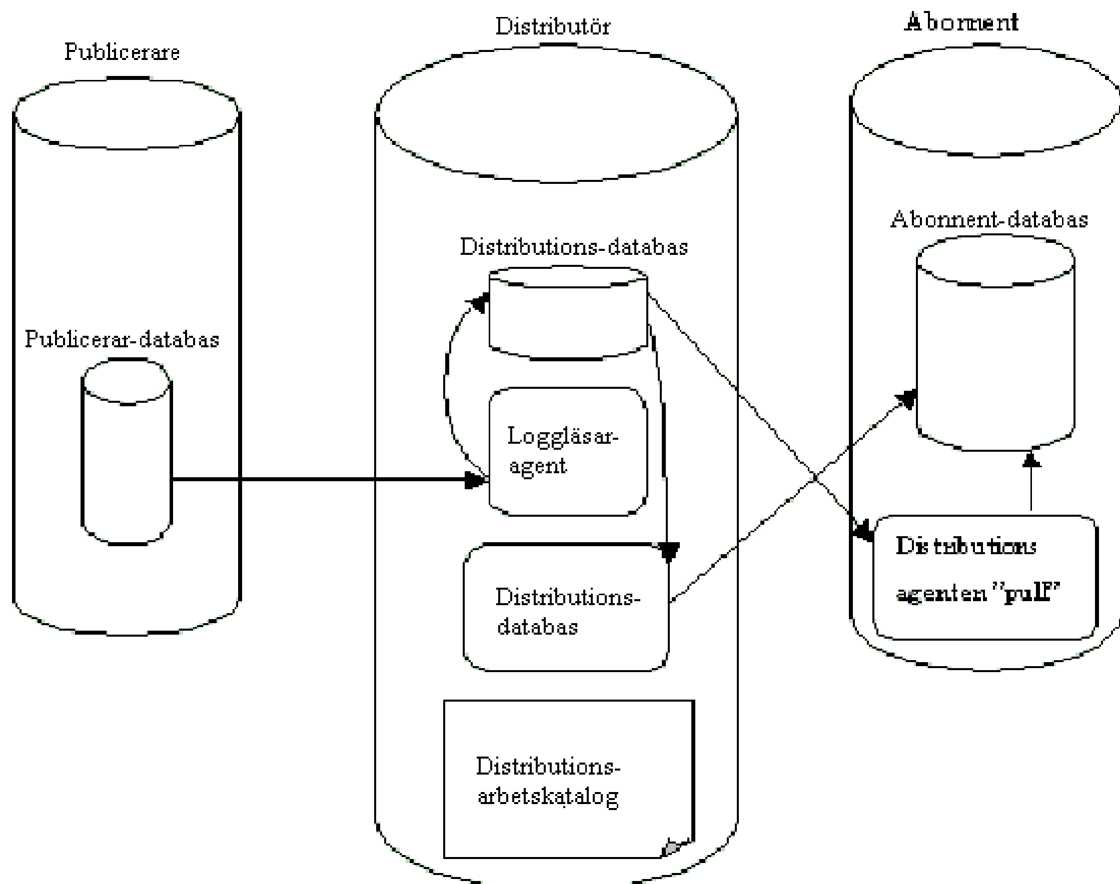
3.4.2 Transaktions-replikering

Transaktions-replikering innebär att så fort en ändring görs i databasen replikeras denna till mottagaren. Transaktions-replikering är oftast enkelriktad vilket gör att abonnent-databasen bör betraktas som skrivskyddad.

3.4.2.1 Agenter vid transaktions-replikering

Transaktions-replikeringsprocessen går till ungefär enligt nedan. Se Figur 3.

1. Snapshot-agenten läser den publicerade informationen och skapar sedan ett tabellschema. Snapshot-agenten finns inte med i figuren.
2. Distributions-agenten läser schemat och bygger därefter tabeller hos abonnenten.
3. Data förflyttas sedan med hjälp av distributions-agenten till de nyligen skapade tabellerna hos abonnenten
4. Om det finns några indexeringar som ska återskapas så görs detta i den nyss synkroniserade databasen.
5. Loggläsar-agenten börjar bevaka transaktionsloggen hos den publicerande databasen för nya transaktioner. Ifall en ny transaktion hittas flyttas den till distributionsdatabasen [1].



Figur 3 Transaktions-replikering

Fördelar med transaktions-replikering

- Goda förutsättningar för liten latens då bara ändringar i databasen skickas.
- Lite data att skicka om uppdatering sker ofta.
- Man skickar endast de ändringar som tillkommit sedan senaste synkroniseringen medan snapshot-replikering skickar hela databasen.

Nackdelar med transaktions-replikering

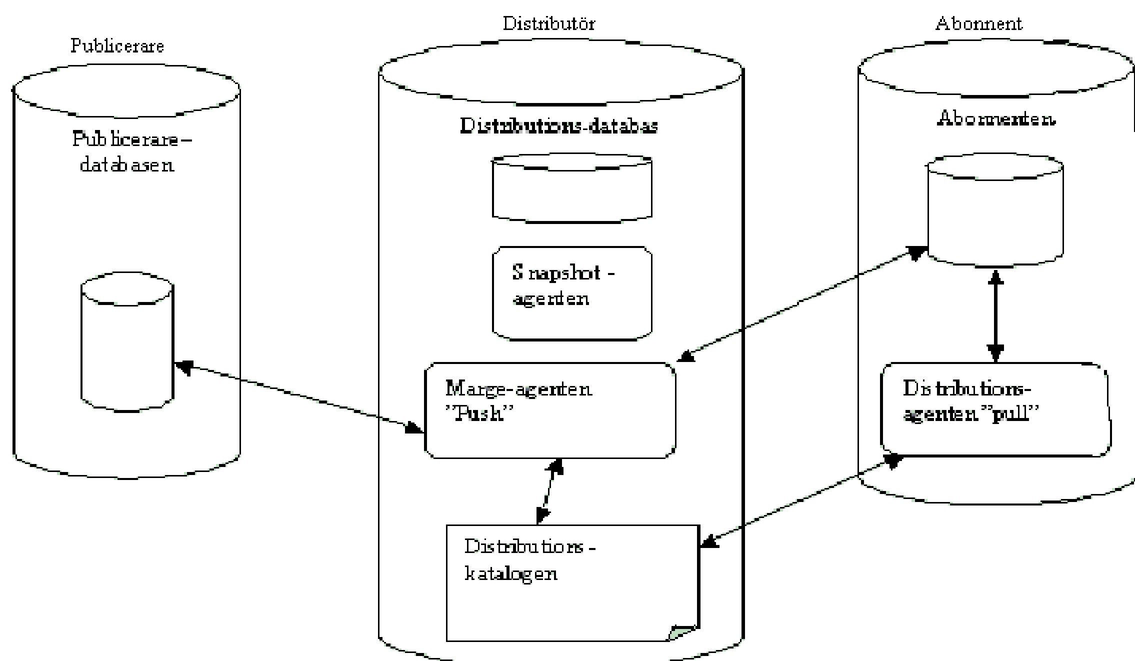
- Om man väljer transaktions-replikering så kan det betyda att man vill uppdatera sin databas ofta. Databasen har då låg autonomi. Detta kräver att man har möjlighet att kunna koppla upp sig relativt ofta. Annars blir databasen snabbt inaktuell.

3.4.3 Merge-replikering

Den tredje metoden för replikering är merge-replikering. Merge-replikering tillåter både publiceraren och abonnenten att göra ändringar i data. Merge-replikering ger god serverautonomi. Om ändringen av data sker när publicerare och abonnent inte är anslutna till varandra uppdateras data så fort det sker en anslutning.

3.4.3.1 Agenter vid merge-replikering

Merge-replikering tillåter flera servrar/databaser att jobba och ändra data oberoende av varandra och när de väl är färdiga med behandlingen av data så skickas alla uppdateringar till alla databaser. På så sätt får alla reda på alla ändringar, det vill säga alla databaser får förr eller senare samma innehåll. Metoden kallas för datakonvergering. Merge-replikering illustreras i Figur 4.



Figur 4. Merge-replikering.

Uppdatering kan ske i form av schemaläggning eller vid en begäran. Då det kan finnas flera abonnenter och publicerare som vill ändra och uppdatera data samtidigt kan det leda till konflikt. Detta har lösts med att man kan konfigurera publiceraren att sätta igång merge-

agenten som kan avgöra vilken data som kan accepteras och vilken som ska skickas vidare till andra sidan, (se [5], [6] och [7]).

Fördelar med merge-replikering

- Bra när flera abonnenter behöver uppdatera data vid olika tillfällen och skicka vidare uppdateringen till andra abonnenter och publicerare.

- Bra när abonnenten behöver ta emot data och göra ändringar då den inte är ansluten och sedan synkronisera ändringarna till publiceraren och andra abonnenter.

Nackdelar med merge-replikering

- Kan medföra svårlösliga konflikter när data från olika ställen skall läggas in i tabell.

3.5 Hur man väljer distributionsmetod

Hur skall man välja replikeringstyp (eller distributionmetod)? Saker man kan tänka på är bl.a. följande:

- Hur mycket data och vilken data som replikerna behöver. Olika repliker kanske behöver olika data beroende till exempel på om de används i olika geografiska områden.
- Man kanske bara behöver synkronisera delmängder av databasen.
- Skall abonnenterna kunna skicka data?
- Hur ofta skall man kunna synkronisera? Vissa data behöver kanske aldrig uppdateras eller åtminstone inte så ofta, till exempel kundnummer eller telefonnummer. Hur beroende är repliken av att vara 100% uppdaterad hela tiden? Här pratar man om autonomi och latens. Nedan kommer vi att ta upp dels när man kan använda replikering och även vilken typ av replikering som passar bäst i olika situationer.

3.5.1 När skall man använda replikering?

- Geografisk placering

Om man behöver använda en databas men inte har möjlighet att använda "originaldatabasen" så kan man använda sig av replikering. Personer som reser kan behöva en replik av en databas om de inte kan nå originaldatabasen från där de befinner sig. När personerna har möjlighet att komma åt "orginaldatabasen" så kan de synkronisera data.

- Lastbalansering

Man kan fördela belastningen så att istället för att ha en server med en databas så kan man replikera databasen och ha repliker placerade på olika servrar. Därefter kan man fördela anropen till databaserna.

- Distribuera mjukvara.

Förändringar i "originaldatabasen" vad det gäller annat än ren data till exempel tabellförändringar eller förändringar i lagrade procedurer kan föras över vid synkronosering. Man behöver då inte göra förändringar av replikerna manuellt utan det sker automatiskt.

Fördelar med replikering

- Tillgängligheten bör öka om man kan lastbalansera
- Snabbt eftersom frågor kan ställas lokalt om det finns en lokal databas.

Nackdelar med replikering

- Synkronisering kan ta tid då alla databaser måste uppdateras

3.5.2 Vilken typ av replikering använde vi?

Vi hade att göra med relativt stora databaser med relativt sett få uppdateringar per dygn. Kommunikationen mellan databaserna kunde anses vara säker i den meningen att det inte var ofta som kommunikationen var bruten mellan servrarna. Vi kunde dessutom anse att databasen som så att säga samlade ihop data från de olika fabrikerna var skrivskyddad. Den typ av replikering vi valde var därför transaktionsreplikering.

4 Filöverföring med FTP respektive HTTP

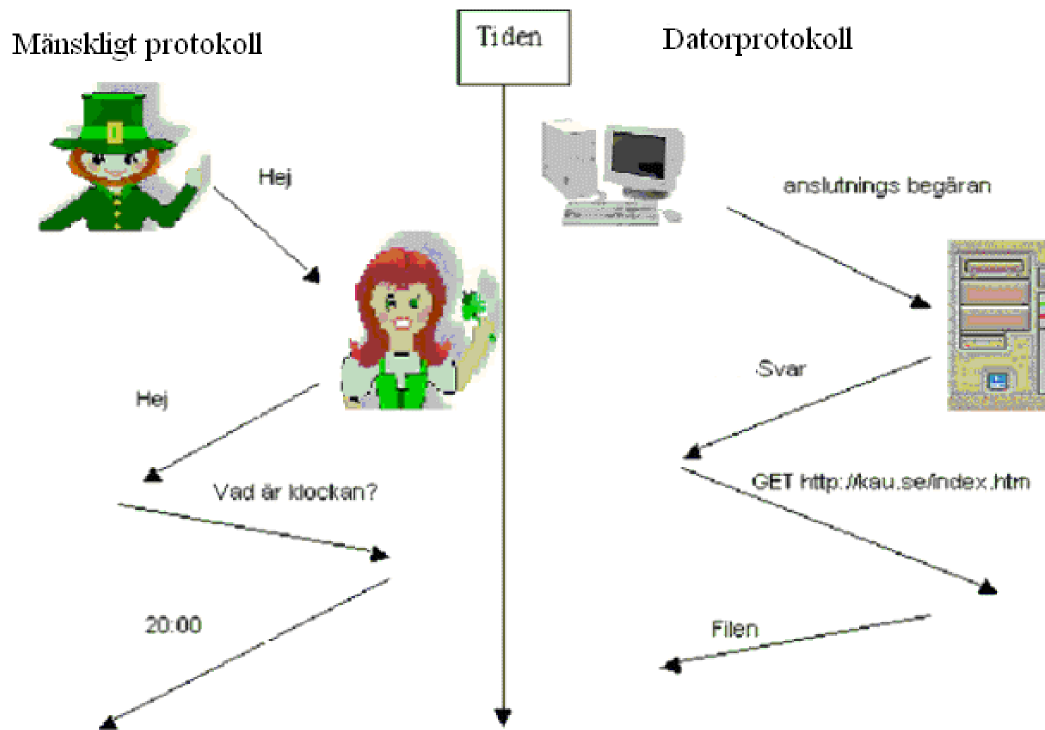
När man använder SQL-servers inbyggda replikering så kan ändringar föras över mellan olika servrar via den port som SQL-server använder, nämligen 1443. Eftersom vi ville föra över data utan att behöva ändra konfigurationen i användarens brandvägg så var alternativen att föra över data styrda av de portar som är öppna. I vårt fall är vi hänvisade till portarna för FTP (portnummer 20 och 21) samt HTTP och HTTPS (portnummer 80 och 443).

I detta avsnitt kommer vi därför att diskutera HTTP och FTP. Vi kommer att diskutera hur dessa protokoll fungerar, vad som utmärker dem samt fördelar och nackdelar med dessa protokoll.

Som inledning till kapitlet kommer vi att beskriva vad ett protokoll egentligen är. Vi kommer även att visa ett exempel på ett mänskligt protokoll och ett exempel på ett datorprotokoll. Kapitlet avslutas med en jämförelse mellan FTP och HTTP samt vilket protokoll som vi valde till filöverföringen.

4.1 Vad är ett protokoll?

Även om vi människor inte tänker på det så använder vi ett eller flera protokoll dagligen. Ett bra exempel är när vi träffar en annan person. Vi brukar nästan alltid börja med ett hej och som svar får man ett hej eller någon annan artighetsfras. Se Figur 5.



Figur 5. Exempel på två olika protokoll

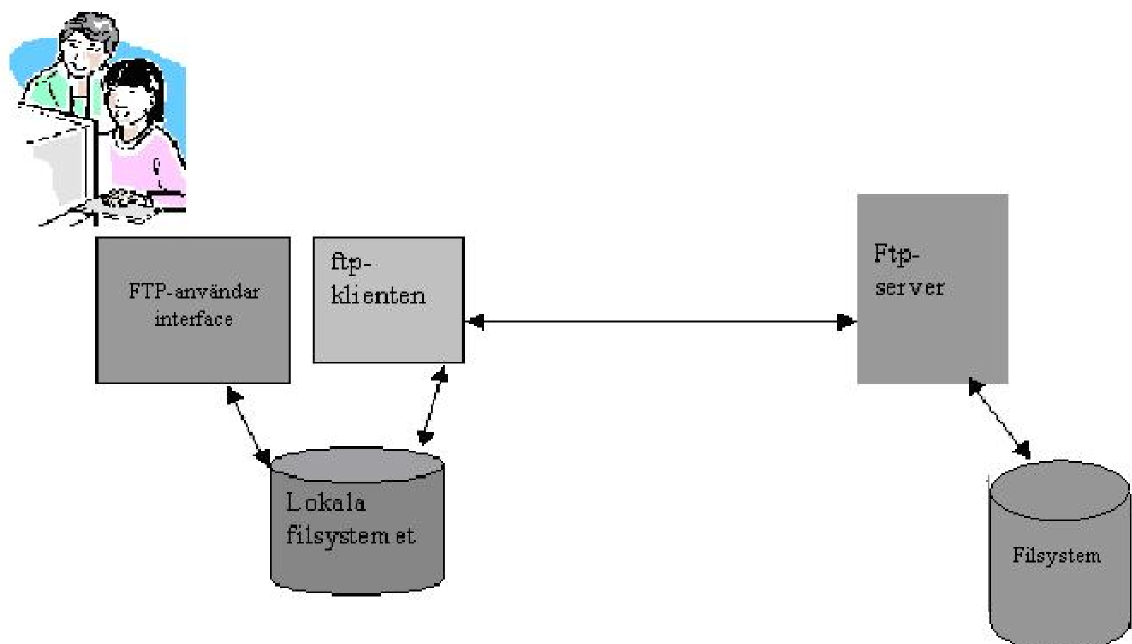
Som vi ser av Figur 5 är det egentligen ingen större skillnad mellan ett datorprotokoll och ett mänskligt protokoll. Både människan och datorn startar med en inledningsfras innan informationen utbytes. En skillnad är dock att medan människan kanske säger hej innan de börjar utbyta information brukar datorer begära att få skapa en anslutning och sedan be att få skicka eller ta emot data.

4.2 FTP (File Transfer Protocol)

FTP är ett protokoll för överföring av data från en dator till en annan dator, där ena sidan kallas server och den andra för klient. Protokollet utvecklades 1971 och beskrivs i RFC 959. Även om FTP är ett ganska gammalt protokoll så är det fortfarande väldigt populärt i dagens Internet. Till exempel många webbservrar använder FTP för att ladda upp eller ladda ner filer. FTP används också flitigt av Internetanvändare som vill dela med sig av sina filer till andra Internetanvändare.

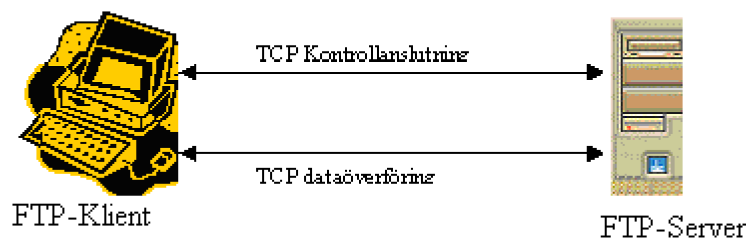
Som exempel anta att Kalle som går mediaprogrammet har gjort en film som examensarbete. Då Kalle äger upphovsrätten till filmen vill han skicka kopior till sina gamla kompisar som bor i Malmö, Umeå och Göteborg. För att kunna göra det kan Kalle lätt sätta upp en FTP-server på sin dator hemma och tilldela varje kompis ett användarnamn och ett lösenord samt ge dem de behörigheter som han vill.

Man måste först ange adressen till FTP-servern för att kunna ansluta. När anslutningen har skapats begär FTP-servern att få användarnamn och lösenord och efter att användaren angett detta tillåts denne att komma åt filerna i servern eller att ladda upp en eller flera filer till servern. Se Figur 6.



Figur 6. Bild över FTP-klient och FTP-server.

FTP är ett applikationslager-protokoll som körs över TCP. En väsentlig skillnad mellan FTP och många andra dataöverföringsprotokoll är att FTP använder sig av två TCP anslutningar, en anslutning för kontroll och en för dataöverföring. Se Figur 7.



Figur 7. Bild över TCP-anslutningarna för FTP- protokollet.

Fördelar med FTP

- Det är relativt lätt för en programmerare att programmera en FTP-klient eller server då det är ett väl standardiserat protokoll.
- Även en icke-expert kan installera och konfigurera en FTP-server.
- För att kunna komma åt data eller ladda upp data måste man kunna identifiera sig vilket gör att man kan skydda filerna från att obehöriga kommer åt dem.
- Under anslutningen behåller FTP anslutningsstatusen. Fördelen med det är att man kan skicka så många filer man önskar sig över en och samma anslutning.

Nackdelar med FTP

- FTP skickar som standard både användarnamn och lösenordet i klartext och då kan vem som helst kopiera det på vägen och utnyttja det vid tillfälle.
- Om man befinner sig bakom en brandvägg kan det leda till en säkerhetsrisk då man måste öppna extraportar för FTP. Med detta menar vi att en obehörig person kan hacka sig in i systemet genom att utnyttja de här öppnade portarna som normalt kanske skulle ha varit stängda [3].

4.3 HTTP (Hyper Text Transfer Protocol)

I detta avsnitt beskriver vi hur HTTP fungerar. Vi tar också upp hur olika versioner av HTTP kan påverka överföringskapacitet och slutligen diskuterar vi säkerheten något alltså i korta drag vad som är bra och dåligt med att föra över data som replikerats från publiceringsservern till abonnentservern med HTTP

4.3.1 Hur HTTP fungerar

HTTP skickar data via TCP. För att skicka en fil så går HTTP igenom följande steg.

1. Klienten säger till TCP att skapa en anslutning
2. Klienten skickar en REQUEST (se nedan)
3. Servern skickar en RESPONSE och säger till TCP att koppla ner anslutningen
4. Klienten tar emot RESPONSEN
5. TCP kopplar ner anslutningen

För att TCP ska kunna skapa en anslutning så måste man ange en IP-adress och ett portnummer. TCP kommer då att försöka ansluta sig till den datorn med den angivna IP-adressen och portnumret. Därefter skickar HTTP en REQUEST. En REQUEST är ett paket som innehåller information om vad klienten vill att servern skall göra och eventuellt skicka tillbaka. En REQUEST kan se ut så som följer:

GET /adir/afile.ext HTTP/1.1	Requestline
Host: www.aweb.net	Header 1
Connection: *Close*	Header 2
User-Agent: *Mozilla/4.0*	Header 3
Accept-language: *fr*	Header 4
(en tom rad)	

1. Den första raden är en REQUESTLINE. En REQUEST har tre fält; metodfältet, URL-fältet och HTTP-versionfältet. Metodfältet kan innehålla flera olika värden till exempel GET, POST eller HEAD. Här anges värdet *GET.* URL-fältet talar om vilken fil som skall hämtas (*/adir/afile.ext*) och HTTP-versionen specificerar vilken HTTP-version som används(*HTTP/1.1*).

2. Connection anger om servern skall hålla TCP-anslutningen öppen när filen är skickad.
3. User-Agent anger vad det är för webbläsare som skall visa filen.
4. Accept-language säger att om det finns en fransk version så skall den skickas.

Servern tar emot begäran, ser efter vad klienten vill att den skall utföra och gör det. I detta fall skall servern skicka en fil. Servern skickar filen i ett RESPONSE-paket. Det kan se ut så här.

HTTP/1.1 200 OK
Connection:*close*
Date: *Thu, 06 Feb 2003 16:03:23 GMT*
Server: *Apache/1.3.0 (Unix)*
Last-Modified: *Fri, 23 Dec 2002 08:03:04 GMT*
Content-Length: *7423*
Content-Type: *text/html*

(data (entity body)…)

Meddelandet innehåller en statusrad och sex headers samt en del som innehåller själva filen (entity body). Meddelandet är ganska självförklarande.

4.3.2 Olika versioner av HTTP och överföringskapacitet

HTTP finns i flera versioner och den senaste är version 1.1. En skillnad mellan 1.0 och 1.1 är att i 1.1 så kan man välja att inte stänga TCP-anslutningen mellan överföring av två filer i det fall man vill hämta flera filer. Man spar då in tiden det tar att skapa en TCP-anslutning samt slipper den extra trafik som går över nätet. En annan fördel med så kallade ”ständiga” TCP-anslutningar (alltså anslutningar som inte kopplas ner mellan de olika filöverföringarna) är att när TCP börjar skicka data så inleder den i ett långsamt tempo för att sedan öka hastigheten efterhand som den märker att nätverket klarar av det. Om man då kopplar ner TCP-anslutningen mellan varje filöverföring så kommer TCP att börja om från låg hastighet vid varje ny filsändning. Detta medför att genomsnittshastigheten för filöverföringen kommer att bli låg. Om man istället använder ständiga TCP-anslutningar så kommer den inledande sändningen av data att vara långsam men påföljande filer kommer att skickas snabbare

eftersom TCP-anslutningen då kommit upp i varv. Detta kommer antagligen att ha störst effekt om man skall skicka flera små filer. Vid stora filer kommer det nog att ha mindre effekt.

Dessutom finns det två versioner av ”ständiga” TCP-anslutningar. Dessa är dels med pipelining och dels utan pipelining. Utan pipelining så kommer klienten att skicka en begäran och sedan vänta på svaret. Därefter kommer den att skicka en ny begäran. Med pipelining så skickar klienten alla innestående begäran den har på en gång och väntar därefter in svaren för dessa. På detta sätt sparas tid och trafik.

4.3.3 Säkerhet

Då .NET miljön (se kapitel 5) använder Internet i stor utsträckning så är mycket jobb lagt på att kunna implementera överföring av data med HTTP på ett säkert sätt. Detta görs dels genom kryptering av det som skickas, dels genom autentisering och auktorisering. Det finns stöd för kryptering i .NET men vi har inte studerat det. Däremot har vi tittat lite på autentisering och auktorisering. Det är möjligt att i .NET-miljön autentisera användare. När man förvissat sig om att användaren är den hon utger sig att vara så auktoriserar man användaren. Det vill säga man kontrollerar att användaren har de rättigheter som krävs för de resurser som användaren begär.

Autentisering och auktorisering sker i flera nivåer. Beroende på hur hög säkerhet man vill ha så är Internet Information Services (IIS), Windows och ASP.NET inblandade på lite olika sätt. IIS har hand om auktorisering med hjälp av IP-adress och domän. IIS kan tillåta respektive avstyra anrop till servern beroende på vilken IP-adress eller domän som anropet kommer ifrån. Därefter sker en autentisering av användaren. Detta kan ske på fyra olika nivåer; Anonym access-, Basic-, Digest- och Windows-autentisering. Säkerheten här är stigande från Anonym access- som är lägst till Windows-autentisering som är högst. När användaren är autentiserad knyts hon till ett Windowskonto. Vid Anonym Access är det ett konto som heter IUSR_*maskinnamn*. Med utgångspunkt från detta konto kan Windows auktorisera användaren som får tillgång till de resurser som kontot har rättigheter till. När detta är avklarat kollar IIS om den kan ta hand om den inkommande förfrågningen. I vårt fall med filöverföring så är det ASP.NET (aspnet_isapi.dll) som tar hand om förfrågan. ASP.NET skickar förfrågan vidare till en ASP.NET-process(aspnet.wp.exe). Denna process kommer att

jobba under en identitet som bestäms beroende av om ASP.NET har personifiering påslagen eller inte. Om den är avslagen så anges processens identitet av den som anges i konfigureringsfilen och om personifiering är på så anges processens identitet av den identitet som användaren som loggade in har.

4.3.4 Fördelar med HTTP

- Om man använder HTTP för filöverföring så har man stora möjligheter att utföra överföringen på ett säkert sätt.
- HTTP1.1 kan eventuellt vara snabbare jämfört med tidigare versioner av protokollet om man skall föra över många små filer.

4.3.5 Nackdelar med HTTP

- Om man läser artiklar om HTTP så verkar det som om HTTP är utvecklat för överföring av små filer. En del personer skriver även att de har haft problem med filöverföring via HTTP i samband med stora filer. Vi hittade inget direkt stöd för att det skulle vara så och ser inga hinder att använda HTTP även för stora filöverföringar.
- En skillnad mot FTP kan vara att FTP stödjer upptagning av en förlorad förbindelse och att man börjar föra över en fil där man blev avbruten.
- En nackdel kan vara att man inte har samma kontroll i HTTP som i FTP. I FTP kvitteras överföringar och i HTTP sker bara en REQUEST/RESPONSE sekvens. Å andra sidan kan man implementera en kvittering för HTTP (se [2], [3] och [4]).

4.4 FTP kontra HTTP

I detta kapitel har vi diskuterat både FTP och HTTP samt dessas för- och nackdelar. Både FTP och HTTP är protokoll för filöverföring över Internet, men det finns väsentliga skillnader mellan dem. Medan FTP använder två portar, en för dataöverföring och en för anslutningsstatusen så använder HTTP samma port för både anslutningen och dataöverföringen. Medan FTP lämpar sig bättre än HTTP när man ska överföra stora filer så kräver FTP längre anslutningstid än HTTP. Under anslutningstiden behåller FTP

anslutningsstatusen medan HTTP inte gör det. De senaste versionerna av HTTP erbjuder effektivare uppkoppling och nedkopplingsmetoder i jämförelse med dessa kontrapartner FTP.

Då detta arbete till största delen har varit av utredande karaktär har vi lagt mycket tid på att undersöka befintlig teknik. För filöverföring valde vi FTP framför HTTP. Det viktigaste skälet till att vi valde FTP var att vi tyckte att det var lättare att föra över data med FTP när man har många filer. Vi trodde också att det var lättare att utföra vissa operationer med FTP jämfört med HTTP. Exempel på en sådan operation är när man ska radera en fil på server-sidan. Dessutom tyckte vi att det krävdes mindre tid för att implementera en FTP-klient än en HTTP-klient.

Vi tycker att det är viktigt att påpeka att HTTP har en hel del fördelar i jämförelse med FTP. Nu i efterhand om vi hade fått chansen att göra om applikationen och hade lite mer tid på oss så hade vi nog valt HTTP 1.1 eller HTTPS istället för FTP av följande anledningar:

1. HTTPS krypterar all data som skickas och FTP gör det inte.
2. Det går utmärkt att föra över andra filer än replikerings-filerna med HTTP / HTTPS det vill säga att det även skulle gå bra att skicka grafiska och multimedia-filer med HTTP.
3. Möjligheten att implementera autentisering i HTTP finns.
4. HTTP 1.1 stödjer *persistent connection* med *pipelining* vilket gör att man kan skicka så många filer man önskar sig över en och samma anslutning.

5 Verktyg och utvecklingsmiljö

I vår lösning har vi använt ett antal verktyg, vilka vi ger en kort beskrivning av i detta kapitel. Vi börjar med att beskriva den plattform som vi har använt oss av i vår utveckling av programmet och sedan ges en introduktion till det programmeringsgränssitt som vi har använt. Kapitlet avslutas med en beskrivning till vad en Windows-tjänst är och hur man installerar och avinstallerar en sådan.

5.1 Microsoft Visual Studio .NET

Som plattform och kompilator har vi valt att använda Microsoft Visual Studio .NET. Microsoft Visual Studio .NET (VS NET) är Microsofts senaste utvecklingsmiljö för utveckling av applikationer i Windows-miljö. I och med Visual studio .NET har Microsoft givit oss programmerare nya möjligheter. Den största nyheten i VS .NET är .NET Framework vilket är ett klassbibliotek. Tanken med .NET Framework är att använda konceptet med arv. Alla objekt i .NET Framework utformar en hierarki med en enkel rot, System.Object klassen (det vill säga att alla andra klasser ärver från denna klass).

En annan nyhet är att alla programspråk i VS .NET är likvärdiga, vilket innebär att man skulle kunna skriva ett program i det programspråk som man önskar sig utan att behöva tänka på språkets möjligheter och begränsningar. Det är arv-tekniken i .NET Framework som möjliggör detta. Till exempel, den del av .NET Framework som kallas Windows-form erbjuder en klass för att rita menyer, fönster, och kontroller. Alla .NET språken kan använda denna klass och därför har de samma möjlighet att skapa användargränssnitt.

Microsoft tillhandhåller några välkända språk med .NET, nämligen Visual Basic .NET, C++, C# (uttalas C sharp) samt Jscript. Vi har valt Visual Basic som programmeringsspråk.

5.1.1 WinInet API

Microsoft tillhandahåller många programmeringsgränssnitt för att programmera både server- och klient-applikationer. En av dessa är WinInet. WinInet stödjer de vanligaste Internet-protokollen såsom HTTP och FTP.

Liksom med sina tidigare komponenter har Microsoft försökt att göra användningen av WinInet så enkel som möjligt. Med WinInet kan man skriva både klient- och server

applikationer utan att behöva lära sig WinSocket-programmering. Att skriva en FTP-applikation är i princip det samma som att skriva en HTTP-applikation med reservation för att man behöver byta namnet på några metoder och variabler.

5.1.2 Windows-tjänster

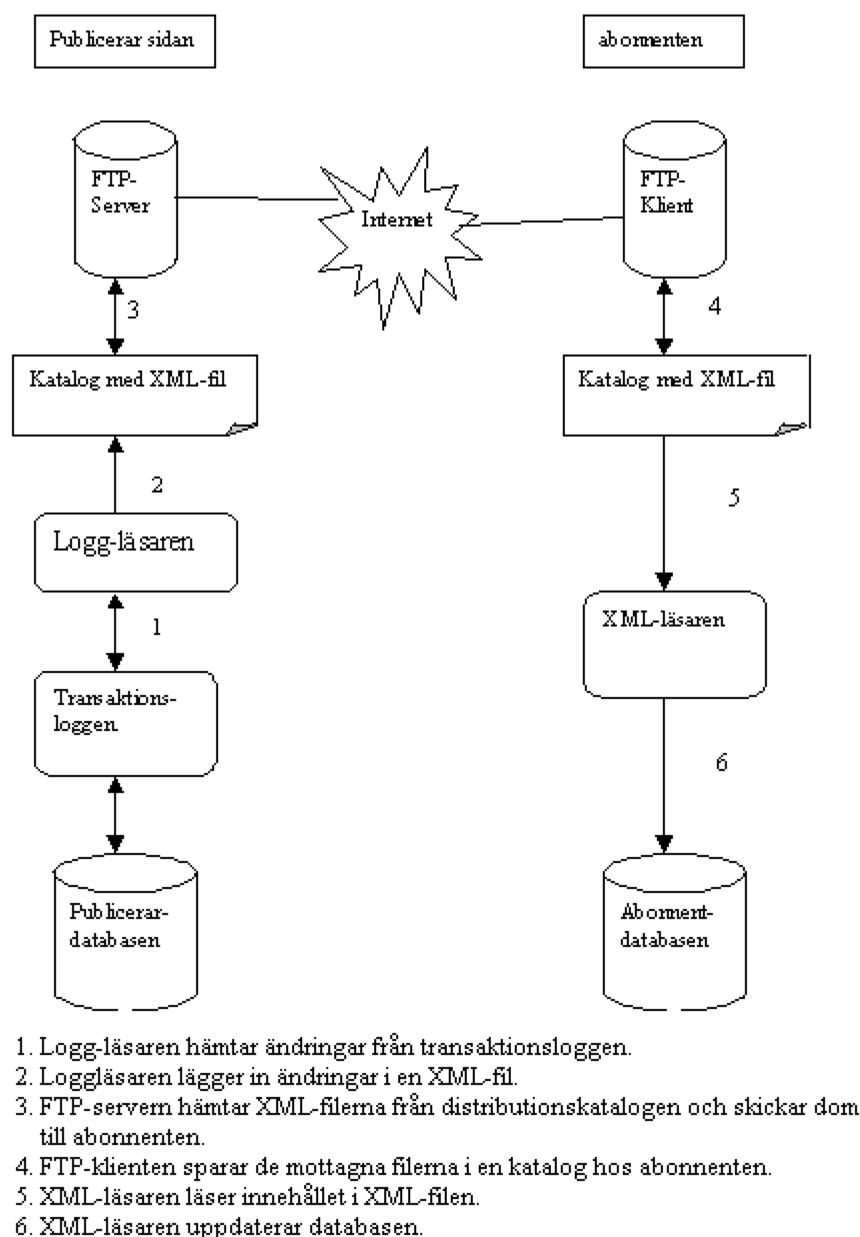
En Windows-tjänst är en applikation som saknar gränssnitt. Den körs i bakgrunden och är inte tillgänglig i aktivitetsfältet. För konfiguration av en Windows-tjänst ska man gå till Kontrollpanelen → Administrativa verktyg → Tjänster (För Windows 2K). När tjänsten ska köras igång är det systemadministratörens uppgift att bestämma när programmet skall startas, vilket görs i tjänstens inställningar. Man kan välja att starta tjänsten automatiskt när operativ-systemet startar eller genom att manuellt markera tjänsten och sedan klicka på start i menyn.

Att installera en Windows-tjänst är lite annorlunda jämfört med att installera en vanlig Windows-applikation. För att installera tjänsten måste man först ha ett program som installerar det åt sig. I vårt fall har vi använt *InstallUtil*[*] som är en .NET-applikation för att installera Windows-tjänster.

[*] InstallUtil finns under *C:\WINNT\Microsoft.NET\Framework\v1.0.3705*. Anta till exempel att vår FTP-klient finns i *C:\exjobb\ftp_klient* – för att installera den bör man skriva följande: *InstallUtil ”C:\exjobb\ftp_klient\tjänstens-namnt.exe”* i kommandofönstret.
För att avinstallera tjänsten skriver man: *InstallUtil /U ”C:\exjobb\ftp_klient\tjänstens-namnt.exe”*.

6 Beskrivning av konstruktionslösningen

Vår lösning består av tre självständiga applikationer. Alla tre delarna installeras och körs separat och oberoende av varandra, det vill säga om en av applikationerna går ner så fungerar de andra två i alla fall. Dessa tre självständiga applikationer är logg-läsare, FTP-klient och XML-läsare. Applikationernas samverkan illustreras i Figur 8. I de följande avsnitten beskrivs hur var och en av dessa applikationer fungerar.



Figur 8. Applikationernas samverkan

6.1 Loggläsaren

När man använder transaktions-replikering så läser man in ändringar som är gjorda i den publicerande databasen och överför dessa ändringar till de databaser som är abonnenter. Dessa ändringar läses från transaktionsloggen och i fallet med replikering av en SQL Server-databas så är det loggläsar-agenten som sköter det. Därefter lagras de nya transaktionerna så att distributions-agenten kan skicka iväg dem. Här kommer vi att beskriva den del av lösningen som gör detta, det vill säga läser i transaktionsloggen, extraherar de data (nya transaktioner) den behöver och lagrar dessa för senare överföring.

6.1.1 Läsa transaktioner från transaktionsloggen.

Det första problemet var alltså hur man skulle läsa in ändringar från transaktionsloggen. Loggen lagras i en fil som heter C:\Program Files\Microsoft SQL Server\MSSQL\Data\ test_Log.LDF om man har en databas som heter test. I SQL Server-versioner tidigare än 7.0 så kunde man läsa transaktionsloggen på samma sätt som en vanlig databas. Det går inte nu, utan man får använda ett kommando som heter dbcc. Dbcc betyder Database Consistency Checker och användes för administrativa uppgifter. För att få fram ändringar så skriver man dbcc log (*databas, nr*). Här anger *databas* den databas man vill se uppgifter från och *nr* är ett nummer från –1 till 4 som anger hur mycket data man vill ha ut. Om man anger 0 får man minst och anger man 4 så får man mest data. –1 betyder att man får väldigt mycket data. De data vi använder är *Lsn* (Log Sequense Number), *Operation* och en hexadecimal sträng. *Lsn* är ett sekvensnummer som räknas upp för varje transaktion. *Lsn* kan man använda för att de ändringar som utförts i publiceringsdatabasen utförs i rätt ordning i abonnentdatabasen. *Operation* är den typ av operation som utförts. Vi är intresserade av tre sådana, nämligen Insert, Modify och Delete. Den sista, den hexadecimala strängen, innehåller de ändringar som gjorts i databasen.
Hur skall denna då tolkas? Vi hittade ingen dokumentation om det utan vi fick anta. Det visade sig att det fanns tre alternativ: att lägga till en post (Insert), ta bort en post (Delete) (som ovan) eller göra ändringar (Modify) i en post. Det verkar dessutom finnas olikheter beroende på vad databasen innehåller för typer på posternas data. Vi gick inte så djupt in på att kolla olika data utan nöjde oss att tolka data av typen *Integer*. För att illustrera detta följer nedan ett exempel på en hexsträng. Exemplet visar hur hexsträngen ser ut om man tar bort en post i databasen. Den ändring som gjorts i databasen var att ta bort posten med id nummer 22.
Nedan har vi ställt upp två tabeller och gjort en genomgång av hexsträngen. Den första tabellen visar dels de Integer värden som var med i databasen, dels motsvarande

hexadecimala värden för att kunna tolka strängen. Därefter kommer en genomgång av hexsträngen uppdelad i grupper om fyra bytes. Observera att ordningen på byten är omvänd, det vill säga om ett Integervärde i databasen är 976 så blir motsvarande hexadecimala värde 3D0. Detta står som D0030000 i hexsträngen. Byten står i Little Endian det vill säga att byten med minsta värde står först. Här nedanför beskriver vi hur man skall tolka hexsträngen. För utförligare dokumentation se bilaga A.4.

Rubrik	Bytes	Kommentar
Längd	4	De två första byten verkar inte användas.
LSN	12	
Transaction ID	4	
Flagbits	4	
Okänt	36	Verkar alltid ha samma värden. Okänd funktion
Post	4	
Kolumnvärden	4	X antal kolumner (vid heltal)
Trailer	4	

De hexadecimala strängarna för Modify och Insert ser ut på ett liknande sätt. Dessa varianter kan studeras i bilagorna A.1 och A.2.

Det vi gör med hexsträngen är att ta ut värdena för posten som ändrats. Det betyder att nu har vi tagit reda på de mest nödvändiga uppgifterna vi behöver för att kunna utföra en uppdatering av abonnentdatabasen. Det som återstår att göra är att spara uppgifterna eftersom det inte är loggläsarens uppgift att föra över uppgifterna till abonnenten.

6.1.2 Lagring av transaktionsdata

Vi valde att spara uppgifterna i XML-filer. Vi har inte satt oss in i XML-strukturen på något djupare plan med schema och liknande utan bara skrivit in våra uppgifter från databasen på ett strukturerat sätt. Då en XML-fil bara är ett textdokument så är detta inget problem och eftersom VS har ett utbyggt stöd för att hantera XML så har vi använt de inbyggda funktioner som VS erbjuder. Strukturen på XML-dokumentet speglar uppbyggnaden av de data som

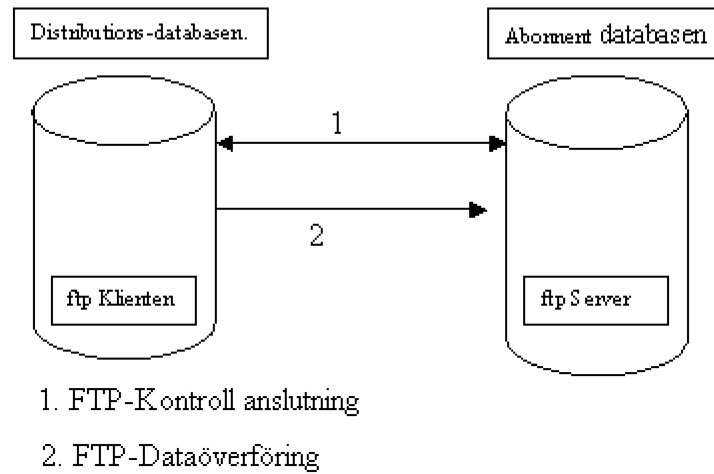
skall skickas på ett som vi tycker lättläst sätt. En fördel med XML-filer var att de kunde vara ganska lätta att läsa och tolka. Namnet på XML-filen är sammansatt av ”xmlReplicator_”, datum och klockslag när filen skapades (2003-04-20_13_37_40) samt ”.xml”. Tanken var att vi skulle kunna sortera filerna med hjälp av namnen. Senare ändrade vi oss och sorterar nu med hjälp av tidpunkten som filen skapades. Vi behöll ändå namnet då det är bra med namn som säger något om filen om man skall leta buggar. Strukturen på XML-filen kan studeras i bilaga A.3. Där kan man se att vi skickar med uppgifter om vad varje kolumn har för typ. Vi antog att man behövde den informationen när man skulle uppdatera databasen. Men eftersom man använder text och inte anger typ när man konstruerar SQL-uttryck så är det kanske inte nödvändigt att skicka med information om typen. Eventuellt kunde man använda informationen om typen för att testa att värdet på posten var godkänt. Därför lät vi det vara kvar.

Ett problem var att det kunde bli väldigt mycket data som skulle skickas om vi sparade dem som XML-filer. Det verkade som om XML-filer innehöll mycket ”luft”. Vi komprimerade en XML-fil med winzip och då minskade storleken från 20 kb till 1 kb; alltså till en tjugondel. Vi har inte undersökt om det finns möjligheter att automatiskt komprimera filerna.

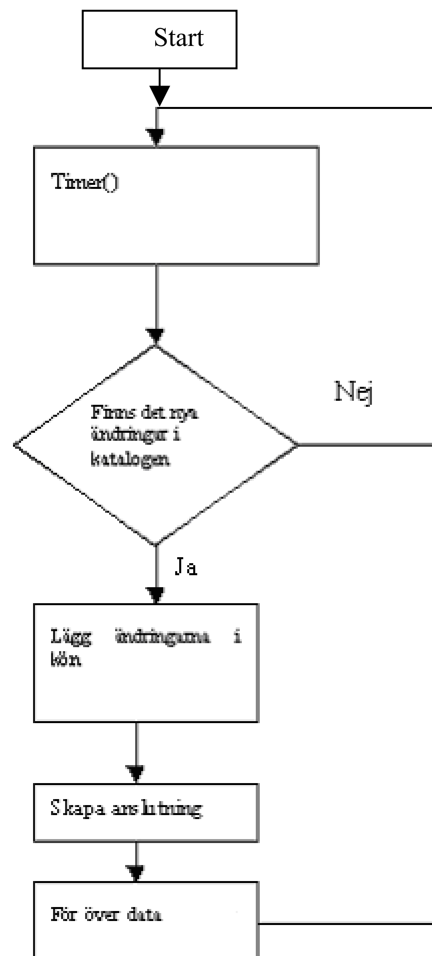
6.2 Överföring av data

Efter att ha bestämt oss för att överföra filerna via FTP gjorde vi en första design och ritade ett flödesschema som vi hade tänkt använda oss av vid implementeringen.

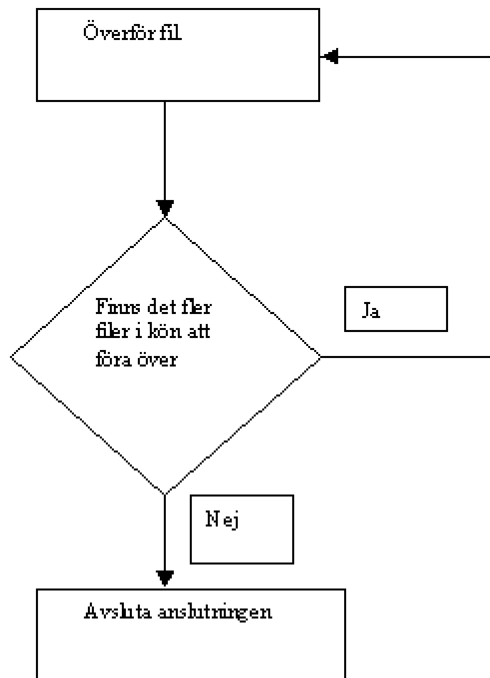
I denna första design tänkte vi använda FTP:s putmetod. Vår FTP-klient skulle installeras på distributionsdatabasen och så fort det skedde en ändring i distributionsdatabasen skulle det skapas en replikering som sedan skulle skickas till abonnenten. Designen illustreras i Figur 9, Figur 10 och Figur 11.



Figur 9. Bild över hur data skulle skickas i vår första design.



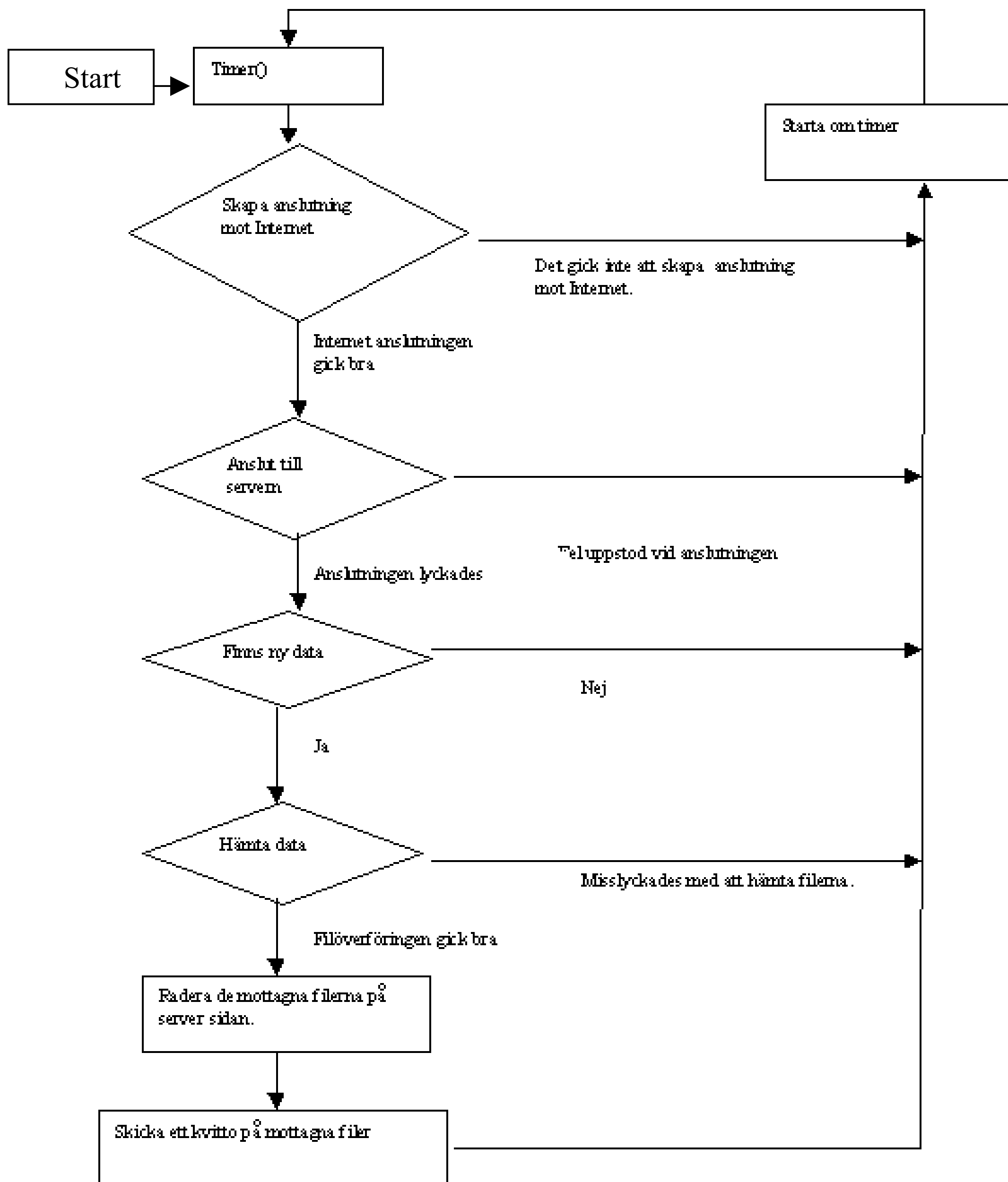
Figur 10. Flödesschema för den första designen av FTP-klienten.



Figur 11. Filöverföringen för den första designen.

När vi gjort den första designen hörde vi med vår uppdragsgivare och fick hans synpunkter på den och dessutom talade vi om våra funderingar om att använda oss av FTP för filöverföringen. Uppdragsgivaren tyckte att det var OK att vi använde FTP, men hade några synpunkter på designen.

Uppdragsgivaren tyckte för det första att vi skulle använda FTP:s Getmetod istället för Put som vi hade tänkt oss från början; det vill säga att abonnentdatabasen begär att få ändringar när denna vill. Uppdragsgivaren ville även att ett kvitto på de mottagna filerna skulle skickas tillbaka till servern. Dessutom var det fortfarande viktigt att programmet fungerade som en Windows-tjänst vilken skulle köras i bakgrunden. Efter detta möte med uppdragsgivaren gjorde vi de ändringar i vår design som vi fann nödvändiga för att uppfylla uppdragsgivarens önskan. Flödesscheman för den modifierade FTP-klienten illustreras i Figur 12.



Figur 12. Flödesschema för den modifierade FTP-klienten.

En grundtanke med designen är att programmet skall vara tidstyrt; en timer aktiveras vid programstart, och efter en förprogrammerad tidpunkt skall den utlösas och aktivera FTP-klienten. När den väl har aktiverats skall den stoppa timern tillfälligt och sedan skapa en anslutning mot Internet. Om anslutningen går bra skall programmet gå vidare och logga in på servern, vid fel skall timern startas om.

Ifall det sker ett fel vid anslutning, till exempel fel användarnamn, fel lösenord eller att servern inte svarar alls så skall timern startas om och ett felmeddelande skrivas i logg-filen. Går allt smärtfritt skall filhämtningen påbörjas. Om filhämtningen lyckas skall ett kvitto

skickas tillbaka till servern och timern startas om. Vid fel på filöverföringen skall timern aktiveras om och felet skall skrivas i logg-filen.

6.2.1 Implementationen av FTP-klienten

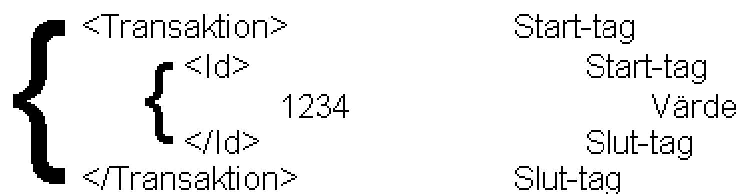
Efter att ha undersökt de olika alternativen som fanns kom vi fram till att vi hade två valmöjligheter för programskrivningen. Det ena valmöjligheten var att börja koda på WinSocket-nivå vilket skulle medföra att vi var tvungna att ta hand om data på transport lagernivå. Den andra valmöjligheten var att använda Microsofts redan inbyggda komponenter (MFC – Microsoft Foundation Class) i vårt fall WinInet API. Vi valde att använda WinInet vid vår implementering. Eftersom vi anser att objektorienterad (OO) programmering är en bra programmeringsmetod har vi försökt implementera FTP-klienten OO. FTP-klienten är en Windows tjänst. Den har installerats och testats i Windows 2K Advanced Server.

6.3 Överföring av data från XML-filer till databasen

När XML-filerna som loggläsaren sparade på publiceringsservern hade förts över till abonnentservern var det dags att läsa in data från XML-filerna och uppdatera databasen.

6.3.1 Läsning av XML-filerna

Då vi inte hade gjort något schema för vår XML-struktur så var läsningen av filen relativt enkel. Ett XML-dokument kan bestå först av diverse metadata och därefter av ”riktig” data. Strukturen bygger på element. Ett element består av en start-tag och en slut-tag. Däremellan kan det komma antingen fler element eller data (se avsnitt 6.1.2). Om elementet innehåller data så är varje separat data innesluten mellan en start-tag och en slut-tag. När man läser ett XML-dokument kan man läsa det tag för tag och eventuellt utföra uppgifter specifikt för den taggen. I exemplet nedan så kan man skapa en ny transaktion när man läser den första transaktions-taggen, däremot finns det inget att spara där. När man läser Id-taggen så finns det ett värde att spara och när man kommer till slut-taggen för transaktionen så markerar det avslutningen för transaktionen. Då kan man göra något med hela transaktionen, till exempel spara den för att sedan fortsätta med nästa transaktion om det finns någon.



Figur 13 Exempel på element och data i en XML-fil.

I vårt testprogram gjorde vi så att vi läste in alla data för en transaktion (se avsnitt 6.1.2), sparade det i databasen och läste sedan in nästa transaktion om det fanns fler.

6.3.2 Uppdatering av databasen

När vi läst in en transaktion skapar vi en anslutning till databasen, uppdaterar tabellen och stänger sedan anslutningen. Det är lite omständligt och förmodligen slöseri med systemresurser då man får öppna och stänga en anslutning för varje transaktion som skall lagras. En bättre lösning skulle vara att läsa in alla transaktioner och sedan uppdatera databasen. Men då detta är en testversion så blev det mer pedagogiskt att göra en sak i taget.

Eftersom värdena som skall lagras i SQL Server ligger som text i XML-filerna och då man skapar textsträngar som SQL-uttryck när man skall uppdatera databasen så var det aldrig några problem att konvertera mellan olika typer på abonnentsidan.

7 Uppbyggnad av programmet

7.1 Inledning

I detta kapitel beskriver vi delar av vår kod. Programmen består av ett antal klasser som vi beskriver i kommande avsnitt. Vi går igenom programmen i den ordning som vi har beskrivit dem i förra kapitlet, det vill säga att vi börjar med logg-läsaren, därefter tar vi FTP-klienten och slutligen beskriver vi XML-läsaren.

7.2 Logg-läsaren

Loggläsaren består av två klasser. Den första, *TransactionSet*, har funktionalitet för att lagra flera transaktioner samt utföra operationer på dessa transaktioner. De enskilda transaktionerna lagras i transaktionsobjekt som är skapade av klassen *Transaction*. De innehåller data för de enskilda transaktionerna.

Programmet börjar med att läsa in den databastabell som skall replikeras för att ta reda på vad den har för struktur. (Vi har begränsat oss till att replikera tabeller här.) Det vi tar reda på är vad kolumnerna i tabellen har för namn och typ samt givetvis hur många kolumner det finns i tabellen. Dessa värden skickas med när vi skapar *TransactionSet-objektet*.

Därefter läser vi i transaktionsloggen för aktuell databas och ser om det finns några nya transaktioner där. Nya transaktioner identifieras dels med *Lsn* dels med *Operation*. *Lsn* skall vara större än det *Lsn* som lästes i den senaste transaktionen från förra läsningen. Det värdet sparas och om programmet skulle avslutas så skrivs värdet till disk. *Operation* skall vara en av `LOP_DELETE_ROWS,` `LOP_INSERT_ROWS` eller `LOP_MODIFY_ROW`. Finns det nya transaktioner så skapas ett objekt från Transactions och de nya transaktionerna läggs till i en lista. Därefter gås listan igenom transaktion för transaktion och skrivs till en XML-fil. På så sätt kommer varje koll i transaktionsloggen att resultera i en XML-fil. Undantaget är om det inte finns några nya transaktioner i transaktionsloggen då skapas inte heller någon XML-fil.

7.3 FTP-klienten

FTP-klientdelen av vårt projekt är uppbyggd av ett antal klasser vilka presenteras nedan. Namnet på dessa klasser är: *Service1*, *ErrorOut*, *Creceipt*, *Errorlog*, *PublicData* och *WinInetMethods*.

7.3.1 Service1

Service1 är den klass som skapar vår Windows-tjänst. Denna har en timer-funktion och när timern utlöses anropas metoden *TimerFired()*. Det är denna funktion som styr resten av programmet. Här skapas ett objekt av typen *WinInetMethods* och sedan skapas en anslutning mot servern. Om anslutningen går bra hämtas en lista på alla filer som finns på servern och sedan hämtas filerna en efter en tills alla är hämtade. Om filhämtningen går bra så skickas ett meddelande till *WinInetMethods* att skicka ett kvitto på de mottagna filerna och därefter avslutar den anslutningen samt startar om timern.

7.3.2 ErrorOut

ErrorOut innehåller en enda metod vilken heter *GetError ()*. Vid kommunikation med *GetError ()* skickas ett argument till metoden vilket är en felkod[*] av typen long. *GetError ()* översätter felkoden till ett textmeddelande som sedan kommer att skrivas i loggfilen.

7.3.3 Creceipt

Klassen *Creceipt* är till för att skapa en fil vilken kommer att skickas som ett kvitto till servern. Den har två metoder, nämligen *receiptToLog* och *RecievedFilesToLog*. Den förstnämnda skapar själva loggfilen och skriver däri vilken tid loggfilen gäller för, samt namnet på alla filer som fanns tillgängliga vid inloggning i servern.

Den sistnämnda metoden, *RecievedFilesToLog*, skriver i loggfilen hur många filer som togs emot och hur dags filöverföringen avslutades. Båda metoderna tar som argument en lista med filnamn och en textsträng som i själva verket är ett meddelande till systemadministratören.

---

[*] Felkod: FTP skickar ett heltal om operationen misslyckas där varje heltal som skickas tillhör exakt ett fel. Till exempel, om servern inte svarar inom en viss tid så skickas felkoden 12002 eller om användarnamnet är fel skickas 12013.

7.3.4 Errorlog

Klassen *Errorlog* skapar en loggfil precis som *Creceipt*, fast det finns en väsentlig skillnad mellan dem. Till skillnad från *Creceipt* som skapar en loggfil över lyckade överföringar eller operationer, skapar *Errorlog* en loggfil där alla felmeddelanden sparas för att systemadministratören vid något senare tillfälle skall kunna se vad som orsakade felet. Klassen har en enda metod, *ErrorToLog* och den tar emot två argument, nämligen felkoden och en textsträng vilket är betydelsen av felkoden.

7.3.5 PublicData

Klassen *PublicData* innehåller ett antal variabler och konstanter som är synliga för alla.

7.3.6 WinInetMethods

Med all säkerhet kan vi säga att klassen *WinInetMethods* är hjärtat av vår FTP-klient, och det är här som de viktigaste processerna äger rum. Klassen *WinInetMethods* använder sig av Microsofts WinInet API för att skapa en anslutning, logga in på servern, ta emot data, skicka data samt skicka olika andra kommandon till FTP-servern. Klassen innehåller ett stort antal funktioner, och vi kommer här beskriva några som vi tycker är extra viktiga. Dessa är: *InternetOpen*, *InternetConnect*, *FtpPutFile*, *FtpGetFile*, *FtpFindFirstFile*, *FtpFindNextFile*, och *InternetCloseHandle*.

Alla ovannämnda funktioner är alias till andra funktioner i WinInet API som importeras med kommandona

```
Declare Function FunctionsName Lib "wininet.dll" Alias
"FunctionTobeImportedt"(Arg[1], Arg[2] , ... , Arg[n]).
```

Till exempel om vi vill importera WinInet-metoden *InternetConnectA* skriver vi följande:

```
Private Declare Function InternetOpen Lib "wininet.dll" Alias
"InternetOpenA"(Arg[1], ... ,Arg[n]) As Integer
```

När man ska använda *InternetOpen* anropar man den – som vilken funktion som helst. Nedan följer en kort beskrivning av varje metod.

InternetOpen

InternetOpen är alias till WinInet:s InternetOpenA. Den skapar en anslutning mot Internet. *InternetOpen* tar emot fem argument och returnerar ett heltal.

InternetConnect

InternetConnect är alias till WinInet:s *InternetConnectA*. Det är med *InternetConnect* som man ansluter sig till en server.

FtpPutFile

FtpPutFile skickar en fil till servern. Vid anrop måste man både ange namnet på fjärrfilen och namnet på filen som kommer att skapas samt den fullständiga sökvägen till den lokala filen.

FtpGetFile

FtpGetFile hämtar en fil från servern.

FtpFindFirstFile

FtpFindFirstFile hämtar namnet på första filen i den aktuella katalogen på servern.

FtpFindNextFile

FtpFindNextFile hämtar namnet på nästa fil i den aktuella katalogen på servern.

InternetCloseHandle

InternetCloseHandle avslutar anslutningen.

7.4 XML-läsaren

XML-läsaren består, liksom logg-läsaren, av två klasser. En *XML-Reader* läser i XML-filen och returnerar värden för de olika elementen. Det finns även en klass för att lagra dessa värden.

Programmet kollar först om det finns filer som skall läsas. Den hämtar dessa om det finns några och lägger dem i en array. Dessa skickas sedan till *XML-Reader*s konstruktor. *XML-Reader* har en konstruktor (`New (ByVal p As String, ByVal files () As FileInfo))` som tar emot en sökväg och en array med filer. Filerna sorteras därefter med avseende på vilken tidpunkt de skapades. Slutligen öppnas den första filen i den nya sorterade arrayen och läsningen kan börja. Programmet skapar ett objekt ur en transaktionsklass. Den klassen har som enda uppgift att lagra värdena för en transaktion. Vi skapar inte flera sådana transaktionsobjekt utan samma objekt återanvänds om det skulle finnas flera transaktioner.
Därefter läser programmet med hjälp av *XML-Reader*-metoderna `get_transaction_id`, `get_action`, `get_col_type` och `get_col_value` in vad transaktionen har för id (`get_transaction_id`) och vad den skall utföra (`get_action`) för att därefter läsa in kolumnerna. Dessa innehåller namn, typ och värde. Programmet läser in dessa värden med metoderna `get_col_type` och `get_col_value`. I testprogrammet använder vi inte namnet på kolumnen så den har vi inte lagt till någon metod för ännu. När en transaktion är inläst så används dessa värden för att uppdatera en databastabell. Finns det fler filer och transaktioner så läses dessa därefter in en i taget och sparas på ovan beskrivna sätt.

8 Tester

Hela tiden under utvecklingens gång har vi kört olika tester för att se hur vårt program fungerar. Vi gjorde även ett antal tester när vi ansåg att vår applikation var färdigutvecklad.

När det gäller dataöverföringen tänkte vi oss ett antal scenarier som kan uppstå när man kör programmet i det verkliga livet. Dessa scenarier var:

- Användarnamnet eller lösenordet är fel vid inloggning
- Servern svarar inte, till exempel på grund av att den har tagits ner för service.
- Dataöverföringen avbryts, till exempel på grund av strömavbrott.

För att se vad som händer när man skickade fel användarnamn till servern försökte vi logga in i servern med ett användarnamn som inte fanns. Då fick vi felmeddelandet: *inkorrekt användarnamn eller lösenord*. Vi testade även med att ange rätt användarnamn men felaktigt lösenord och då fick vi samma felmeddelande som ovan.

Nästa test var att logga in på en FTP-server som inte fanns, det vill säga vi stängde av FTP-servern. Vid detta försök fick vi felmeddelandet: *Servern svarar inte, timern startas om*.

Slutligen ville vi även testa vad som händer om man avbryter dataöverföringen, till exempel genom att stänga av serven medan den skickar data till klienten. Detta test fungerade ganska bra, vi fick för det mesta meddelande om att det uppstod ett fel vid filöverföringen. Men det hände även att vi inte fick något meddelande alls. Om till exempel två av tre filer hade laddats ner och det blev fel vid nerladdningen av den tredje filen då skickades inget kvitto på de mottagna filerna och inte heller gjordes något försök att ta bort filerna på servern.

Logg-läsaren fungerade korrekt i samtliga tester om vi utförde ändringarna på ett korrekt sätt. Med det menar vi till exempel om man ville ta bort en post så skall hela posten tas bort och om man gör en INSERT så skall rätt data-typ skrivas i korrekt fält.

De tester som vi utförde på XML-läsaren var bl.a. att lägga in andra filer i samma katalog som XML-filer fanns, med detta försök vill vi se hur XML-läsaren uppförde sig om det fanns andra fil-typer i samma katalog när den skulle läsa XML-filer. Från början hade vi lite problem med det då den läste in även andra filer än XML-filer vilket ledde till att programmet kraschade. Detta löste vi genom att programmera om XML-läsaren så att den läste in endast filer som hade händelsen ”*.XML*”. Efter ändringen fungerade programmet korrekt i alla våra tester, även när vi försökte hämta data från en tom katalog.

9 Slutsatser

När vi började på vårt examensjobb hade vi ringa kunskaper inom områdena datareplikering och programmering i Microsoft Visual Studio .NET. Idag, efter flera månaders undersökningar och arbeten, vet vi en hel om olika replikeringsmetoder. Dessutom har vi lärt oss att programmera i Visual Basic .NET vilket ingen av oss kunde innan. Ett annat område som var nytt för oss och som vi idag kan en del om är hanteringen av XML-filer. Det har varit givande och intressant att utföra detta arbete. Vi har också lyckats programmera en exempel-applikation som har fungerat ganska bra i våra tester.

Den kanske enklaste replikeringsmetoden är att använda Microsofts inbyggda replikering. Fördelen med att använda inbyggda replikeringsmetoder är att man inte behöver programmera, utan systemadministratören konfigurerar databasen så att den replikerar till en abonnent-databas. En nackdel med MS inbyggda replikering tycker vi är att man måste ändra på brandväggar vilket kan innebära säkerhetsrisker. En annan nackdel med denna metod kan även vara att man måste administrera domäner och filåtkomst och liknande. En annan metod för replikering är att själv konstruera ett program som hanterar det. Vi valde den sistnämnda metoden efter att vår uppdragsgivare önskade att de helst inte vill ändra konfigurationen på sina brandväggar. Vår lösning är mer lättadministrerad än Microsofts, tror vi, då den inte kräver mer än skrivrättigheter i en katalog samt läsrättigheter i databasen. Då programmet är uppbyggt i moduler så är delarna så att säga skyddade från varandra. När vi skapar en replikering sparar vi den i en XML-fil. För att skapa en replikering använder vi transaktionsloggen. På grund av tidsbrist har vi inte kunnat implementera någon sorts felhantering i koden. Man skulle kunna lägga till felhanteringen i koden för att göra programmet robustare.

För att det ska vara lättare att utföra våra tester har vi implementerat både transaktionsläsaren och XML-läsaren som en vanlig Windows-applikation. Man skulle ytterligare kunna förbättra den genom att implementera det som en Windows-tjänst och med hjälp av en timer skulle man kunna automatisera applikationen så att den replikerar regelbundet.

Förutom replikeringsfilerna ville man ju överföra en del andra binära filer också. Vi valde FTP för överföring av dessa filer över Internet. Vi tycker att vår filöverföring fungerar ganska bra men det finns en hel del förbättringar som skulle kunna göras om man vill gå vidare med den. Följande är ett par exempel på förbättringar som man skulle kunna göra:

- Förbättra felhanteringen ytterliggare. Den felhantering som vi har implementerat innefattar endast de fel som inträffar under uppkopplingsfasen, nedkopplingsfasen samt vid dataöverföringen. Det vill säga fel som har med protokollet att göra och inte systemfel.
- Man skulle kunna implementera en krypteringsalgoritm för att skydda informationen som skickas.
- För att ytterligare förbättra säkerheten skulle man kunna implementera en metod som möjliggör dynamiska lösenord.
- En annan förbättring vore att implementera stöd för återuppladdning av filer som tidigare misslyckats med nerladdningen.

Referenser

[1] Planning for transactional replication(MSDN) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/replsql/replplan_1l4e.asp Mars 2003

[2] Rick Sawtell och Richard Waymire. *Lär dig SQL Server 7.0 på tre veckor*, Sams, Första tryckningen, 1999

[3] James F. Kurose och Keth W. Ross. *Computer Networking*, Addison Wesley, 2001

[4] FTP versus HTTP

http://www-cad.eecs.berkeley.edu/~mds/research/1995/http.html Mars 2003

[5] Replication Types (MSDN) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/howtosql/ht_repl_sph_7ka0.asp Mars 2003

[6] SQL Server (MSDN) http://msdn.microsoft.com/nhp/default.asp?contentid=28000409 Mars 2003

[7] Introducing Replication (MSDN) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/replsql/replintro_5ir2.asp Mars 2003

A Bilaga

A.1 Modify

Rubrik	Bytes	Annat
Längd	4	De två första byten verkar inte användas.
Current LSN	12	
Transaction ID	4	
Okänt	20	
Previous LSN	4	
Flagbits	4	
Ändring	4	De två första byten är vilken kolumn som ändringarna börjar i. Om värdet på de två första byten är x så är första kolumnen som ändrats nummer x/4. De två sista är hur många columner som ändrats. Om x är värdet av de två sista byten så är antalet kolumner som ändrats (x + 3) / 4
Okänt	8	
Gamla värden	4	*Antal ändrade kolumner (se ovan) vid Integer
Modifierad rad	4	*Antal ändrade kolumner (se ovan) vid Integer
Okänt	4	

A.2 Insert

Rubrik	Bytes	Annat
Längd	4	De två första byten verkar inte användas.
Current LSN	12	
Transaction ID	4	
Flaggbitar	4	
Okänt	36	
Radnr	4	Radnummer eller primary key
Kolumnvärde	4	4 ggr antal kolumner
Okänt	4	

A.3 Exempel på en XML-fil med en transaktion.


```
<?xml version="1.0" ?>
- <!-- Replikering 2003-04-20 13:37:40 -->
- <Transactions>
  - <Transaction>
      <TransactionID>0000:000002d3</TransactionID>
      <Action>LOP_MODIFY_ROW</Action>
    - <Columns>
      - <Column>
          <Name>rowid</Name>
          <Type>System.Int32</Type>
          <Value>41</Value>
        </Column>
      - <Column>
          <Name>origin</Name>
          <Type>System.Int32</Type>
          <Value>not modified</Value>
        </Column>
      - <Column>
          <Name>dbid</Name>
          <Type>System.Int32</Type>
          <Value>1118483</Value>
        </Column>
      - <Column>
          <Name>cust_nr</Name>
          <Type>System.Int32</Type>
          <Value>19</Value>
        </Column>
      - <Column>
          <Name>order_nr</Name>
          <Type>System.Int32</Type>
          <Value>not modified</Value>
        </Column>
      - <Column>
          <Name>ref_nr</Name>
          <Type>System.Int32</Type>
          <Value>not modified</Value>
        </Column>
      </Columns>
    </Transaction>
  </Transactions>
```

A.4 Tolkning av hexsträng

Här är data för den post vi har tagit bort

	Int	Hex
Post	22	16
Kolumn 1	121726	1DB7E
Kolumn 2	78	4E
Kolumn 3	98	64
Kolumn 4	976	3d0
Kolumn 5	687	2AF

Här är vår tolkning av den hexsträng som vi fick från transaktionsloggen. Vi har grupperat byten i grupper om fyra bytes det vill säga åtta hexadecimala siffror.

00006800 08000000 6A010000 01000313 9D020000 00001200 81000000 01000C00
09B8272D 08000000 69010000 03000000 01000000 01002F00 10002C00 16000000
7EDB0100 4E000000 62000000 D0030000 AF020000 07000000

Längd - 4 bytes

00006800

Loggsekvensnummer (LSN) – 12 bytes

08000000 6A010000 01000313

Transaktions Id - 4 bytes

9D020000

Flagbits – 4 bytes

00001200

Info? – 36 bytes

81000000 01000C00 09B8272D 08000000 69010000

03000000 01000000 01002F00 10002C00

Post - 4 bytes

16000000

Kolumnvärden i posten – 4 bytes gånger antal kolumner vis Integervärden

7EDB0100 4E000000 62000000 D0030000 AF020000

Trailer - 4 bytes

07000000