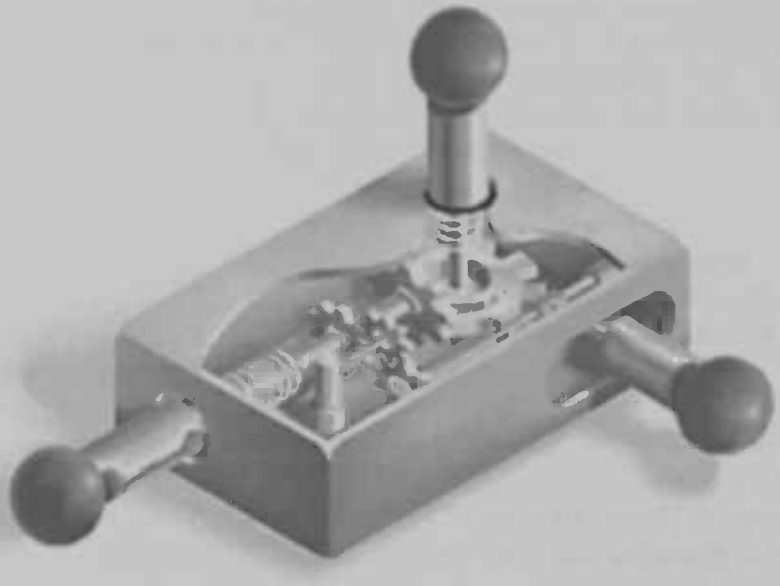
wordt
NIET
uitgeleend

Institute for Mathematics and Computing Science

M.Sc. Final Research Project

System Wide Database Notification Service

Samvel Petrosyan



· April 2007 ·

This thesis is submitted to the Department of Computer Science at Rijksuniversiteit of Groningen in partial fulfillment of the requirements for the degree of Master of Science in Software and System Engineering.

Contact Information
Author: Samvel Petrosyan
E-mail: s.petrosyan@student.rug.nl

Supervisors:
Dr. Paris Avgeriou
Department of Mathematics and Computing Science
Software Engineering and Architecture (SEARCH) Group
University of Groningen
Email: paris@cs.rug.nl

Richard Storm
Head of Software development dep.
ChipSoft BV.
Email: richard@chipsoft.nl

Abstract

There is an increasing amount of data in databases that is shared among many client computers. Some of this data may be relatively static. However, certain types of data may be changing frequently. Client applications may be interested in changes that are made to these data. Notifications of such changes allow a client to be made aware of changes that are made by another client application in a timely fashion. In this paper, I describe my work on exploring, designing, implementing and integrating of a notification mechanism in the CS-ECIS application developed at the company ChipSoft. To achieve this goal is the publish/subscribe technology, which is emerging as an appropriate communication paradigm for large-scale distributed systems, used. In the designed architecture, each client application defines and advertises change events to the notification service. The clients can subscribe to events of interest, and can refine their subscriptions through attribute-filter expressions. A notification is sent whenever a database change, detected by the notification service, matches some active subscriptions.

Contents

List of Figures

List of Tables

Notation and Abbreviations

API Application Programming Interface

ASP Active Server Pages

BL Business Logic

COM Common Object Model

CS-ECIS Electronic Care Information System

CQ Continuous Queries

DBMS Database Management System

DCOM Distributed COM

DLL Dynamic Loadable Library

DTC Diagnosis Treatment Combination

ECA Event-Condition-Action

EPR Electronic Patient Record

GUI Graphic User Interface

GUID Globally Unique Identification

HR Hardware Requirement

ID Identification number

JS Java Script

LAN Local Area Network

LAZR Landelijke Ambulante Zorg

LCE Loosely Coupled Events

MS Microsoft Co.

OK Operatie Kamer (Operation Room)

OLTP Online Transaction Processing

OOP Object Oriented Programming

PDA Personal Digital Assistant

RPC Remote Procedure Call

SAAM Software Architecture Analysis Method

SFR Software Functional Requirement

SMTP Simple Mail Transfer Protocol

SNFR Software Nonfunctional Requirement

SQL Structured Query Language

TCE Tightly Coupled Events

TLB Type Library

TPC Transaction Processing Performance Council

VCL Visual Component Library

VB Visual Basic

WML Wireless Markup Language

XML Extensible Markup Language

Acknowledgments

I would like to express my heartfelt thanks to my supervisor at ChipSoft, Richard Storm. His thoughtful support and guidance throughout the duration of this project truly helped make this one of the most useful learning experience of my study. I would like to thank all of colleagues at ChipSoft for so selflessly giving of their time to help me to understand the architecture and the working of CS-ECIS. I would like also thank my supervisor at RuG, Paris Avgeriou for reviewing and giving helpful hints for improvement of this document.

Chapter 1

Introduction

1.1 The Company

ChipSoft is a software house that has been specialized in computerization in health care and from the outset it is focused exclusively on supplying the health care sector. In 1986, ChipSoft began developing software products for medical specialists and various hospital departments. In 1994, ChipSoft introduced its first hospital information system. In 2001, ChipSoft launched the first integrated Windows-based care information system: the Electronic Care Information System (CS-ECIS). ChipSoft is a successful, innovative, flexible and dynamic company with a contemporary vision of computerization in health care. On the basis of a thorough understanding of the developments within the health care sector, ChipSoft provides accurate ICT solutions. The Windows-based CS-ECIS offers a seamless solution for the increasing need for flexible and deployable functionality and transmural communication facilities. Although the products of ChipSoft are already used in six European countries, ChipSoft is still a purely Dutch software house.

1.2 CS-ECIS Application

The CS-ECIS application is based on various care processes and divided into a few concepts, such as: inpatients, outpatients, order communication, EPR (Electronic Patient Record) generation, invoicing and procedure administration. In a single system, patient records, planning schedules, patient flow organization, medication and administrative transactions are processed and made readily comprehensible. The CS-ECIS is an evolving system and has the foundations for future expansions. These expansions can lead the application beyond the traditional boundaries between medical and management information. The building blocks of the CS-ECIS are individual functions, which can be assembled as components in a configure-it-yourself user interface, based on the user's specific work processes. On the basis of their function profile, users can consult or register various data.

Figure 1.1 depicts the conventional CS-ESIC database configuration, wherein the users access information stored in the database. The application is divided into the following three tiers: the graphic user interface, the application business logic and the database drivers. The application and the database interact in a client/server relationship, where the application is the client and the database is

the server. Application business logic establishes a connection to the database using the database driver. The application business layer uses the API provided by the database driver abstraction layer. This allows the application to have more than one database driver in a transparent for the application manner.

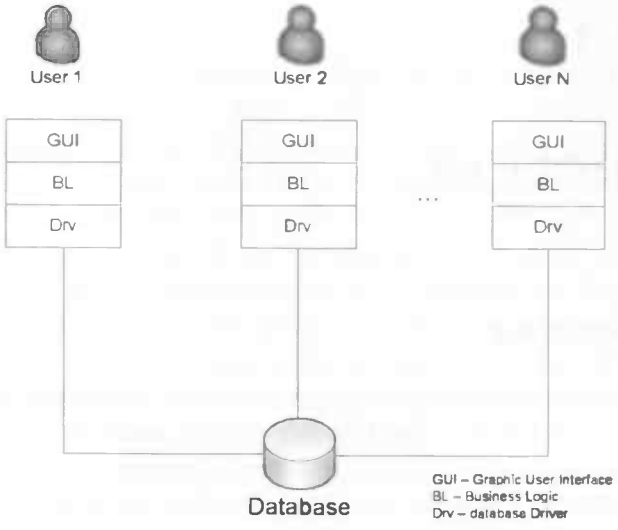


Figure 1.1: Database configuration of CS-ECIS

Data retrieved from the database displayed to the user in grids and other GUI elements. In order to keep data in the application up-to-date (i.e. sync with the database), the client applications poll the database once in two minutes (or once in 30 seconds dependent on functionality) and refresh stale collections of data. Because of this, the ability of CS-ECIS to display critical and timely data is restricted by the substantial amount of time delay between these data refresh cycles.

1.3 Problem Definition

The CS-ECIS enterprise system is a complicated database system which is extending rapidly with respect to functionalities and function groups of the system. The system based on three-tier client server architecture. The several hundred clients working with shared data in the database, hold passive data such as views, records etc. Each client has own view on the current state of shared data. If the state of shared data in question is changed, the data will not be able to notify the clients about this change. That means that the client views become out of sync with the database and the users get inconsistent views. This requires continuous fetching data (i.e. polling) out of the database server to refresh data collections. The above described situation is depicted in figure 1.2.

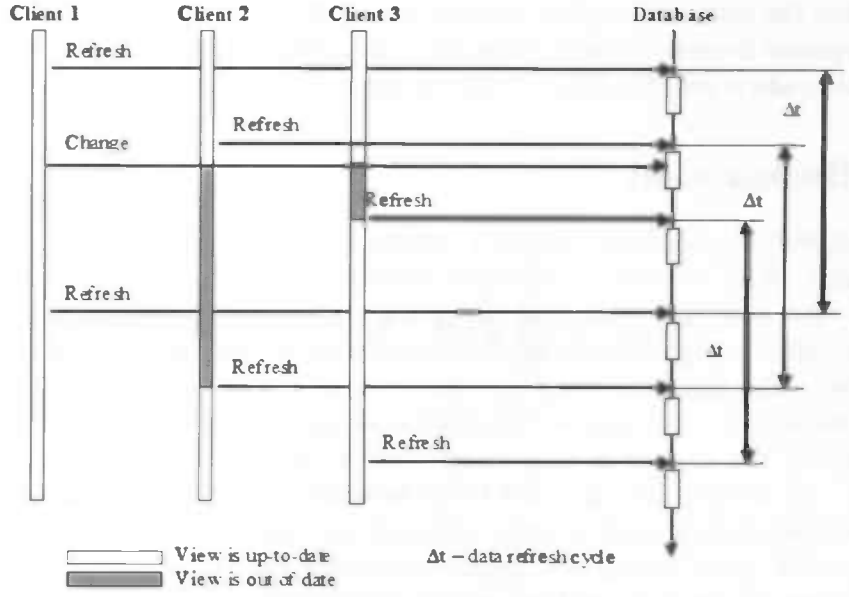


Figure 1.2: Client views become out of sync

The clients refresh their data collections once in Δt period of time. If the client 1 changes a record in the database, depending on the polling time-frame/frequency, other clients (in this case client 2 and client 3) become temporarily inconsistent with the database. These clients have to wait maximum Δt period of time to get the changed data.
However, the main problem is that the system constantly deals with big number of growing new users (physiotherapists, nurses etc.). Because of this, constantly polling the database has created capacity and performance problems. The decreasing performance has become a bottleneck of the system. At the present, some complex database forms/screens in the system are refreshing once every 30 seconds, regardless the data has changed or not. This results in heavy load on the network and on the database. Simple operations (e.g. making an appointment, receiving a reception etc) take much longer than acceptable. In order to improve the performance of the system, ChipSoft desires to develop a system and methods for automatically notifying users upon the occurrence of selected events (e.g. changed data in the database). In addition, it would be desirable for such notification system to be able to handle different types of events and to present notifications to the users in different ways.

1.4 Purpose

Currently, the users of CS-ECIS are working with views that often contain out of date data. Beside this, the increasing number of users causes high database and network load. This all makes difficult to efficiently perform work in hospitals and care centers. ChipSoft desires to improve the usability of CS-ECIS and reduce the current load of the system by introducing a database change detection and user notification facility, so that the system can handle the increasing capacity of users and decrease likelihood of stale data. The purpose of this project is to help Chip-

Soft to find the roots causing low performance of the system, analyze them, design an appropriate database change detection and notification mechanism, implement it and integrate it into CS-ECIS environment.

1.5 Background

Many computer applications utilize a database to store, retrieve and manipulate information. Such systems have a wide range of usage in health care information systems. For example, a hospital department might use a database to store information about their patients, appointments with doctors, occupation of surgery rooms, etc. The database store is a collection of data structures organized in a disciplined fashion that makes stored information of interest quickly accessible. The retrieval and display of this information is essential in modern information systems. To achieve this goal CS-ECIS provides access to information on the database that enables users to enter, organize and select data in the database.
In the last few years, closely controlled environments have evolved into large scale collaboration spaces that are both highly distributed and dynamic. This evolution has led to new requirements (e.g. displaying timely and critical data, etc), as well as to new application domains for distributed information systems. Examples arise in health care information systems, business process (workflow) control, network management, etc. These areas share the same basic requirement of tracking changes at a collection of information sources in order to detect situations of interest. The information system must identify relevant changes and notify them to users or client applications, without requiring them to know a priori where and when to look for data. The challenge of this project is to provide CS-ECIS the necessary mechanisms and strategies for a scalable, reliable and secure notification service to meet the new requirements.

Chapter 2

Performance of Existing System

2.1 Introduction

One of the goals of this project is to reduce the load on the network and on the database server to an acceptable level by reducing the refresh queries to minimum. In order to find the roots of the performance bottleneck in the system, a performance test is performed. There can be a few sources that can cause a performance bottleneck in the system. One of them is the refresh queries. The ECIS client application consists of several screens which must be always kept up-to-date with the database. Each of these screens regularly issues one or more refresh queries to the database. The number of the refresh queries depends on the number of opened screens in the ECIS client at the moment. The aim of this test is to make an estimation about the frequency and the quantity of these refresh-queries in the system. Further, it has to be examined if these queries cause a bottleneck in the system.

2.2 Test Environment

The test is performed on the production database at the hospital of SFG (Sint Franciscus Gasthuis in Rotterdam). MS SQL 7.0 on a dual processor server as the production database, and ECIS 4.8 as client application are used during the test.

2.3 Test Description

During the performance test, the queries generated by 251 online clients has been traced and logged. There are two types of online clients: active clients and passive clients. An active client is referred to the client which user was active working on the database. A passive client is referred to the client that was idle during the test and has only issued refresh queries. During 6 minutes were all the queries fired by 251 active/passive clients captured and stored in the database. The logged queries consist of a few comment lines and a native SQL query. The comment lines provide debug information about the query, such as the logic from which the query is originated, the action during which the query is fired, the scope and

the filter of the query, etc. In order to estimate the performance of the database server is the reference TPC-C benchmark used. TPC Benchmark C is an on-line transaction processing (OLTP) benchmark. TPC-C is an industry standard benchmark for measuring the performance and scalability of OLTP systems. It tests a broad cross-section of database functionality including inquiry, update, and queued mini-batch transactions. Many IT professionals consider TPC-C to be a valid indicator of "real-world" OLTP system performance. The benchmarks of Compaq servers (we take an average Compaq server as a reference point) have shown that a four processor database server scores an average of 100000 tpmC (transactions per minute of type C). It means that a dual processor system would be limited by approximately 833 transactions/sec. In order to avoid the bottleneck, the number of transaction in the ECIS system should be below this limit. The ECIS clients are programmed to use automatic transactions which means that one transaction comprised from one query. So the number of queries is equal to the number of transactions in the system. Further in this section, all calculations applied to queries are the same for transactions.

2.4 Data Analysis

Working with the ECIS client requires from the user to keep number of screens open simultaneously. One of the opened screens, that is active at the moment, fires "normal" queries to retrieve data from the database. Other screens, at the background, fire refresh-queries in order to refresh data and keep data collections up-to-date with the database. It is very difficult task to distinguish the "normal" queries from the refresh-queries among all logged queries. It took a lot of time from me to dig these refresh-queries out and make an estimation.
During 6 minutes, there were total 65584 queries logged in the database. If we assume that the queries are uniform distributed over the total time of the test then we get an average query-rate of:

$$65584 \text{ queries}/6 \text{ min} = 182 \text{ queries}/ \text{ sec} \tag{2.1}$$

This is not so much for an advanced database server such as MS-SQL. However, this estimation of the average query-rate may not represent the reality. The query-rate depends on several variables (such as the type and a number of open screens in ECIS, how long a screen was open, the way of behaving of the user, the time when the test is taken, etc.) that have sporadic behavior. However, the analysis of the logged queries revealed many patterns of SQL queries which have regular basis and it seems that they are generated by the refresh timers of passive ECIS clients. An example of such query is shown below:

```
* Generated by ChipSoft Base 4.7.0.27
 *
 * ADO Version : 2.7
 *       Logic : CSAgendaService2.CS_AfspraakMutatieLogic
 *      Action : CreateView(Scope, Filter)
 *     Trigger : NeedsLast
 *  ScopeOrder : SubagendaID,Mutatiedatum,Mutatietijd
 * ScopeFilter : (<SubagendaID> = 'S00034' EN <Mutatiedatum> = 05-10-2005 EN
                                        <Mutatietijd> >= 01-01-0002 15:27:09)
 *      Filter : Altijd Waar
 */
```

```
sp_executesql N'
SELECT TOP 20 A.[AFSPRAAKNR],A.[DATUM],A.[TIJD],A.[CONSULTNR],A.[AGENDA],A.[SUBAGENDA],
              A.[PATIENTNR],A.[AFSPTYPE],A.[COMBINR],A.[INVOERDAT],A.[DOORWIE],
              A.[TERMIJN],A.[OMSCHR],A.[CODE],A.[DUUR],A.[DIAGNOSE],A.[CONSTYPE],
              A.[NIEUWDECL],A.[ONTBRREDEN],A.[KLINPOLI],A.[AANVRAGER],A.[UITVOERDER],
              A.[MOBILITEIT],A.[VOLDAAN],A.[VERPLREDEN],A.[LOKATIE],A.[AUTODAT],
              A.[FAKTDAT],A.[LAZRDAT],A.[FAKTSTATUS],A.[LAZRSTATUS],A.[EPB],A.[MUTDAT],
              A.[MUTTIJD],A.[MUTWIE],A.[MUTTYPE],A.[MEMO],A.[EPISODE],A.[LISTCODES],
              A.[AANKOMST],A.[OPROEP],A.[VERTREK],A.[STATUS],A.[GROEPSNR],A.[HERHAALNR]
FROM [dbo].[AGENDA_MUTATIE] AS A
WHERE ((A.[SUBAGENDA] = @SV1 AND A.[MUTDAT] = @SV2 AND A.[MUTTIJD] >= @SV3)
                                                        AND A.[DATUM] = @SV4)
ORDER BY A.[SUBAGENDA] DESC,A.[MUTDAT] DESC,A.[MUTTIJD] DESC,A.[AFSPRAAKNR] DESC,
                                                              A.[MUTTYPE] DESC
OPTION (KEEP PLAN, KEEPFIXED PLAN, LOOP JOIN)
', N'@SV1 nvarchar(6),@SV2 datetime,@SV3 nvarchar(11),@SV4 datetime',
@SV1='S00034',@SV2='20051005',@SV3='15:27:09:91',@SV4='20051005'
```

Finding in the log all instances of this query revealed that the query is issued once each 30 seconds. It is shown in the table 2.1.

Database Id	Hostname	Duration	Start Time	Reads	Writes	CPU
8	N26293	0	15:29:55	4	0	0
8	N26293	0	15:31:09	4	0	0
8	N26293	0	15:31:24	4	0	0
8	N26293	0	15:31:54	4	0	0
8	N26293	0	15:32:24	4	0	0
8	N26293	0	15:32:54	4	0	0
8	N26293	0	15:33:24	4	0	0
8	N26293	0	15:33:54	4	0	0
8	N26293	0	15:34:24	4	0	0
8	N26293	0	15:34:54	4	0	0

Table 2.1: Pattern of queries issued by a client

The above described query is a small query that requires 4 reads from the database server. It seems that the database can easily handle these kind of refresh-queries because of the null duration of the queries. Among these small refresh-queries, I found huge blocks of refresh-queries. For instance, the following block of refresh-queries consists of 70 refresh-queries. The snipped block is shown below. The refresh cycle takes place once in 2 minutes (the second block begins at 15:34:23:023).

TextData	HostName	SPID	Duration	StartTime	Reads	Writes	CPU
/* CreateVi...	N6466	426	30	2005-10-05 15:31:22.990	574	0	31
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.083	2	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.300	91	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.300	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.317	5	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.350	37	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.363	8	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.380	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.397	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.397	73	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.410	8	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.427	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.443	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.460	85	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.473	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.473	5	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.503	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.503	73	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.520	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.537	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.550	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.567	91	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.567	28	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.583	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.600	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.613	22	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.630	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.647	6	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.660	73	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:23.677	7	0	0
/* CreateVi...	N6466	426	16	2005-10-05 15:31:23.693	4	0	15

Figure 2.1: Query trace

/* CreateVi...	N6466	426	0	2005-10-05 15:31:24.210	37	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:24.223	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:24.240	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:24.253	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:24.270	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:24.287	5	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:31:24.300	5	0	0
/* CreateVi...	N6466	426	30	2005-10-05 15:34:23.023	523	0	16
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.117	2	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.320	91	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.337	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.350	5	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.380	37	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.443	8	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.460	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.477	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.490	73	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.523	8	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.523	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.553	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.553	85	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.570	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.587	5	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.600	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.617	73	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.630	7	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.647	4	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.663	3	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.680	91	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.680	28	0	0
/* CreateVi...	N6466	426	0	2005-10-05 15:34:23.693	4	0	0

Figure 2.2: Query trace (Continued)

From picture above we can see that these queries require much more reads from the database server and some of them took pretty much time and CPU resources to execute. It is clear that too much of this kind of queries can easily cause a bottleneck in the system.
In order to get a complete picture of what is going on in the system, I made a histogram where the number of fired queries by the clients is shown. The histogram is depicted in the figure 2.3.

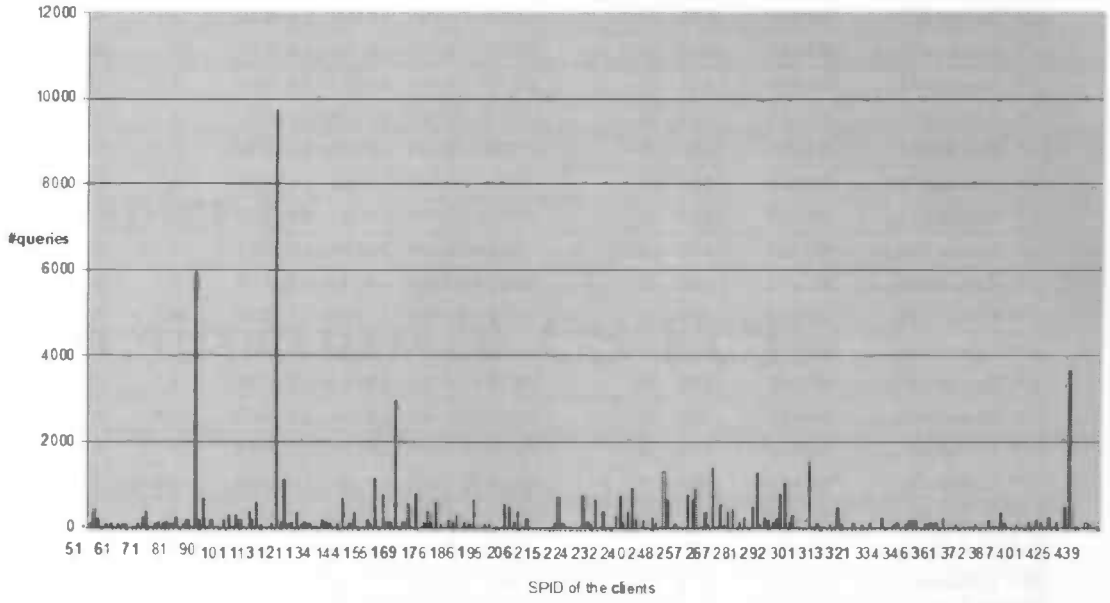


Figure 2.3: Number of queries of the clients

From the histogram we can see that about 30% (22000 queries) from all queries belongs to 4 clients. In the table 2.2, the top 10 clients with the number of fired queries are shown.

SPID (Security Process ID)	#Queries
119	9734
89	5963
434	3655
167	2963
305	1523
267	1374
250	1309
288	1271
157	1116

Table 2.2: Top 10 clients

The queries, that have been fired by these clients, are studied but unfortunately no regular patterns were found among them. That means that there were no refresh-queries queries, or blocks of refresh-queries were too long and it was impossible to recognize them.

2.5 Estimation

Many patterns of refresh-queries have been found in the sql-trace. If we assume that all 250 users have three simultaneous open screens then each screen would generate in average 60 queries/min (arithmetic average of all blocks of refresh-query patterns that are found in the trace). In the table 2.3, the estimated and

the worst case values of the number of queries are shown.

	Estimated average value	Worst case value
Number of users	250	500
Open screens	3	6
Query /min (per user)	60	120
Query / sec (all users)	750	6000

Table 2.3: Estimated and worst case values of number of queries

The above calculations show that the estimated value of the number of all queries remains under the limit of an average SQL server (833 transactions/sec). However, when the number of users and the number of open screens increases, the number of queries increases too and in worst case we get three times more queries than an average SQL server can handle. It is clear that in worst case the refresh queries can cause performance problems.
This research shows that the database queries generated by the GUI widgets of CS-ECIS can be the source of the network and the database heavy load. They can cause a bottleneck in the system. By introducing a system wide notification facility, CS-ECIS application will be able to heavily relay on cached data and will be able to decrease the number of refresh-queries sending to the database. This way, the performance of the system can be increased and the usability of CS-ECIS application improved.

Chapter 3

Architecture of CS-ECIS

The CS-ECIS application has component based architecture and entirely based on the Common Object Model (COM) technology of Microsoft. It makes heavy use of COM-objects. In order to understand the rationale behind CS-ECIS architecture, it is important to know the basic concepts of COM. In the following section I revise the basic principles of the COM technology.

3.1 Common Object Model

The Microsoft Component Object Model (COM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is a framework for developing and supporting program component objects. It provides the underlying services of interface negotiation, life cycle management (determining when an object can be removed from a system), licensing, and event services (putting one object into service as the result of an event that has happened to another object). On the one hand, the COM is a specification for writing reusable software that runs in component-based systems. On the other hand, COM can be thought of as a sophisticated infrastructure that allows clients and objects to communicate across process and host computer boundaries. COM is a model based on binary reuse. This means that software (components) adhering to COM can be reused without any dependencies on source code. From developer's point of view, using a COM-object is exactly like using a local object; however the COM infrastructure is a bit more complex. COM was built from the ground up to be object-oriented. It is based on clients, classes and objects. Classes are defined in binary files called servers. These servers make it possible for clients to create and connect to objects whose code lives in a separate binary file. For example, in CS-ECIS application each module that resides on GUI tier and each service that resides on business logic tier are COM servers. At the run-time, the application creates these servers and connects to them in order to use the services offered by these servers. After a client connects to an object, it simply invokes methods as in any other OOP environment. COM fully supports the object-oriented concepts of encapsulation and polymorphism as means to achieve code reuse. Moreover, it is completely language independent and virtually any programming language can be used to create COM powered components and application. That means that scripting languages such as VB, JS, ASP, etc. are able to use COM objects.

These languages lack of compile time interface binding that COM objects require. However, COM provides so-called interface late binding technique. Late binding is all about asking an object what methods it supports instead of what interfaces. Whereas early binding passes a GUID of the interface it is interested in to COM object, late binding presents the object with a method name in the form of a wide-character string. If, after interpreting the string, the object supports a method by that name, it returns with a number representing that method. This number is referred to as a dispatch ID or DISPID. After the client has this number, it can call the method by invoking the number.

3.2 Software Architecture

Figure 3.1 depicts the global architecture of the CS-ECIS client.

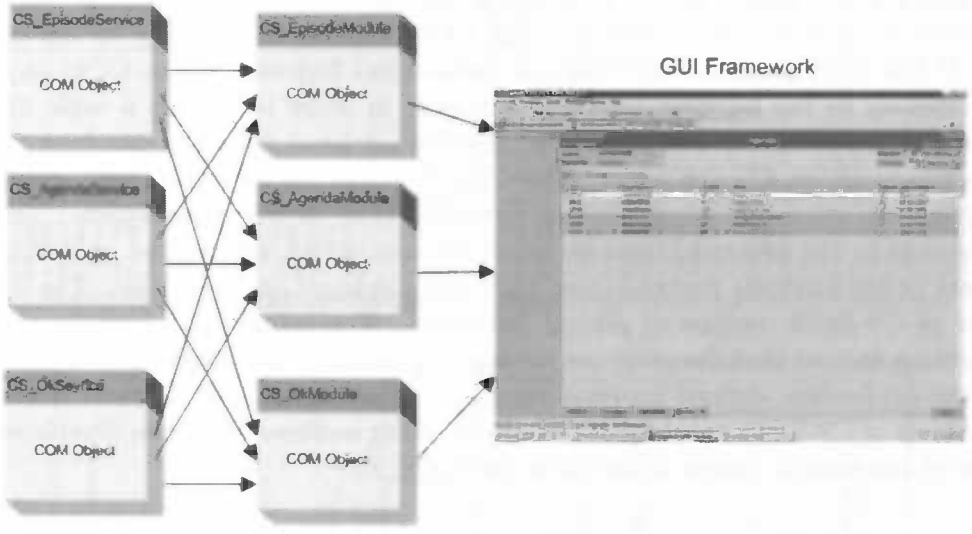


Figure 3.1: Architecture of CS-ECIS

The application consists of services, modules and a GUI framework. As I explained earlier, each module is a COM object that implements the desired functionality. The services are also COM objects and they take care of providing data to modules. The modules get desired data from services and show it in grids and other GUI controls. The framework hosts and manages the windows, exposed by modules that contain these GUI controls. Communication between these COM objects goes only via defined interfaces that the COM objects expose. The figure 3.2 shows how the framework and COM servers from different layers connect to each other via defined interfaces. For example, the framework connects to the modules from the business layer via *ICS_OK* and *ICS_Agenda* interfaces. The modules, in their turn, connect to the services from de the data layer via *ICS_OKLogic* and *ICS_AgendaLogic* interfaces.

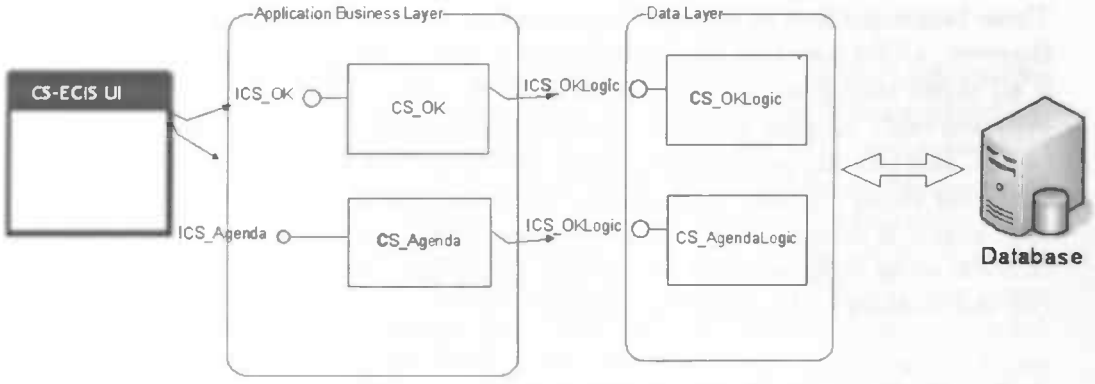


Figure 3.2: Layered architecture of CS-ECIS

Every COM object in the application belongs to the corresponding tier dependent on provided functionality. The COM objects that belong to the data tier of the application (i.e. services) are data-object factories for the COM object that belong to the business tier. For example, in order to display a table filled with OK data-objects, first, the "raw" data is retrieved from the database then it is processed by *CS_OKLogic*. The *CS_OKLogic* converts the "raw" data to a typed collection and corresponding sub-collections of OK data-objects. These data-objects are processed against business rules which are defined in *CS_OK* object in the business tier and then, they are displayed on the screen. The data layer of CS-ECIS consists of several interfaces. The cooperation between these interfaces ensures that the presentation layer operates on data which are validated by the application defined business rules. The descriptions of the most relevant interfaces of CS-ECIS are provided in the following sections. The simplified logical view of the related classes is shown in the figure 3.3.

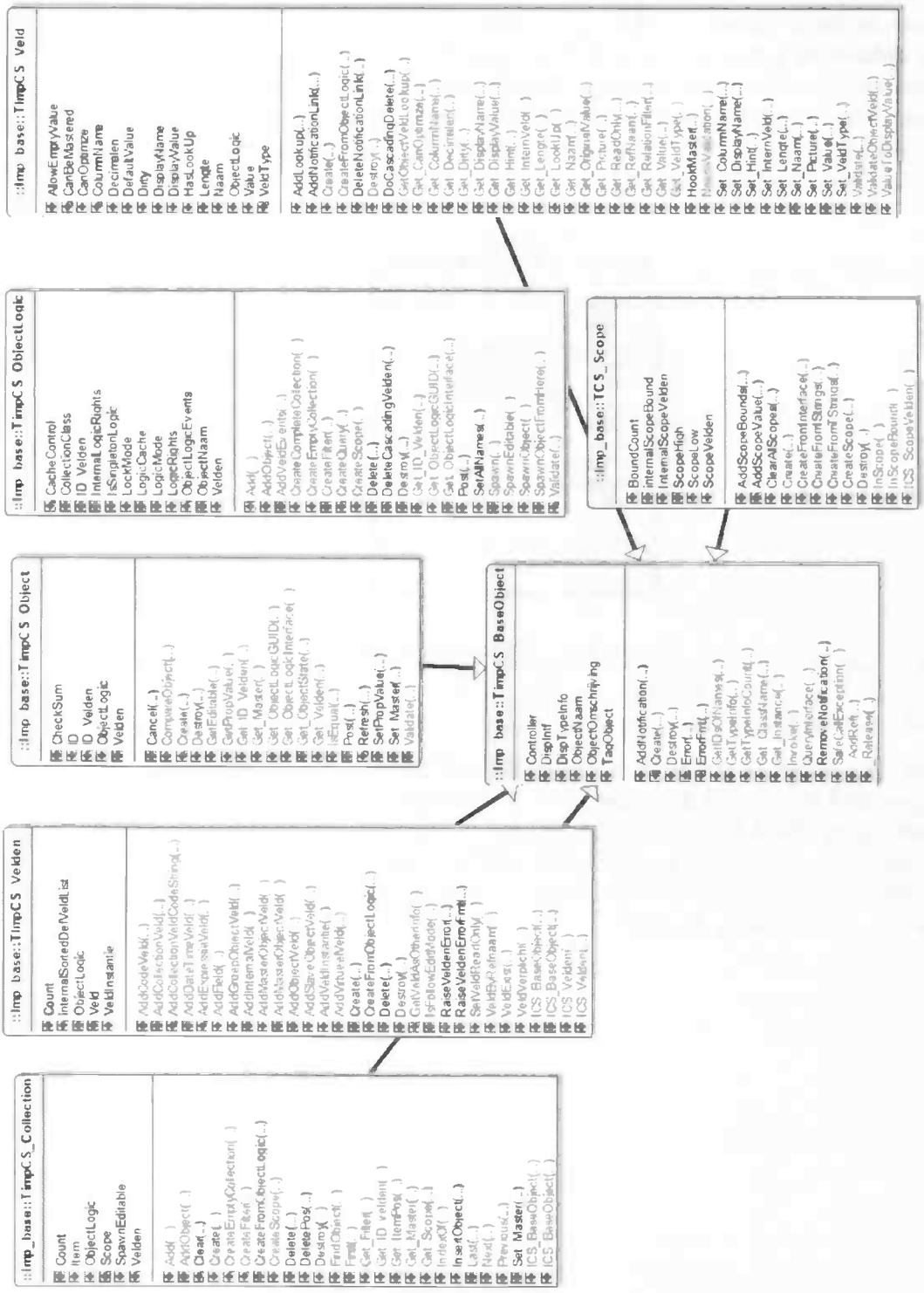


Figure 3.3: Class Diagram

3.2.1 ObjectLogics

ObjectLogic is the base class of all logics in the business layer. ObjectLogic resides on the data layer and provides data-objects to the higher layers of the application. Single data-objects are provided by means of an unique key (ID), and collections of data-objects by means of a scope and/or a filter. The ObjectLogic takes care of that the objects can be read, erased or written to the database. Before objects are written to the database, ObjectLogic validates these objects against the business rules defined in logics such as *CS_OKLogic*, etc. Usually, an ObjectLogic maps objects to one table in the database. But it can also map objects to zero or several tables. The ObjectLogics know which fields the table contains and which fields are ID fields. The interface that exposes these functionalities is *ICS_ObjectLogic*. It is shown in the following diagram.

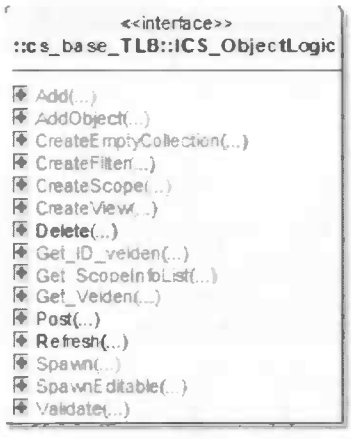


Figure 3.4: *ICS_ObjectLogic*

3.2.2 Data Objects

Every data object in CS-ECIS has *ICS_Object* interface. A data object is usually mapped to a record in a database table. It contains some fields, but in contrast to ObjectLogics, these fields really contain the values of the fields of the record which represents the object. All data objects have an unique key (ID). The data objects are created, read, stored and removed by ObjectLogics. *ICS_Object* interface is shown in the following diagram.

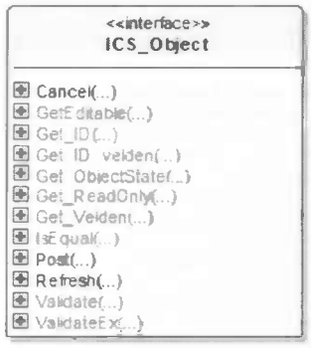


Figure 3.5: *ICS_Objects*

3.2.3 Data Fields

ICS_Velden is a simple interface. As shown in the following diagram, the interface contains nothing else but a collection of the type of *ICS_Veld*.

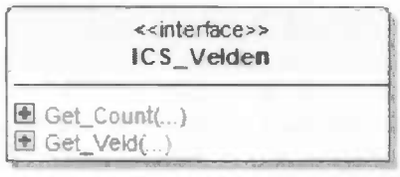


Figure 3.6: *ICS_Velden*

3.2.4 Data Field

ICS_Veld interface represents a field from a database table and it has a large number of properties. It contains the value of a field of a database table. The *ICS_Veld* can be readonly. In this case the value of the field cannot be changed. Otherwise, if the value changes, the original value of the field is kept. It makes possible to be able to undo the change. The fields are lazy updateable. When a change occurs, the modified fields become marked as dirty and ObjectLogic generates an update query only for these dirty fields. This ensures that the data transferred to the database is minimal. It keeps network and database load low. The following diagram depicts the *ICS_Veld* interface.

Figure 3.7: *ICS_Veld*

3.2.5 Collections

ICS_Collection is a collection of data objects. The collections can be obtained from the ObjectLogics, but also from an already existing collection. The collection is in the first place composed by means of criteria which are given by means of scope and/or filter. If no criterion is given, the collection will contain all objects of the database table. There can be also empty collections. If a collection is created then the objects of the same type can be added to the collection, removed or updated. To get a data object from the collection, the ID of the data object can be used. If the ID is unknown, the collection can be iterated in order to find the desired object. One of the important properties of collections is that the number of data objects that the collection contains is unknown. It is not always clear how many objects the collection contains and it entirely depends on the implementation of the collection interface. The following diagram depicts the *ICS_Collection* interface.

Figure 3.8: *ICS_Collection*

The relation between ObjectLogic, Filed, Collection and Data Objects is shown in the figure 3.9.

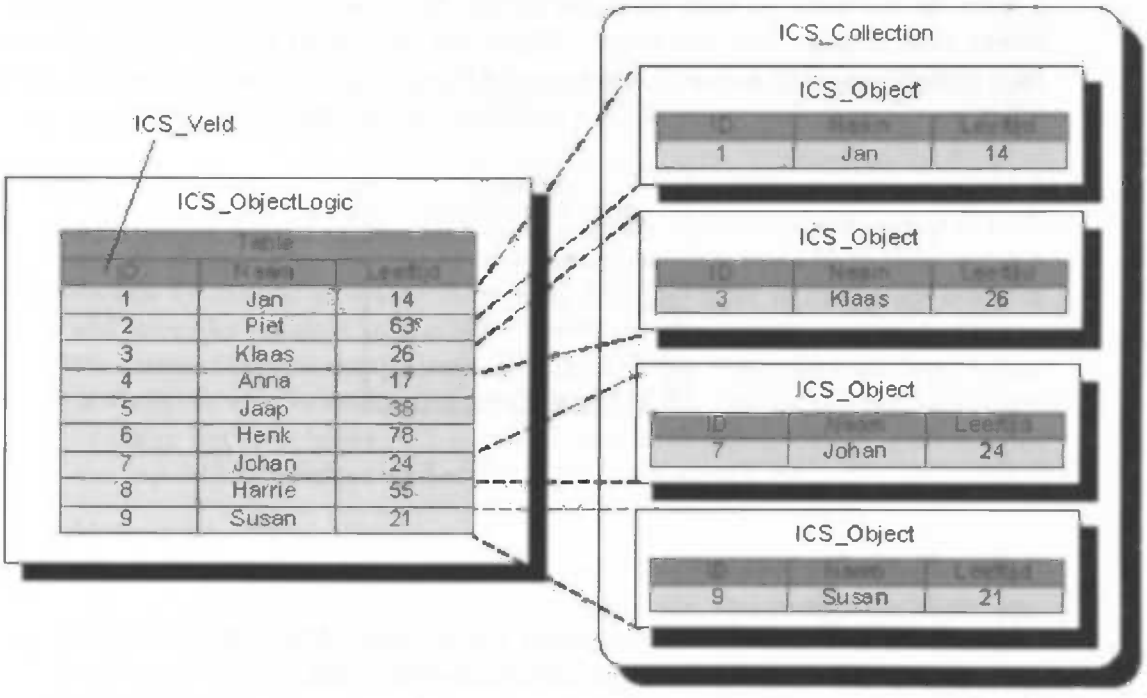


Figure 3.9: Relations between *ICS_Veld*, *ICS_ObjectLogic*, *ICS_Collection* and *ICS_Object*

3.2.6 Scope

Scope specifies the sorting criteria of data in collections. It is possible to define low and high bounds to which the certain fields of the objects must satisfy. The scope can be only created on indexed database fields. It can be created from Ob-

jectLogic or from data object collections and it exposes the interface *ICS_Scope*. The *ICS_Scope* interface is shown in the following diagram.

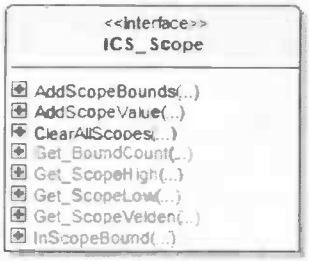


Figure 3.10: *ICS_Scope*

3.2.7 Filter

A filter, also called Filterexpressie, has no sorting criteria. In contrast to scope, it gives much more freedom. Several filters on different fields can be combined with operators such as "larger than", "smaller than", "contains", etc. It is also possible to create sub-filters and combine these in a tree structure. This way, every desired selection from the database can be realized. The fields which are used in a filter do not need to have an index in the database. However, the filters are slower than scopes. Just like scopes, filters can be created from ObjectLogics or from collections. The exposed interface of filters is *ICS_FilterExpression*. The following diagram depicts the *ICS_FilterExpression* interface.

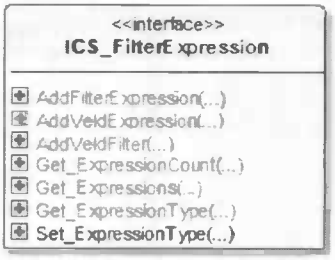


Figure 3.11: *ICS_FilterExpression*

3.3 Key Drivers

The architecture of CS-ECIS is designed around many functional and quality requirements. All these requirements have been influencing the architecture during the design process. Some of these requirements or combination of them have higher priority than others. They are standing high on the top-priority list of the requirements. These requirements can be identified as architectural drivers that are key-drivers of the design. These key-drivers are:

- Extendibility and Adaptability
  The architecture is highly extendible. By implementing new modules and services, it can be easy extended with new functionalities. Because of the layered design and modularity of the system, it can be easy adapted to

changes in requirements as required by evolving business needs and industry standards. For example, the hospital has deployed a new database back-end and ECIS has to be adapted in order to use it.

- Modifiability
  Most of the entities in the software communicate with each other through well defined interfaces (contracts). This interface based design ensures that when a modification occurs in an entity, it remains local and will not propagate behind its interface. Every other entity in the system will remain unaffected by the modification. This makes the whole architecture less sensitive to local modifications.

- Usability
  In order to improve the usability for the majority of CS-ECIS customers, ChipSoft has the overall appearance of the application changed from the standard way. These changes have affected the menu structure, the toolbar User Interface, etc. and finally the overall window layout. While ChipSoft has own GUI style-guides for the target platform, they take care of familiarity of users to this interface and provide predictable and consistent GUI. More than 100 proprietary custom GUI design components ensure high flexibility, high robustness, high customizability, and high learnability of ECIS GUI concepts and features.

Achieving these requirements in CS-ECIS is supported by architectural patterns and architectural decisions. These are:

- Layered design
  The architecture of ECIS is divided into three layers, stacked on top of each other. These layers are: presentation, business and data. Each layer focuses on one functional problem. Each layer uses the layer immediately below it and provides services to the layer above it. The layered design facilitates the reuse of system components, e.g. the services reuse the data-layer below of them. It also facilitates the modularity of the system by minimizing the coupling between layers, i.e. a layer knows little about other layers behind the exposed interface.

- Modularity
  The entire functionality of ECIS is spitted up and assigned to different modules. Each module is developed independently and design decisions are, taken in different modules, do not interference. Modularity of ECIS makes the complexity of the entire system manageable, enables parallel and independent development process and ensures easy integration.

- Reuse of components
  One of primary goals of the designed architecture is making reuse of the parts of the system. The COM based architecture makes it possible to be able to reuse the different parts of the application. For example, the modules can reuse services that belong to other modules. The reuse gives huge advantage during development process. It decreases the development effort and results in shorter development time.

3.4 Industrial Relevance of the Project

Distributed client-server applications with powerful database servers as back-end are increasingly deployed in industry as information systems in banking areas, in health care areas, in customer service areas etc. The strong current growth of the market of information systems, which is facilitated and supported by ubiquity computing, is predicted to accelerate in the next years. It is clear that database technologies and client-server applications will play a major role in the future information systems. Therefore, the industrial relevance of this project is high, and many useful results are expected. Many applications foreseen in the information system areas as well as the other areas in Information Technologies will benefit from robust and flexible data change notification systems. For example, an airline reservation system includes the central computer that contains flight schedules, passenger reservations, seat assignments, etc. Currently, travel customers are typically notified of airline schedule changes or other information via a public address system and airport monitors. The customers of airlines who are paid for ticket desire increased level of customer service. They desire more reliable, faster and better methods of receiving notifications of airline schedule information which affects their travel. Using data change notification system can help airline companies to increase customer service of travelers. Data change notification service will notify Airline customers about changes in airline information which affects their travel plans as well as other type of information (e.g. a gate change, alternate flight options, ticket overbooking, etc.). These notifications can occur via standard communication devices commonly carried by travels such as pagers, cellular phones, mobile computers (e-mail, web, etc.), PDA's etc. Another example is a bank computer system. The computer contains account balances and other information respecting the bank customers. Tellers in various branches of the bank must have access to this information to process deposits, withdrawals and other transactions. It will be apparent that when a several users simultaneously working with the database two or more of them may attempt to alter the same item of data at the same time. The system should ensure that balances shown to each party are in synch with each other. An absolute consistency is required in such applications that demands the freshest data. Data change notification service can help to maintain data consistency across these applications. Another examples; in hospitals could be a data change notification service used in blood bank databases: "Notify if the amount of any blood type falls below a certain threshold". In stock exchange companies the data change notification service can monitor stock data and notify clients if the stock price below certain value which investors think should consider purchasing the stock. To further reinforce the industrial relevance of data change notification technologies and this project in particular, several major industrial database product developers, such are Microsoft, Oracle and IBM are highly interested in data change notification services. Their last developed database products have integrated database facilities to support data change detection and notification mechanisms.

Chapter 4

Background Theory and Problem analysis

In the following sections, I present and analyze some basic theory related to the problem and required to understand the in's and out's of information change notification concepts and problems that can arise. Here presented theory will be later, during the design process, applied to the architecture of the notification service. In the section 4.1 are the basis technologies that could be used for notification facility discussed. In the section 4.2, two categories of applications, waiting and non-waiting applications, are defined and ECIS application categorized according to these definitions. The section 4.3 discusses and compares two different mechanisms for notifications (push and pull). The section 4.4 gets in the reliability and correctness problems that can arise. The sections 4.5-4.8 attend to the issues such as types of notifications and their propagation, duplicated, wanted and unwanted notifications.

4.1 Base Technologies for the Notification Facility

In this section, three different technologies that can be used as a base of the notification facility are discussed. The first two technologies are based on tightly integration of notification facility with databases. Third approach uses an external message broker to serve as a notification service. At last, a comparison of these three approaches is given.

4.1.1 Passive Databases

In traditional database systems, data is created, retrieved, modified and deleted, in response to requests issued by applications or users. These databases systems refer as passive databases. The passive databases only execute queries or transactions explicitly submitted by the user or an application. For many applications, however, it is important to monitor situation of interest, and to trigger a timely response when the situations occur. For example, the system needs to monitor the occupancy of surgery rooms in a hospital, so if a new room is reserved, the staff will be informed about. This behavior could be implemented over a passive database system as follows. In order to be informed about changes in such passive database, users can issue a permanent (continuous) query and they will be notified whenever

data matches the query. This class of queries, continuous queries, is similar to conventional database queries, except that they are issued once and henceforth run "continually" over the database. As additions to the database result in new query matches, the new results are returned to the user or application that issued the query. For its storage, the continuous queries (CQ) use a relational database that supports SQL. A straightforward method of implementing a continuous query over such database is to periodically execute the query, say once every hour. The outline of the basic algorithm is shown below.

```
FOREVER DO
    Execute Query Q
    Return results to user
    Sleep for some period of time.
ENDLOOP
```

This approach has a number of disadvantages:

- *Nondeterministic results.* The records selected by the query depend on when that query is executed. A query that is executed every hour on the hour may produce a different set of results than the same query executed once per day or even every hour on the half hour. This means that two users with the exact same continuous query could be presented with a different set of results.
- *Duplicates.* Each time the query is executed the user will see all records selected by the query, old as well as new. Moreover, the set of records returned by the query will increase steadily over time. In practice, users are only interested in the records matching the continuous query that have not been previously returned.
- *Inefficiency.* Executing the same query over and over again is too expensive. Just as the size of the query result set increases over time, so does the execution cost. Ideally, the cost of executing the continuous query should be a function of the amount of new data, and not dependent on the size of the whole database.

Because of these disadvantages, the passive database pattern approach cannot be used effectively to model requirements that involve monitoring and reacting timely to particular situations in multi-user database environment, such as informing the doctor when a new appointment in the doctor's agenda created by an assistant.

4.1.2 Active Databases

An active database system, in contrast to passive databases, is a database system that monitors situation of interest, and when they occur, triggers an appropriate response in a timely manner. Active DBMSs extend passive DBMSs with the possibility of specifying reactive behavior. The desired behavior is expressed in production rules (also called event-condition-action rules) which can be defined and stored in the database. Such behavior allows monitoring for and reacting to specific database circumstances at the DBMS itself. This has benefit that the

rules can be shared by many applications, and the database can optimize their implementation. An active database built on top of a passive DBMS specifies a knowledge model and an execution model. The knowledge model defines the database reactive behavior, generally in terms of event-condition-action (ECA) rules. The event part of the rule defines the situation that triggers the rule, the condition evaluates the context in which the event takes place, and the action formulates the task to execute after the rule has been triggered and its condition validated. Typically an ECA rule is of the form:

```
on event
if condition
then action
```

This allows rules to be triggered by events such as database operations. When the triggering events occur, the condition is evaluated against the database. If the condition is satisfied, the action is executed. The trigger mechanisms in commercial relational database systems such as Oracle, MS SQL, IBM DB2, etc. process active ECA rules on low level operations such as insert, delete, update, etc. These mechanisms are based on exact trigger conditions and actions. The implementation of this kind of database side capabilities is often proprietary to the database manufacturer. For an application such as CS-ECIS that uses different types of databases as backend storage, it will be difficult and too expensive to develop and maintain these proprietary features for each type of database. Moreover, using the trigger mechanism as a base for a notification facility can lead to heavy database load. For example, If the number of S(T) interested clients, with T as an abbreviation for 'table', have registered for modifications on the given table T, and the number of N(T) modifications happen per second on this table, than S(T)N(T) triggers have to be run per second for table T. In practice, this number is even larger because more than one table will be of interest for clients. If there are 10 modifications per second and 100 clients interested in, 1000 triggers will run on a single table. This is out of scope of current database technology. So, an active database technology as a base for notification facility for CS-ECIS seems unsatisfactory as being too difficult and expensive to realize, and to slow in performance.

4.1.3 Publish/Subscribe Mechanism with Message Brokers

Third approach, I want to consider, is to design a notification mechanism using an external message broker. It can help to meet the challenge by providing efficient, scalable, many-to-many, push-based delivery of messages to the clients within the system. In general a message broker is a pattern which is applied in a situation where messages are often, and sometimes periodically, sent from a plurality of message sources m to a plurality of message sinks n in m/n relationship. Special cases of $1 : m$ and $1 : n$ occur as well. The message brokers 'share' messages in a way that a source being any kind of application needs to transmit only one message and the broker delivers one of many versions of the message to one or more sinks which are applications as well.
More particularly the message broker is like a hub where messages are streaming in and are streaming out. The messages which are streaming into the broker are

referred to as being published. Messages which are streaming out of the broker are referred to as being subscribed. A subscription request specifies the subset of all incoming messages a particular application is interested in, and the format in which it has to be presented to the subscriber. For example, various sources in a hospital might publish data about the agenda of a doctor i.e. data is send to message broker. Each data source might use a different format to transfer data. A subscriber might have register to all messages about the agenda or only the messages of the current day, and may request its delivery in XML or WML format.
The message brokers typically deal with messages that are being published and the subscribers would like to know about. The published messages could contain any information relevant to subscribers including information about data changes occured in a database. When analyzing the behavior of publish/subscribe mechanism using a database oriented view, the following observation can be made: A subscription request is similar to a query. Based on what has been specified in the subscription request, messages are filtered out and are delivered to its recipient. However, there is a fundamental difference between a query and a subscription. The subscription does not operate on all messages streaming to the message broker like a query but only on a single message at time, namely the message that has just been published to the message broker.

4.1.4 Comparison of Active, Passive Databases and External Message Brokers

In this section the above described technologies are compared; thereafter the right technology that satisfies our needs will be chosen. The comparison is conducted using the following three criteria: efficiency, maintainability and flexibility.
Efficiency is the degree to which a system or a component performs its designated functions with minimum consumption of resources (CPU, Memory, I/O, Peripherals, Networks etc.). In the matter of efficiency the passive and active databases are far from satisfactory. In case of passive databases, as discussed in the former section, applications would poll the database which can lead to database and network overload; or to critical situation when a change is not detected in a timely manner. In case of active databases, where triggers used to detect changes, many instances of ECIS application can be interested in the same situation (e.g. a new record in a doctor's agenda) which will cause execution of many triggers simultaneously (or too long execution of one trigger) and can lead to the database overloading. In contrast, the message brokers are free from abovementioned shortcoming of passive and active databases. The message brokers can be notified about changes by clients and they will push notifications to subscribers in a timely manner keeping the load of the network and of the database at minimum. This way, the message brokers provides more efficiency for notification purposes than active/passive databases.
Maintainability is defined as the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to changed environment. As stated in above chapters, the CS-ECIS system supports a number of databases as backend storage. The implementation of triggers in active databases is often proprietary to the database manufacturer and for such application as CS-ECIS that uses different types of databases as

backend storage, it will be difficult and too expensive to develop and maintain these proprietary features for each type of database. In case of passive databases, maintainability is high because the semantic of condition checking of situation of interest has to be embedded into ECIS client that updates the database and it can be maintained independently from the backend database. However this is a poor approach from the software engineering perspective. In case of an external message broker the maintainability is high because the broker is independent from the backend database and from the clients. This allows different database servers and different versions of clients to work at the same time with the broker.
Flexibility is the ease with which a system or a component can be modified for use in applications or environments other than those for which it was specifically designed. It is clear that adaptation of notification systems based on active/passive databases to work in new environment can be very hard. For example, if ChipSoft ever decide to employ object oriented databases as backend storage, then all kinds of client software (PC, Web, PDA, etc.) should be redesigned. Using brokers as notification base does not require modifying any client software because the broker adds some kind of abstraction level between the database and the client. By only modifying the broker, the system can be adapted to new environments.
The table 4.1 provides the summary comparison of the active/passive databases and external message brokers as the base of the notification facility using efficiency, maintainability and flexibility as the comparison criteria.

	Efficiency	Maintainability	Flexibility
Passive databases	Low	High	Low
Active databases	Low	Low	Low
External message broker	High	High	High

Table 4.1: Comparison of the active/passive databases and external message brokers

From this comparison of three possible base technologies for the notification facility becomes clear that the external message broker can better satisfy our needs than solutions based on active/passive databases.

4.2 Waiting and Non-Waiting Applications

Applications that use some kind of notification facility can be divided into two classes: waiting and non-waiting. In order to design a proper architecture for notification facility we have to understand requirements concerning these classes of application and we have to classify CS-ECIS client according to these two groups. Waiting applications refer to those applications that, while waiting to be notified of a change, do not require any access or modifications of data on the database. The system therefore does not have to provide access to data during the period of time in which the application is waiting for a notification. In CS-ECIS environment, an application can act as a waiting application if it does not need any data from the database to operate on it (e.g. an application on a mobile phone device that is only interested in notifications of new items in a agenda). In this case, the

client does not require access to any pieces of data other than the data specified in the notification, for which it awaits modifications. The impact of implementation of notifications for waiting applications on the complexity of the system would be minimal, because there are no potential concurrency issues that need to be addressed. If the application is not reading or writing any other piece of data on the database, then there is no danger of operating on stale or invalid data. Notifications are essential for these class of applications, as they will proceed only upon receiving a notification. The CS-ECIS client application clearly falls out of this category of applications because it needs to access and to operate on data in the database while waiting for notifications.
Non-waiting applications encompass those applications that actively read and write data on the database, even while potentially waiting for a notification of an update to a member of its data set. Such applications proceed without waiting for notifications, and do not require them for the continuity of the application. CS-ECIS client application is an example of a non-waiting application.

In non-waiting applications, one of benefits of change notifications is that the applications can better ensure that data being used in a transaction is fresh, and the result of the transaction is more likely to be valid. In such cases, change notification can potentially eliminate the need of unnecessary transaction aborts and the need of polling queries by providing updated and valid copies of data. If the user is notified of a data modification before attempting to modify it, time and resources will not be wasted on performing invalid transactions and on keeping data up-to-date by means of polling the database. This way, the overall load in the system is reduced.

4.3 Push and Poll Notifications

One possible approach to get notifications is to have the client poll the system (the notification service or the database) to check whether any new notification are available for it. If the application periodically polls for notifications it is interested in, the data, at the time immediately before the polling call is made, is guaranteed to be no fresher than the periodicity of the polling. For example, if an application polls the server for notifications every 30 seconds, then data that the application holds could be as old as 30 seconds by the time the application had access to the data (disregarding the network latency). Periodically polling the system for notifications in many cases can be inadequate, if not done frequently enough then data of the client become stale, but if done too frequently then unnecessary processing overhead is incurred. Moreover, repeatedly making calls over a network connection to inquire about the changes could potentially tie up the network. Such network calls will become even more costly in the scenario where the client works off-line and must repeatedly establish a connection to the database or to the notification service. Although the application may be able to make general estimation on frequency of polling, it has no means of precisely determining the exact time the value is changed. Alternatively, having the system notify the waiting client application upon change of time-to-arrival requires only one network call at the exact time that data is actually modified. Pushing the notifications to the client provides two benefits over polling: preservation of network bandwidth and system resources as well as providing new data at time closer to that at which the

data was changed. While push-notifications may exist as an alternative to polling in situations where conservation of network bandwidth is desirable, situations in which this is not an issue pushing can be used in conduction with polling to improve the reliability of the system. If pushing of notifications is used with the polling, we will see an improvement on the latency of getting notifications. The data will be guaranteed to be at least as fresh as the polling would provide, and in all likelihood fresher than polling. In case that the push-notification is successfully transferred to the application via the network, the application would have access to the new updated data faster than through simple polling. If the push-notification fails, then the underlying polling will ensure that data will be received in time no slower than the simple polling. Clearly, the push-notification allows the application to receive an updated data notification in a shorter amount of time while the poll-notification mechanism ensures high reliability of the system.

In case of polling in ECIS environment, the notification server can poll the database to get the changed data and the clients can poll the notification server to get notifications. Such design can help us to reduce the overload of the database server but it does not scale as well. If the number of clients increases polling the notification server can become a bottleneck on the network. The better solution would be pushing the notifications to the clients. By notifying the client application of a data modification, the overhead cost of polling the database or notification server to refresh the data will no longer have to be undertaken by the client application, nor will the network and the database server be burdened by a potentially heavy load of polling queries. This way, we can reduce overload of the network and the database. Moreover, the pushing will reduce the latency at which data is refreshed and therefore the usability of the ECIS client will be improved too. The combination of poll and push notifications would provide the highest reliability of the system with decreased average latency through which data consistency is established but because of the complexity of the implementation and the limited time of the project it is rejected as practically not feasible.

4.4 Reliability and Correctness

Implementing notification facility decreases the latency of changed data propagating throughout the system, but it also introduces complexity issues that must be deal with to ensure reliability and data consistency. The reliability of the notification message transmission and handling protocol necessitate a definition of correct behavior that will maintain the system data consistency. The following subsections examine this issue.

4.4.1 Reliability of Notifications

The system sends notification messages to applications via network connections that are assumed to be inherently unreliable. In other words, there is no guarantee that a single notification message sent to an application will be received, due to unplanned occurrences such as network partitions, network device failures, etc. Beside this, the clients and the notification server can be considered unreliable as well (because of software failures, hardware crashes, etc). In order to keep the reliability of the notification system high, there are few issues that we have

to deal with. For example, the notification server tries to send a notification to the client but the client fails to respond to the server. There can be two causes of the failure: network failure or the crash of the client. In case of a network failure, the notification server has knowledge of which notification messages may not have reached the clients. It should try to resend the notification messages. After recovery from the network failure, the client will receive the notification and will continue normal operation. In case of client crash, the notification server after few times attempting to resend the notification may consider the subscription of the client invalid, and it may remove the client from the subscriptions list. The notification server has to ensure that the subscription is invalid before removing it; otherwise the client will "forever" stay waiting for notifications without receiving any. In other situation, the notification server may crash and the clients have no knowledge about the crash. The clients, being unaware of unavailability of the notification server, will stay waiting for notifications while they will not get any. Such cases must be handled correctly and very carefully, otherwise the reliability of the system will be low and unacceptable. One possible solution to increase the reliability of the system is to make the clients and the notification server aware of each other's availability. It can be done, for example, by implementing "is alive" communication messages. The clients will send "is alive" messages to the server. If the notification server does not receive such messages from the subscribed client during determined period of time then the subscription of the client will be considered as invalid and it will be removed from subscribers list. In case that a client sends "is alive" message to the notification server and gets no response from it, the client will consider the notification server as unavailable and it will take appropriate actions to handle the situation (e.g. switch to poll-database mode, etc).

4.4.2 Consistency and Invalidation

In distributed systems, there are several situations where the inconsistency problems may arise. For example, a global resource such as a database may be accessed from many users at the same time. It is not easy to maintain data consistency by sending a notification to all distributed users running on different locations in a synchronized manner whenever the global shared resource changes. If few users miss the change notification or receive the notification asynchronously as it is often the case, data consistency between users and the database may fail (e.g. user's data become stale). Also, if database resource utilization is not optimized and multiple users get notification about the changed resource almost the same time, the database which shares the resource may become a performance bottleneck because all of the clients may attempt to access this resource at the same time. Additionally, if a global resource changes while the notification facility is temporarily unavailable, it is not possible for the users to know the current level of information relating to the shared resource and consistency will be almost always lost. It would be desirable in CS-ECIS application to have some kind of mechanism in order to handle such situations. A special attention must be given situations where due to concurrency issues between publishers and subscribers data inconsistency problems may arise. Figure 4.1 shows such situation. While the client 1 is fetching data from the database, client 2 updates a record in the database.

After that, the client 2 notifies the notification service about the change. Once the notification service receives the "Notify" message from client 2, it begins iterating through the list of the subscribers and sending notifications to each of them. Just after the iteration process ends, the client 1 sends a request for a subscription to the notification service. If the subscription successes, the client 1 will never know about the just missed notification and will continue to work with stale data.

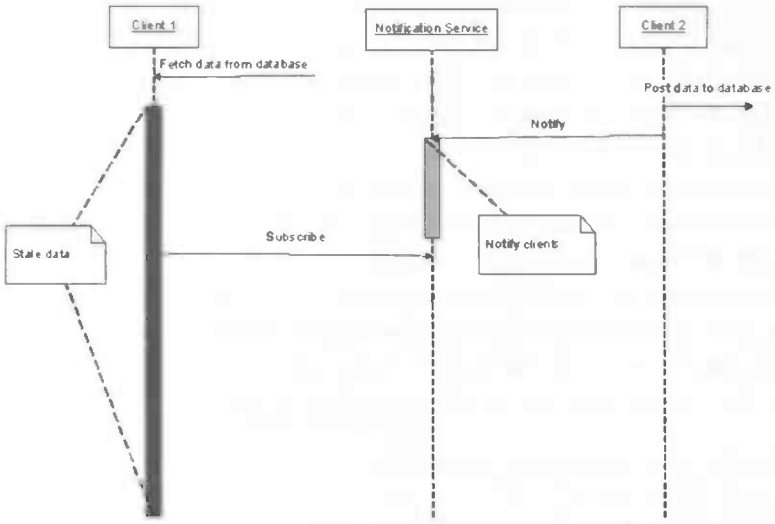


Figure 4.1: Data inconsistency due to a concurrency issue

Absolute and Relative Consistency

There are essentially two types of data consistency enforcement among objects of distributed systems: absolute and relative. Absolute consistency requires that the copies of the objects held by the users (the applications in our case) are consistent with the centralized copies stored in the database at all times. Relative consistency, on the other hand, demands that each of the members of the set of objects contained by one application is consistent with one another at a specific time t. An application requiring relative consistency does not place emphasis on the freshness of its data. The absolute consistency is required in applications that demand the freshest data. In CS-EICS client-server environment, it will be difficult to provide an absolute data consistency within this project. Because of the restrictions of the existing framework, there is no easy way to integrate appropriate facilities (e.g. global locking, global transactions awareness by clients, etc.) into the middleware without significant architectural changes which are out of boundary of the purpose of this project. The notification facility will aim to provide a globally relative consistent environment, in which applications are locally aware of the global state of shared resources. The relative data consistency is sufficient in this case and acceptable if the period of inconsistency of data sets is smaller or equal to the original refresh period of ECIS client.

Invalidation

Invalidation is a means to preserve consistency within a distributed system. Invalidation refers to the process of rendering a subset of data objects as being stale, and therefore unusable. Invalidation is used to maintain the correctness and data consistency of the system. It often results in data collections being refreshed. Figure 4.2 shows the invalidation mechanism involving the central notification service and two clients that share the common set of data. The key point of this invalidation algorithm is early subscriptions of the clients. Before fetching the data from the database, the client subscribes for the change event at the notification service by providing the attribute filter of the desired dataset. This can be done because the content of the attribute filter is known at the subscription time. Client 2 modifies a record during the transaction and commits the changes to the central database server. The database updates its own copy of the record. Note that this update introduces inconsistency among the shared copies of the record. To remedy this, the client 2 publishes the notification message to the notification server. Once the notification sever receives the "Notify" message, it begins to send invalidation messages to all subscribers which have stale data. Client 1 can receive the invalidation message for the record with an *id* at two points of time:

- Before the fetch operation completes
- After the fetch operation completes

If Client 1 receives the invalidation message after the fetch operation is finished then this message will be considered as a normal invalidation message and the application will perform refresh of the record specified in the invalidation message. If Client 1 receives the invalidation message before the data fetching is completed then the application will consider the retrieved data collection as invalid and it will perform a full refresh of it. After the refresh, data consistency will be restored to the system. Because the rate of inserts/update/delete operations in CS-ECIS environment is quite low (about 15 op./sec) and not all those operations affect the same dataset, the chance that the full refresh of data collections will occur is very small.

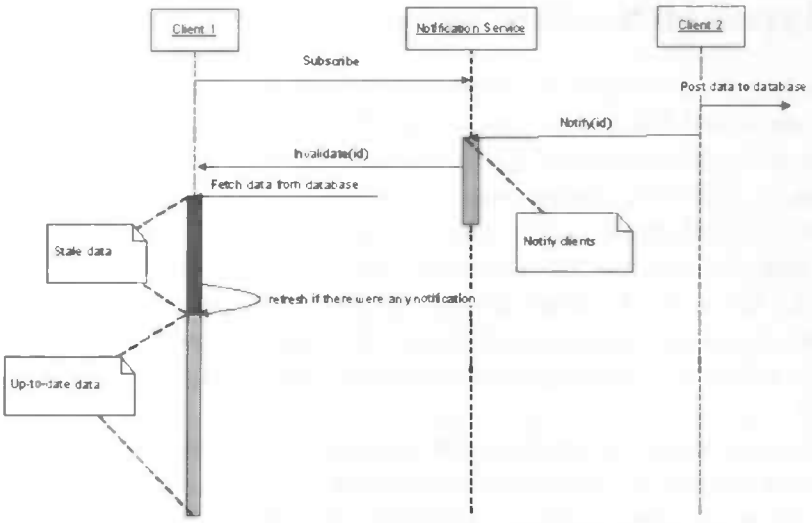


Figure 4.2: Data invalidation mechanism

4.5 Notification Propagation

There are three levels in the client to which a notification message may propagate: the application cache, the application, or the user. The application and its usage determine how far notifications must propagate for correct behavior. Generally, the more frequent the notifications are expected, the less propagation is necessary. In some cases when the rate of data changes is extremely fast, such as the database conversions modules of CS-ECIS, the application can assume that data is constantly changing, and there is no need to be notified every time. In such cases the high rate of notifications would be burdensome and not particularly useful. The messages that propagate to only the application cache result in cache updates of data objects without the application or user having any knowledge of the changes. The use of the data cache in CS-ECIS is optional. The developers may choose to use it or not. In some modules, the data cache is used for data collections that are not changing very frequently and these collections use the automatic refresh facility of CS-ECIS client. Using the notification mechanism for cache updates will reduce the number of refresh-queries sent to the database. A number of modules in CS-ECIS use data collections with no local cache. Such data collections require application-level notifications. Propagating the notification message all the way to the user consists of the application somehow conveying an update through its GUI interface to the user. This is needed when changes may impact the user interaction with the application. One example of the application that needs to notify its user is the agenda module of CS-ECIS. If an assistant adds a meeting to doctor's calendar, the agenda module of the doctor will receive the notification and it should have some mechanism of notifying the doctor of this change.

4.6 Types of Notifications

There are two general types of notification techniques in distributed environments: message-based and flag-based. In message-based notifications the system notifies its clients of modified objects. Message-based notification is further divided into immediate or deferred. Immediate message-based notification requires that the clients are notified immediately after the changes to an object are committed, whereas deferred notification allows for the notification to take place at a time specified by the user. In flag-based notification, the system simply updates the data structures that it maintains (e.g. sets the flag dirty of the object to true), so that users will have knowledge of the changes only when they specifically access the object.
In ECIS client-server environment, the changes in the database that are relevant to the client should be immediately reflected on GUI of the client, so the user can take the appropriate actions. Therefore, in this project we will focus on the implementation of message-based immediate notifications. Immediate notifications force the application to receive the notifications as soon as they are received from the client that has committed the changes. This could be accomplished by having the application assign a single processor thread whose sole purpose is to wait for notifications allowing the application to be aware immediately upon receiving the message.

4.7 Duplicate Notifications

Duplicate notifications can occur when clients consecutively changes the same data in short amount of time. There are two cases concerning duplicate notifications that must be considered. In the first case, if after consecutively changes of the same data by the clients, the subscribers have not been notified yet, the notification server would detect the duplicate notifications for the same data and would send only one notification to the subscribers. In the second case, if during consecutively changes of the same data by the clients, all subscribers or some of them have been already notified, these subscribers will receive the second notification for the same dataset. It must be noted that the duplicate notification messages do not have the potential to negatively affect the correctness of the system. This is because the client that receives a duplicate notification can simply refresh the data specified in the notification message more than once. Clearly, this method is inefficient as data is needlessly refreshed more than needed but it does not endanger the correctness of the system. In some cases the client application can receive the notification of its own change of data. For example, the client application changes the data and at the same time interested in changes of the same data. This kind of notification messages can be safely ignored by the application because the stale data can be updated locally without query the database.

4.8 Wanted and Unwanted Notifications

It is possible that an application desires notification messages only during a limited period of time. To ensure that the system does not send notification messages

at an unwanted time the application must be aware of when it is interested in notifications and ignore messages when it is not. It is necessary to provide a means for the application to not only request or subscribe for notifications, but also to de-request or unsubscribe for notifications. One method of achieving this is to explicitly have the application request the notification of specific data. When the application is no longer interested in receiving notifications, it can make a de-request call for the earlier specified data. So the notification messages are not sent and the application does not expect any notifications once the de-request is made. Sometimes the client applications would need to refine the scope of the dataset for which notifications are requested without de-requesting and requesting again. Alternatively, the system may allow a request for notification to expire (like in MS SQL server 2005). Applications may only want to receive notifications regarding a certain dataset for a given period of time. The system can allow the applications to determine the lifespan of notifications that are not permanent. The irrelevant notifications will not continuously be sent to the application that no longer is looking for them. In this case, there is no need for a de-request message. It should be noted that in such scenarios, there is a chance that an unwanted notification messages may still be sent. It is the duty of the application to appropriately ignore these messages.

Chapter 5

Requirements

A requirement is a necessary attribute of a system, a statement that identifies a capability, characteristic, factor of a system in order for it to have a value and utility to a user. Requirements are important because they provide the basis for all of the development work that follows. In this section, based on the analysis of the problem and background theory concerning the problem domain in former sections, the initial requirements of the notification service are elicited. The system should satisfy to the following requirements shown in the table 5.1.

The use case scenarios that describe the sequence of software actions required to complete the specified functionality are provided from table 5.2 to 5.5

The identification of the real requirements requires an interactive and iterative requirement process, supported by effective practices, processes, techniques and tools. In this case, the elicited initial requirements with described use case scenarios provide sufficient basis to initiate the development of the notification service including system design, implementation, testing, deployment of the prototype etc. The identified initial requirements will be changed and/or refined during the lifetime of the project as the system evolves.

Entry	Requirement specification	Type	Priority	Use Case Scenario
1	The notification service shall run on a x86 platform using the Microsoft Windows 2000 Server OS	HR	essential	N/A
2	The notification service and the clients shall use an ethernet network to communicate with each other	HR	essential	N/A
3	The notification service shall be able to subscribe and unsubscribe clients for notifications	SFR	essential	1,2
4	The clients shall provide an attribute-filter of data in order to subscribe for notifications	SFR	essential	1
5	The notification service shall be able to refine the subscription by an attribute-filter provided by the client	SFR	desired	N/A
6	The notification service shall send notifications to the clients according to their subscriptions	SFR	essential	3
7	The clients shall be able to receive the notifications on which they are subscribed	SFR	essential	4
8	The notification service shall notify the subscribers within 30 seconds	SNFR	essential	N/A
9	Up to 500 users may use the notification service without any noticeable degradation of performance	SNFR	essential	N/A
10	The Notification service shall authorize and authenticate the clients in order to allow them to access and use notification facilities	SNFR	essential	N/A
11	The clients shall fall back to the original refresh mode when the notification service is not available	SNFR	secondary	N/A
12	The notification service shall be 99% available for operational use in 24 hours a day, 365 days a year	SNFR	essential	N/A
13	The response time of the notification service to requests of the clients shall not take more than 1 second	SNFR	essential	N/A
14	The utilization of the processor, memory and network resources shall not be more than 80%, 80% and 100Kb/sec respectively	SNFR	essential	N/A

Table 5.1: The requirements

Use case Name: Request for subscription **Use case scenario:** 1
Description: The client sends a request for subscription by specifying an attribute-filter for the desired dataset to the notification service. The notification service satisfies the request by providing the client with a subscription ID.
Precondition(s): 1. The client's identity has been authenticated. 2. The client is authorized to use the notification facilities
Postcondition(s): Subscription is stored in the notification service
Normal Course: 1. Client specifies the attribute-filter for desired dataset. 2. Client make a subscription request by providing the specified attribute-filter. 3. Notification service verifies that the request is valid. 4. Notification service stores the subscription in the subscriptions list. 5. Notification service returns the subscription ID to the client.
Exception(s): The request of the client is invalid (at step 3) 1. Notification service returns 0 as subscription ID. 2. Notification service terminates the scenario.

Table 5.2: Use case scenario 1

Use case Name: Request for unsubscription **Use case scenario:** 2
Description: The client sends a request for unsubscription by specifying the subscription ID to the notification service. The notification service satisfies the request.
Precondition(s): 1. The client's identity has been authenticated. 2. The client is authorized to use the notification facilities
Postcondition(s): Client is unsubscribed.
Normal Course: 1. Client specifies the ID of subscription. 2. Client make a unsubscription request by providing the specified ID. 3. Notification service verifies that the request is valid. 4. Notification service removes the subscription from the subscriptions list. 5. Notification service returns "true" to the client.
Exception(s): The request of the client is invalid (at step 3) 1. Notification service returns "false" to the client. 2. Notification service terminates the scenario.

Table 5.3: Use case scenario 2

Use case Name: Notify the subscribers **Use case scenario:** 3
Description: The client sends a notification request to the notification service. The notification service receives the request and notifies the subscribers about it.
Precondition(s): 1. The client's identity has been authenticated. 2. The client is authorized to use the notification facilities
Postcondition(s): Subscribers are notified.
Normal Course: 1. Client specifies the ID of the changed data object. 2. Client sends a notification request by providing the specified ID. 3. Notification service verifies that the request is valid. 4. Notification service accepts the request by returning "true" to the client 5. Notification service notifies the subscribers which satisfies the notification request.
Exception(s): The request of the client is invalid (at step 3) 1. Notification service returns "false" to the client. 2. Notification service terminates the scenario.

Table 5.4: Use case scenario 3

Use case Name: Client receives a notification **Use case scenario:** 4
Description: The subscriber receives a notification from the notification service.
Precondition(s): 1. The client is subscribed in order to receive notifications.
Postcondition(s): Data specified in the notification is updated.
Normal Course: 1. Subscriber receives a notification from the notification service. 2. Subscriber extracts the ID of the object specified in the notification message. 3. Subscriber verifies that the collection of the specified object is valid. 4. Subscriber informs the client that the collection of the specified object is stale. 5. Client updates the invalid collection.
Exception(s): The collection of the specified in the notification message object is not valid (at step 3) 1. Subscriber unsubscribes the client. 2. Subscriber terminates the scenario.

Table 5.5: Use case scenario 4

Chapter 6

Design and Implementation

6.1 Design Decisions and Related Design Patterns

In the following sections are the architectural design decisions that were made during the development of this project discussed and the rationale behind of them is given. The design patterns related to the architecture of CS-ECIS and the notification service are explained.

6.1.1 DCOM

The CS-ECIS application has component based architecture. The application is entirely based on the COM technology of Microsoft and makes heavy use of COM-objects. It is logical to follow the current design guidelines of CS-ECIS and extend the existing architecture further with a notification facility. However, the notification facility should be able to communicate across computer boundaries which cannot be done with COM. Fortunately, Microsoft provides a solution that extends COM and enables programmers to create distributed COM (DCOM) applications. This means that clients can create objects in separate processes on remote computers. COM's distributed capabilities are based on an interprocess mechanism known as Remote Procedure Call (RPC). RPC is an industry standard that has matured on many different platforms. Microsoft's implementation is known as MS-RPC. Together COM and RPC have a symbiolic relationship. COM offers RPC an object-oriented feel. RPC offers COM the ability to serve up distributed objects. All the while, clients still invoke methods as usual, as if the objects were very close at hand. DCOM extends COM. It works on top of COM and uses the functionality provided by COM. It takes care of issues such as security, remote communication and deployment the components over a network. DCOM can be seen as a high-level network protocol that enables objects to work together across a network in a distributed environment. From this perspective, DCOM is a high-level network protocol because it is built on top of several layers of existing protocols (udp, tcp, http, etc.). These underlying protocols are transparent to the component developers and can be configured at deployment time.

Decision: DCOM is the right technology to extend CS-ECIS with a notification service. Using DCOM, we will take full advantage of the existing investment in CS-ECIS COM-based components, tools and knowledge. Also, we will take

full advantage of typed data over the network, an efficient and independent communication protocol, reliable communication, integrated security and late interface binding.

The rationale behind this design decision is that CS-ECIS is entirely based on COM and integrating it with DCOM should preserve the methodology of CS-ESIC. Figure 6.1 shows how the CS-ECIS client and the DCOM notification server will relate to each other.

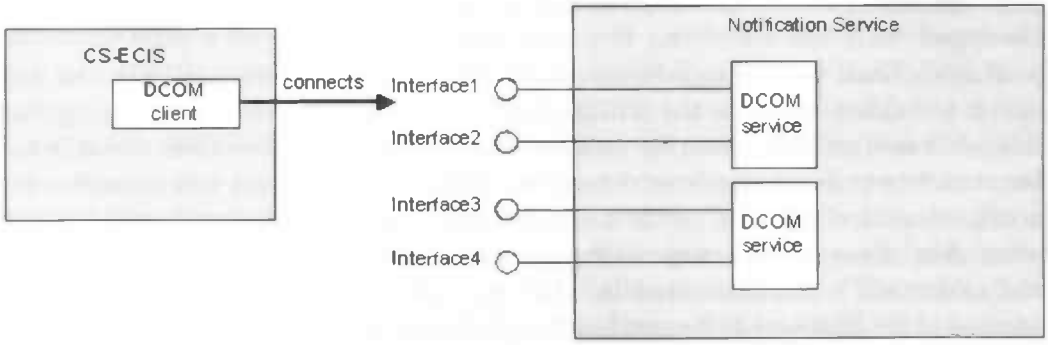


Figure 6.1: CS-ECIS client and the DCOM notification service

The client and the service code will be actually running in different processes on different machines. For the client application, it will be not much difference between using COM and DCOM. The client will connect to the interface exposed by DCOM server and call methods on it. DCOM will take care of establishing an efficient and reliable communications channel between these two parties.

6.1.2 Topic and Content Based Publishing/Subscription

The publish-subscribe paradigm is an important paradigm for asynchronous communication between entities in distributed environments. In this paradigm, as described in earlier sections, subscribers specify their interests in certain event conditions, and will be notified afterwards of any event fired by a publisher that matches their registered interests. This many-to-many data dissemination model removes for clients (publishers and subscribers) the need to know each other. All that is needed is that the definitions of events of potential interests are advertised before such events are published in the system. Publish-subscribe systems can be characterized into three broad types based on the expressiveness of the subscriptions they support. In topic-based and subject-based schemes, events are classified and labeled by publisher as belonging to one of a predefined set of subjects. Publishers generate events with respect to the topic or subject. Subscribers specify their interest in a particular topic, and receive all events published on that topic. Defining events only in terms of topic names is inflexible and requires subscribers to filter events belonging to general topics. Content-based publish-subscribe is a more general and powerful paradigm, in which subscribers have the added flexibility of choosing filtering criteria along multiple dimensions, using thresholds and conditions on the contents of the message, rather than being restricted to (or even requiring) pre-defined subject fields. Content-based publish-subscribe applications

present a unique challenge not only for efficient matching of events to subscriptions but also for efficient event delivery. In particular, content-based subscriptions can be highly diverse, and different events may satisfy the interests of widely varying groups of subscribers. In CS-ECIS environment are all data objects created by ObjectLogics. ObjectLogics belong to the application business logic tier of CS-ECIS. They are factories for creating data objects and object collections. They also map data-objects to corresponding tables on the database. Notice the difference between data and data-objects. As data I refer to not typed piece of information (e.g. content of a row of a database table). The data-objects, in contrast to data, are typed (they are not strong but weak typed) and belong to a particular ObjectLogic. Each ObjectLogic in the application is identified by a GUID. On one side it is convenient to use the GUID property of ObjectLogics as topic-name, like in topic based publish-subscribe notifications, because the subscribers are only interested for notifications of one data-type. This way subscribers can subscribe for notifications and only for notifications for data-objects of its own type. On the other side, there can be many notifications for data-objects of its own type which the subscriber is not interested in. For example, the subscriber in the agenda module of CS-ECIS could be notified for all changes in the agenda while he is only interested in changes for the current day. In this case the subscriber has to ignore superfluous notifications. This is not a very efficient solution. More effective solution would be a notification service with capabilities of filtering the notifications based on the content of the changed data. That means that subscribers should be able to specify a scope of notifications which they interested in and notification service should, based on provided scope, filter the notifications for each subscriber.

Decision: The notification service shall combine the topic-based and content-based publish/subscribe mechanisms to achieve high efficiency. The GUID of ObjectLogics will be used as topic names and XML representation of the filter of object-collections will be used as content based evaluation rules.

The rationale behind this decision to combine these two models is to create such publish-subscribe mechanism that embodies a communication model providing the necessary component decoupling, flexibility, expressiveness, and scalability.

6.1.3 Tightly and Loosely Coupled Events

Notifying interested parties that there have been changes to information is a fundamental issue of distributed computing. In the simple case, subscribers are provided with an interface on which a method could periodically be called to determine if changes are took place. Such a procedure, as discussed in previous chapters, is known as polling. While polling is a simple way to obtain notifications, there are some disadvantages to using this method. Polling wastes time for both the publisher and subscriber. The subscriber must send a method call to the publisher and the publisher must relay an answer to the subscriber even if there have been no notifications. Polling also involves a time lag between when data actually changed and when the subscriber polls for the notification of that change. For our purposes, a notification system should provide a way to initiate the notification process of interested parties when data changes. There are two design techniques that provide this facility: tightly (TCE) and loosely (LCE) coupled events. Coupling is a

term that describes the level of common knowledge necessary by a publisher and a subscriber in a distributed computing environment. In a tightly coupled environment, the programmer of one participant (e.g. the subscriber) must have detailed knowledge about the behavior, such as the method calls, messaging protocol, synchronous behavior, or message semantics, of the other participant (in this case, the publisher) in order to successfully complete the required interaction between the two pieces of software. Likewise, in a fully decoupled environment, the two participants need have no knowledge about each other in order to interact. And finally, in a loosely coupled environment, the two participants may have specific, but more limited knowledge about each other. In a tightly coupled environment, the publisher is provided with an interface on which it calls a method when a change occurs. The subscriber in a TCE system knows which publisher to request notification from and the interfaces that are exposed. During run time, the subscriber enlists itself with the publisher to receive events, and un-enlists itself when it is no longer interested in receiving the events. In order for this technique to work, the publisher and the subscriber need to agree upon a predefined interface to be used for communication (i.e. the callback interface). The subscriber provides the publisher with an object that implements this interface, and the publisher calls a method on it when something interesting happens. This bidirectional communication between the publisher and the subscriber is depicted in Figure 6.2.

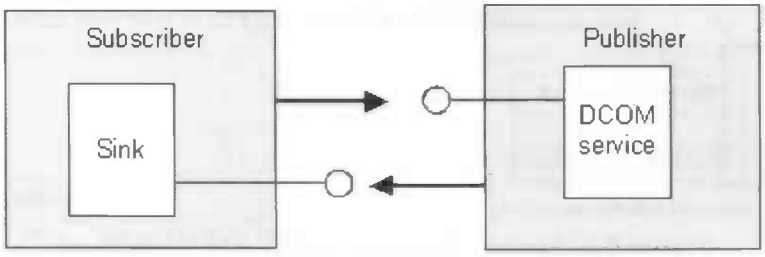


Figure 6.2: Tightly Coupled Events (TCE) pattern

The agreed-upon predefined interface is referred to as the source or the outgoing interface and the object the subscriber provides to the publisher is called the sink object. While the TCE technique is quite useful, it has some drawbacks:

- The lifetime of the publisher and the subscriber is tightly coupled. The subscriber has to be running and connected to the publisher to receive the events. In some cases this is not problem for subscribers which have no reason to receive events outside the lifetime of their publishers. However, in CS-ECIS environment the publishers are clients at the same time, forcing the subscribers to be connected to publishers at all times does not scale well.
- Under TCE, there is no mechanism in place to filter events. For instance, in the earlier agenda example, a subscriber ends up getting the appointment changes for all the agenda events even though the subscriber is interested in watching only the current day changes.

One way to address these problems is to bind the two entities at a higher level of abstraction than the predefined interfaces. Using this higher level binding

information, the publisher can connect to the subscriber at the time the event is published. The lifetime of the publisher and the subscriber are no longer tightly coupled. Likewise, a subscriber can still subscribe to an event even if there is no publisher running. Such a system is referred to as a loosely coupled event (LCE) system. The LCE architecture is shown in Figure 6.3. In this architecture, the publisher and the subscriber are decoupled by means of an intermediary object called the event class. An event class is a software component that contains the interfaces and methods used by a publisher to advertise events. An event is a single call to one of the interface methods from the event class. A subscriber implements the interfaces and the methods of the event class it wants to receive events from while the publisher calls a specific interface method to publish an event.

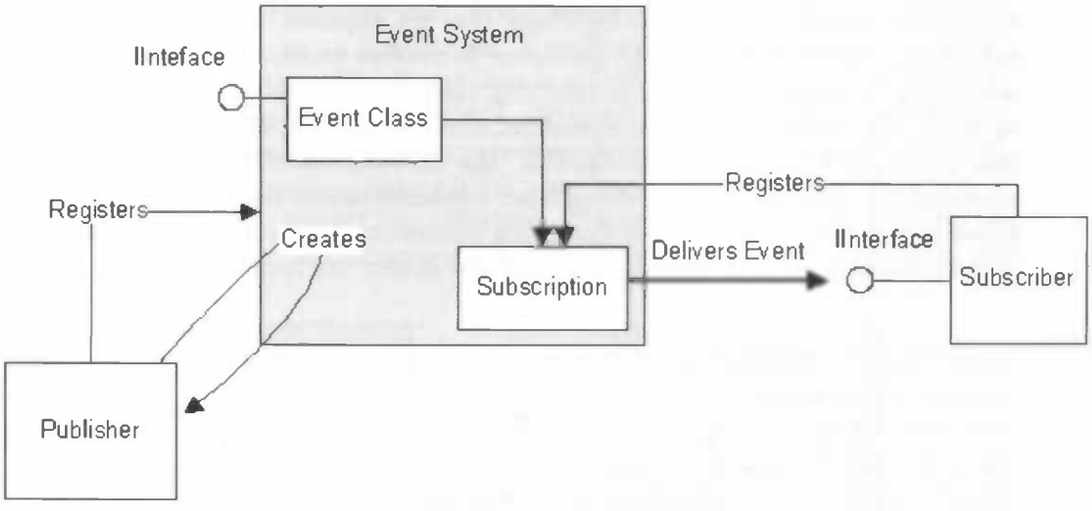


Figure 6.3: Loosely Coupled Events (LCE) pattern

Event classes are stored in a catalog, typically placed there by the publishers. The part of the catalog that stores event information is referred as event store. To indicate its desire to receive events, the subscriber registers a subscription with the event service. A subscription is a data structure that provides the event service with information about the subscriber so that it can connect to the subscriber to deliver the event. Among other things, it specifies the event class and the specific interface or method within the event class the subscriber wants to receive calls from. Subscriptions can be registered either by the subscribers themselves or by some administrative programs. When registered, the subscription information is stored in the subscriptions catalog.

Decision: The notification server will use the Loosely Coupled Events pattern in order to decouple publishers and subscribers, store subscriptions and deliver the notifications to subscribers.

The rationale behind this design decision is that such decoupling will allow us to minimize the interdependency between objects that interact and to build more flexible, extensible and an easy modifiable architecture.

6.1.4 Strategy Design Pattern

In CS-ECIS database environment, there can be different classes of subscribers that require delivering change notifications in different ways. All the subscriber objects are basically the same, and differ only in their behavior. For example, CS-ECIS subscribers prefer to get push-notifications, internet subscribers desire to poll the notification server, mobile device subscribers have to get notifications by some other means, etc. It should be convenient to decouple the notification algorithms from the whole and encapsulate it in a separate class. More simply putting a subscriber object and its behavior separated into two different classes will allows us to switch the algorithm that we are using for notifications at any time. If we ever want to add, remove, or change any of the notification algorithms (e.g. add notification by SMTP), it is a much simpler task since each one is its own class.

Decision: The notification server will use the strategy pattern in order to decouple the notification algorithm from subscribers and encapsulate it into its own class: a strategy class.

The rationale behind this design decision is to define a family of notifications algorithms, encapsulate each one and make them interchangeable. This will allow the algorithm of notification to vary independently from client that use it. The applied strategy pattern is depicted in figure 6.4.

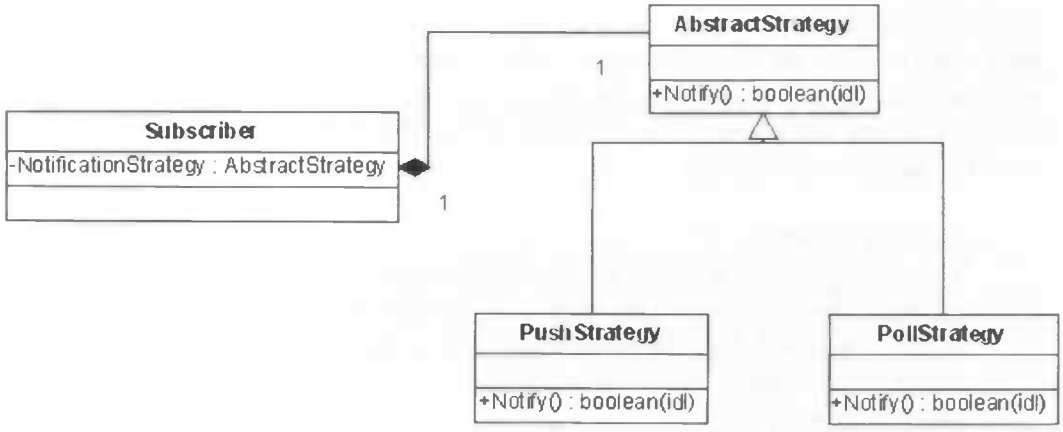


Figure 6.4: Applied strategy pattern

6.2 Architecture of Notification Service

The architecture is based on a distributed, content and topic based, publish/subscribe model. It is built on DCOM technology to provide scalable event dissemination and fault-tolerancy. The distributed event-based system implemented on top of the existing architecture of ECIS and consists of the following components: the client applications and the notification service. The notification service is the event broker. It can be seen as an application-level overlay network that performs event propagation using attribute-based filtering and notification algorithms. The client applications are event publishers and event subscribers. They

use the services provided by the notification service to communicate to each other using events. The subscribers are stored in a list maintained by the notification service. The notification service receives the calls made by publishers and directs these calls to the matched subscribers from the list without requiring the knowledge of the publisher.
The architecture enforces to use strong typed events. It means that the events have their own type and every published event in the system is an instance of a typed event. In order to publish an event, the publishers specify the event type using the GUID according to their notification needs. This GUID is used to filter the publications and subscriptions at runtime. Before publishing an event, the publisher creates the object of the associated event class by sending a request message to the notification service The interface of the created event object is returned to the publisher as a result of this request. The connection between the publisher and subscriber(s) is known as an EventClass. The EventClass matches and connects publishers and subscribers. The EventClass is a COM component that declares an interface used by the publisher to publish the event. The subscriber must implement this interface in order to be able to receive the event. EventClasses are stored on the notification server and their GUID's are known by publishers. An event is a single method in the EventClass interface. The publisher calls the methods on this interface to initiate the event and the subscriber receives these calls from the publisher.

The architecture is in the form of a client-server model and depicted in Figure 6.5. The notification service consists of a number of COM objects that expose their interfaces to the outside world. The client applications manage their subscription and notify about changes by making calls to these COM objects using the appropriate interfaces. The following is the descriptions of the COM objects depicted in the architecture:

- CSSubscription
  This COM object is used by the notification server to represent the subscriber. The client uses this object to manage its own subscription. For example, the client can via the presented interface change the attribute-filter, temporarily disable receiving notifications or permanently unsubscribe for notifications.

- CSNotificationEvent
  This COM object is a factory for creating of notifications objects. The client application passes the GUID of the desired notification event to this object and requests the notification server to create it. The client gets the interface of the created notification object. This way there can be different notification events created each with different interfaces. This ensures the extensibility of the system.

- CSDBNotification (EventClass)
  This is the notification event object. Each created notification event object operates on its own thread. In order to support notifications with scripting languages such as VB, JS, etc, the late binding on the interface of this object is used.

The definitions of the interfaces in the form of type library can be found in the appendix of this document.

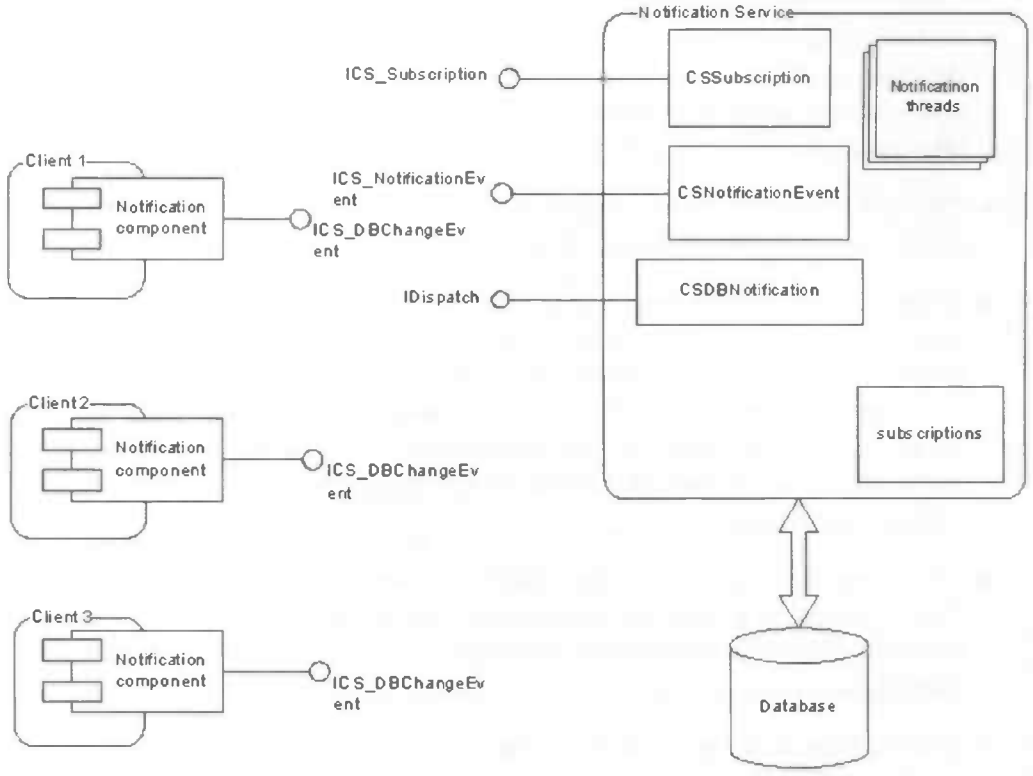


Figure 6.5: Architecture of the notification service

6.2.1 Logical View

The logical view of the notification service is depicted in the figure 6.6. The view consists of number of classes, interfaces and dependencies between them. Each class has own responsibility in the system and implements one or more interfaces. Below, the description and the responsibility of each class are given.

- IOwner is a helper interface to allow access to data of the owner (the object which implements it) of this interface.
- TDBChangeNotification is a class that implements *ICS_EventClass* and IOwner interfaceses. It is a COM object without a type library and depends on *ICS_Subscribers* interface. The responsibility of this class is receiving published change events from clients (publishers). This class is also responsible for creating threads for notifying the subscribers.
- TSubscribers is a class that implements *ICS_Subscribers* interface. It is an interfaced object that has basic reference-counting functionality which makes memory management task easy. The responsibility of this class is to store the subscribers. It also provides facilities to manage the stored subscribers. This class is thread safe and can be used simultaneously by many threads.

- TCSSubscription is the class that implements *ICS_Subscription* interface. The class represents the subscriber in the system. The responsibility of the class is to store data about subscribes (event name, cookie, etc.).

- CSDBNotificationThread is a class that implements thread functionality. It depends on IOwner interface. The responsibility of this class is dispatching notifications about data changes to subscribers. The class uses late interface binding to make a call-back to the subscribers.

- TEventCatalogBase is an abstract class. The abstraction of the event catalog allows us implement different kind of catalogs.

- TEventCatalog is the implementation of the abstract TEventCatalogBase class. The responsibility of this class is to register, store and manage the events on which data changes can occur. The class is also responsible for creating the event classes. Every event class implements the interface defined by by type of the event. It also implements the strategy of the notification mechanism. In our case the TDBChangeNotification class is the only event class in the system.

- TNotificationEvents is the class that implements notification events storage. The notification events are strings that defined in clients and published by clients. This class stores the notification events and their corresponding client interfaces.

- TCSNotificationEvent is a class that implements *ICS_NotificationEvent* interface. This class is a COM server and its responsibility is to create event classes based on event name and event class GUID provided by the clients. This class ensures that the client gets IDispatch interface of the created event class, so he can use late binding to connect to the server.

- TGIP is a helper class. The responsibility of this class is to marshal COM interfaces to the Global Interface Table of the application. This is required by COM in order to be able to use the interfaces in different threads.

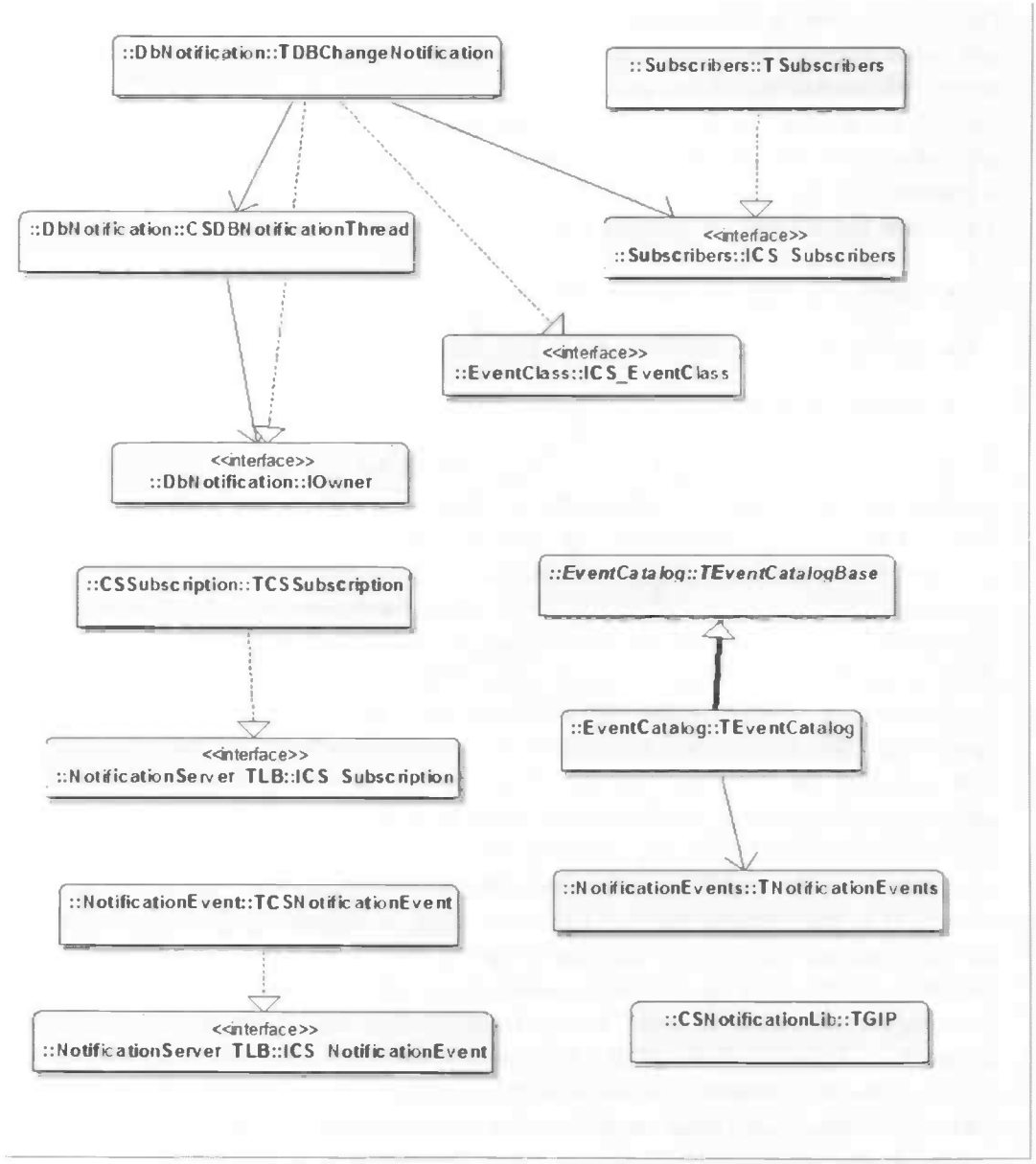


Figure 6.6: Logical view

6.2.2 Dynamic View

The dynamic view of the architecture addresses behavioral aspects of a system and contains dynamic elements of the architecture including flow of calls in the system, sequence diagrams, process/threads spawning, interprocess communications, etc.

Figure 6.7 shows the flow of the calls in the designed system. The numbers of the calls represents a one possible sequence of calls that would be made during subscribe-notify process. The client side of the system consists of the Notification component, which exposes the *ICS_DBChangeEvent* interface to the notification service. The notification component and the notification service communicate

through a network connection over which DCOM calls are made. To be able to get notifications, the subscribers register desired subscriptions at the notification server. When the publisher wants to publish an event, it passes the GUID of the desired event class to the notification service and the notification service creates an instance of the specified event class. Publishing of a change event looks like a transaction and occurs in two stages. First, the publisher prepares the change. There are three types of changes that the client can perform.

- Insert of a new data in the database
- Delete of an existing data from the database
- Update of an existing data in the database

In order to determine if a subscriber should be notified for the change, the notification service has to evaluate the attribute-filter of the subscriber against the changed data. Each time the change operation takes place, the notification service retrieves the required data-object from the database. For insertion operations, the attribute-filters are evaluated against the value of the inserted data after insertion. For deletion operations, the attribute-filters are evaluated against the value of to be deleted data before deletion. Update operations on the database can be seen as a combination of deletion of the old and insertion of new data. So during the update operations, the notification server retrieves two values (i.e. the old one and the new value) of the data and evaluate them against the attribute-filters. From the publisher's perspective, publishing an event is as simple as calling the appropriate method on the event class. By invoking the appropriate method through the event class interface, the publisher automatically publishes the event on the EventClass object. It is the responsibility of the event class to deliver the published event to the subscribers. After the publisher commits the change, the EventClass creates a thread which retrieves the list of subscribers from the subscribers catalog that have registered subscriptions. Then, it notifies (or skip notifying dependent on evaluation of the attribute-filter of the subscriber) the subscribers by calling the specified event method of the subscriber's interface. If notification of one or more subscribers fails, there is no simple way for the publisher to identify the subscribers or the reason for failure. This should generally be not a problem as the publishers rarely cares about subscribers' identities.

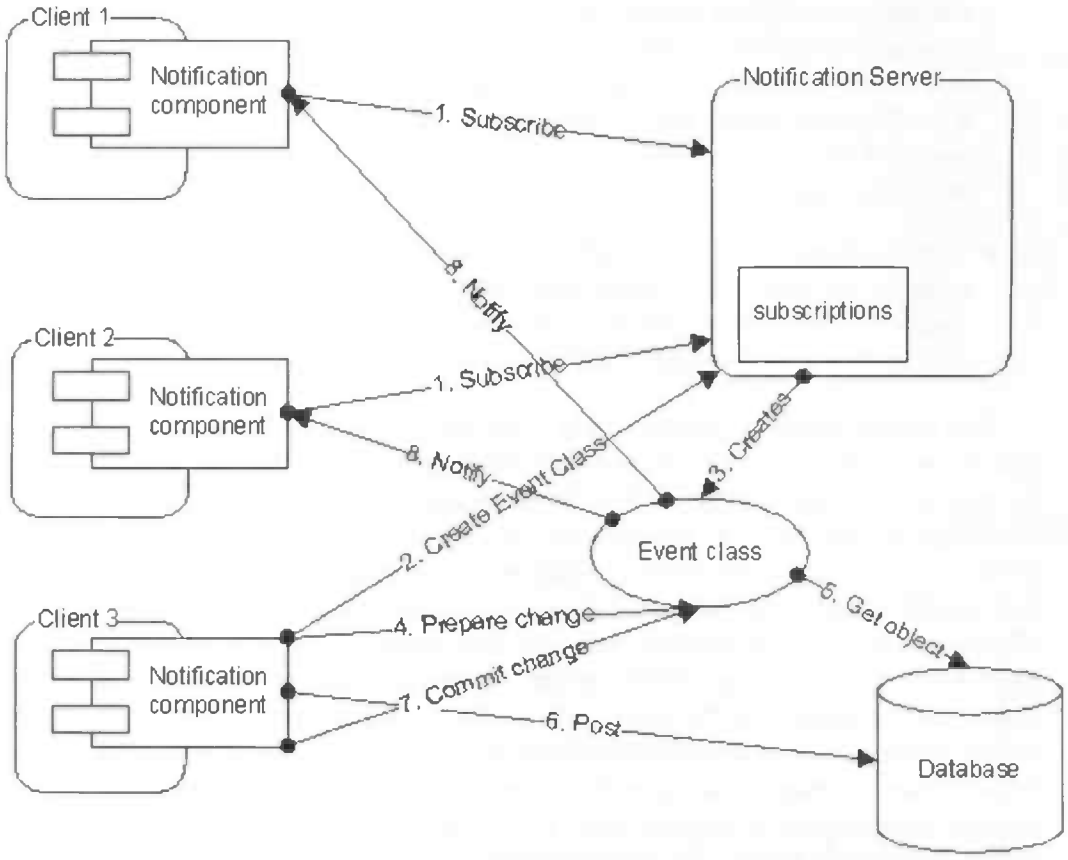


Figure 6.7: Flow of calls in the system

The event service does not provide any mechanism for specifying the order in which the subscribers receive the notifications. The default protocol is to deliver the event to one subscriber at a time. However, this default behavior can be changed by designing the event class to notify the published events in parallel, that is, to deliver the event to multiple subscribers concurrently. This will reduce the average delivery time of event notifications. Figure 6.8 is an example of a push notification for an update operation, being pushed to the CS-ECIS client. It shows the communication flow or the call sequence between different components of the system. There are five participants that take part in the communication.

- CS-ECIS is the existing client application which displays data-object collections (*ICS_Collection*) and needs to get notifications for these collections.
- Notification Component is a component that augments the client application in order allow making subscriptions and receiving notifications.
- Notification Service is an application that runs on the notification server and takes care of maintaining and managing of the client's subscriptions, creating of instances of event classes (in this case the event class is the *CS_DBCahngeNotification*), notifying the subscribers, etc.
- *CS_DBChangeNotification* is a COM object that receives published events from the clients (publishers). In order to avoid blocking calls from clients

during publishing of events (because the notification process can take some time and synchronous call here is not an option), this object tries to return the control back to the client as soon as possible. It does do this by creating a notification thread which takes care of further processing of the published event. This makes sure that the client application does not freeze long time when the number of subscribers is quite big.

- The notification thread, which is created by *CS_DBChangeNotification*, notifies the subscribers upon published events. This thread takes care of evaluation of attribute-filters of the subscribers against changed object. The thread dies as soon as all subscribers are notified.

The communication between these components flows as follow: In order to be able to get notifications, the client application requests the notification component to subscribe the specified *ICS_Collection* for change notifications. The namespace of change notifications, i.e. EvnetName, are divided into different partitions. The partition of the notification where it belongs is identified by the GUID of the factory which created the changed data-object. The event partitioning ensures the efficient notification mechanism, i.e. only the attribute-filters of subscribers that are interested in this class of events are evaluated. The notification component passes the EventName of the collection and the call-back interface of the subscriber to the notification service. The notification service stores the subscription in the subscriptions storage of the specified event class and returns the interface of the created subscription (FSubscription) to the notification component. The notification component passes the FSubscription back to the client. The client stores this interface in order to be able to manage the subscription. At a given time a change of a data-object in the *ICS_Collection* occurs. The internal notification mechanism of the client fires OnBeforeObjectPosted event. By catching the OnBeforeObjectPosted event, the notification component knows that the client is about to change a data-object with a ID which is at this time already known. The notification component requests the notification service to create a notification object of the type classId. The EventName in CreateNotification call specifies the name of the group of subscribers on which the notification object should operate.

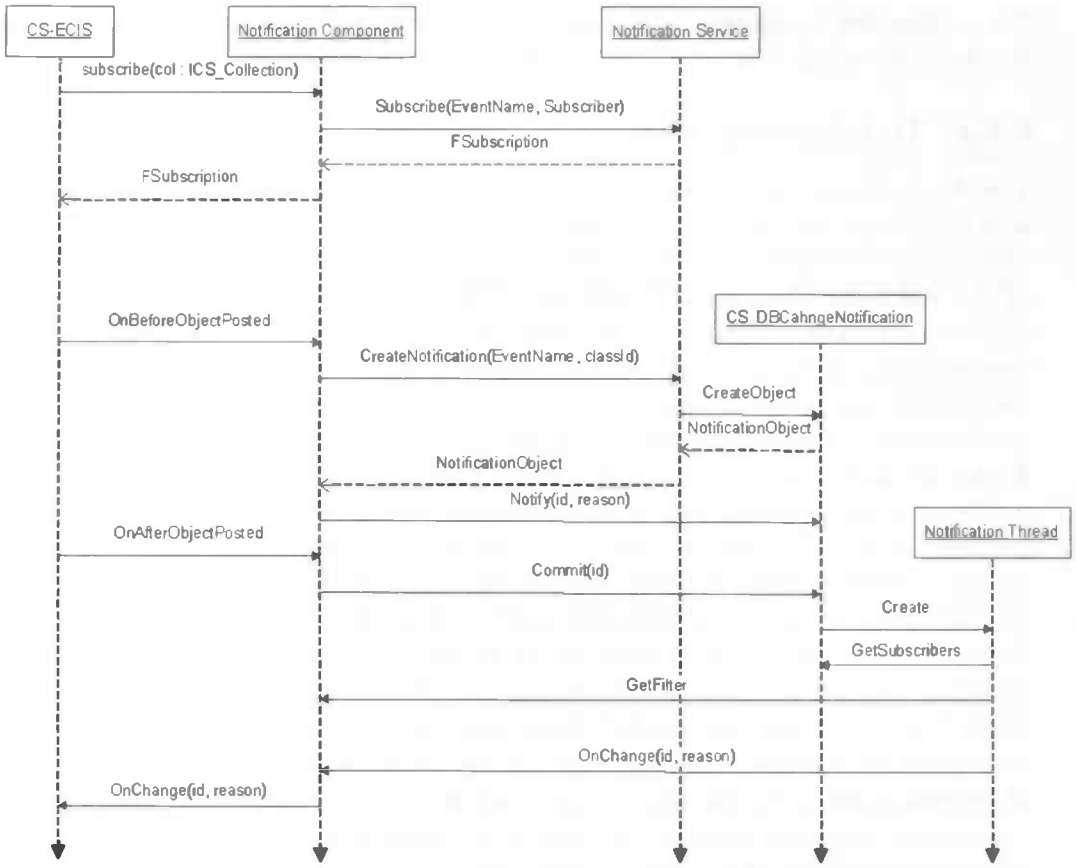


Figure 6.8: Sequence diagram of communication between the clients and the notification service

After the notification object is created (i.e. *CS_DBChangeNotification*), the notification component prepares the change notification by calling Notify method of notification object and passes the ID of the changed object and the reason of the change (i.e. insert, delete or update) to it. After that, the client application posts the data to the database and if the database transaction succeeds, it fires the OnAfterObjectPosted event. The notification component catches this event of succeeded transaction and commits the earlier prepared change to the notification service by calling Commit method of the notification object. At this time, the notification object creates a notification thread, starts it up and returns the control back to the caller. This ensures that the callee does not stay blocked upon the completion of the notifying process. The just started notification thread retrieves the list of subscribers for the specified event class and begins the notifying process. It iterates the subscription list and notifies the subscribers as follows. First, it requests the attribute-filter from the first subscriber in the list (the call GetFilter) and evaluates the changed data-object against this filter. If the filter evaluates to true then the thread makes a call-back to the subscriber by calling OnChange(Id, reason) method on the subscriber's interface. If the filter evaluates to false, the subscriber is skipped. In either cases, the process continues with the next subscriber in the subscribers list until all subscribes are passed. If for some reason the subscriber is unreachable then it is removed from the subscriptions list. Once

the notification component gets OnChange notification, it passes the notification to the client which handles it by refreshing of the affected data collection.

6.2.3 Development View

The development view of the architecture is depicted in the figure 6.9. The view is divided into five layers [21]. Each layer has well-defined responsibility en only depends on subsystems in the same layer or layers below. The layer 1 consists of the standard libraries of Delphi and COM servers supplied by the operating system. The layer 2 consists of the base and run-time libraries of ChipSoft. The responsibility of these libraries is adding some abstraction levels on used database technology and providing basic services for higher layers. The layer 3 consists of the ChipSoft's Visual Component Library (CS-VCL) and COM services. CS-VCL library is based on Delphi's standard VCL. Besides some added extra features (e.g notification component, etc), it takes care of own look and feel of ECIS client. In contrast to VCL library, most CS-COM services are domain dependent. For example, some of them map data objects to user tables and to their relations in the database. Others offer functionalities such as data change notifications, etc. Independent development of these COM servers is based on versioning system of interfaces exposed by these COM servers and fully supported by operating system. Layer 4 is a customer and product dependent layer. Services offered by this layer depend on the customer requirements and vary from hospital to hospital. Because of layered system, the libraries are subject of an independent versioning system. This means that each of them can have own version number and can be developed independently from other versions of libraries. This way, the dependencies between development layers are minimal which ensures an easy maintainable system.

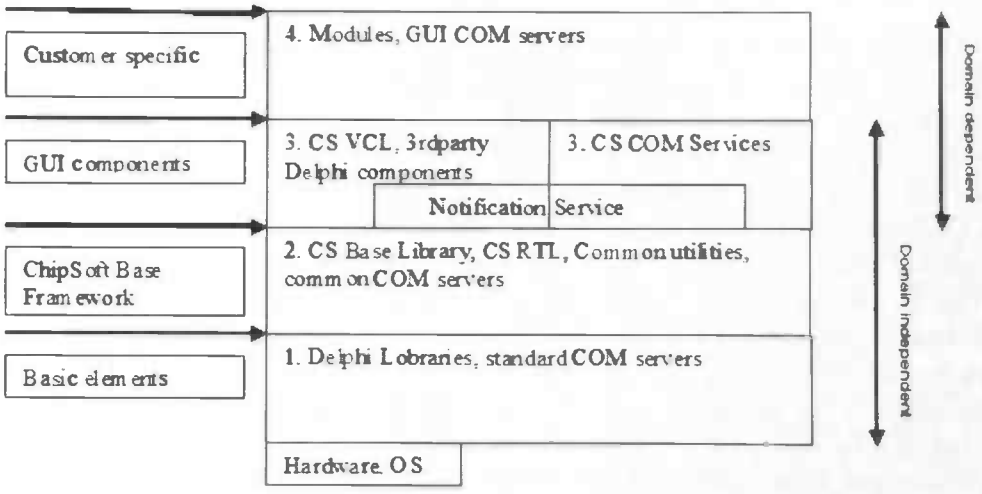


Figure 6.9: Development view

Figure 6.10 depicts the modules of the notification service and dependencies between them. Each module encapsulates a certain functionality of notification service. The description of each module is given below:

- FNotificationServer contains the main COM object of notification service.

- EventClass module contains the definition of an abstract event class. All concrete implementations of event classes should inherit from this abstract class.
- DbNotification module contains the implementation of the thread related notification classes.
- EventCatalog contains the definition of an abstract container for event classes. All concrete implementations of event containers should inherit from this class.
- NotificationEvent contains the implementation of COM objects of notification events.
- CSSubscription module contains the implementations of different kind of subscription classes (e.g. pull, push).
- NotificationServerGlobals contains some global variables (because Object Pascal lacks of static class variables, I was forced to use some global variables).
- UnitComp LogtoFile module contains logging facilities for debugging purposes.
- CSNotificationLib module contains definitions of some utility functions which are used by other modules.
- NotificationEvents contains the concrete implementation of notification events container.
- Subscribers module contains the implementation of internal representation of subscribed clients and their related clesses.
- $NotificationServer\_TLB$ module is a type library and contains all declaration of used interfaced.

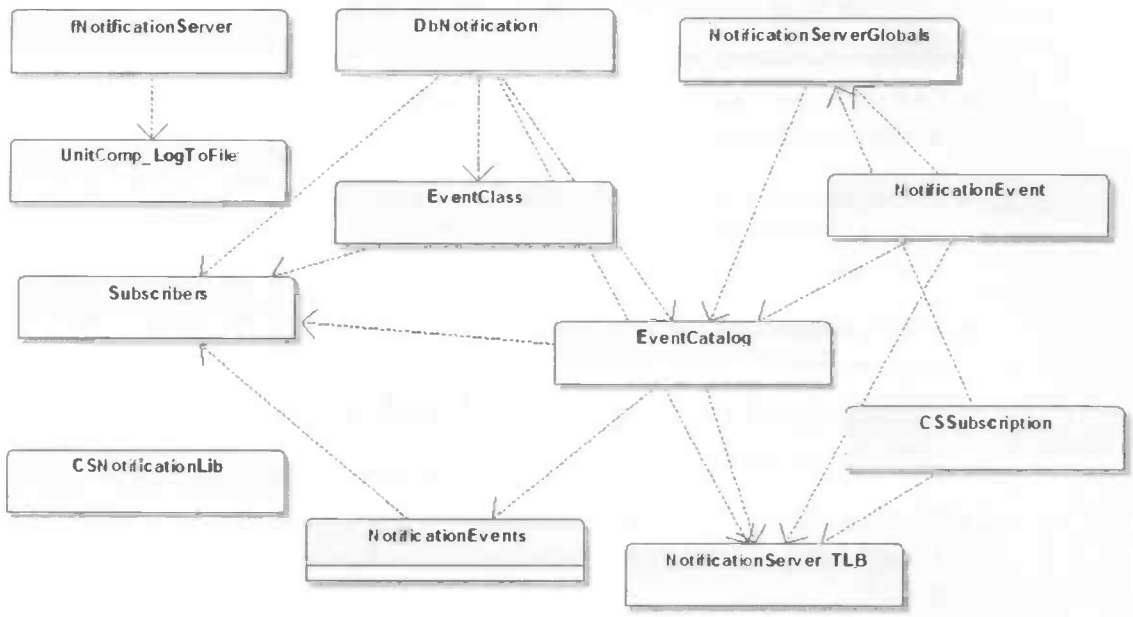


Figure 6.10: Dependencies between modules

6.2.4 Deployment View

The figure 6.11 depicts the deployment view of the architecture of the CS-ECIS system, including the notification service. The CS-ECIS applications are located on the client's computers. The notification service located on the notification server and the database is run on the database server. The client-server deployment architecture allows running the CS-ECIS applications on client's computers and keeping data on the centralized database server. The clients and the servers are connected through the local area network. The communication between the client and the database server relies on proprietary protocols offered by the used database. Generally, the database is used to store the application's data and the data processing occurs at the client's side. However, the clients can process data on the database side by calling stored procedures defined in the database. These stored procedures have to be set up at deployment time of the database.

The clients and the notification service use the network protocols offered by DCOM to communicate with each other. The communication relies on RPC protocol. It should be taken into account during configuring of network. In order to to use the notification service, first it must be installed on the notification server. The installation process involves the following steps:

1. installation of ChipSoft's runtime modules, libraries and the notification service executable.

2. registration of COM DLL's in the system.

3. configuration of the DCOM security settings of the clients and the notification server.

The client computers should be configured to permit anonymous logins, in order to make possible to receive call-backs from the notification server. At the notification server side, the authentication and authorization of the users should be set to appropriate *username* by which the users are able to log in on the notification server, e.g. "DomainUsers".

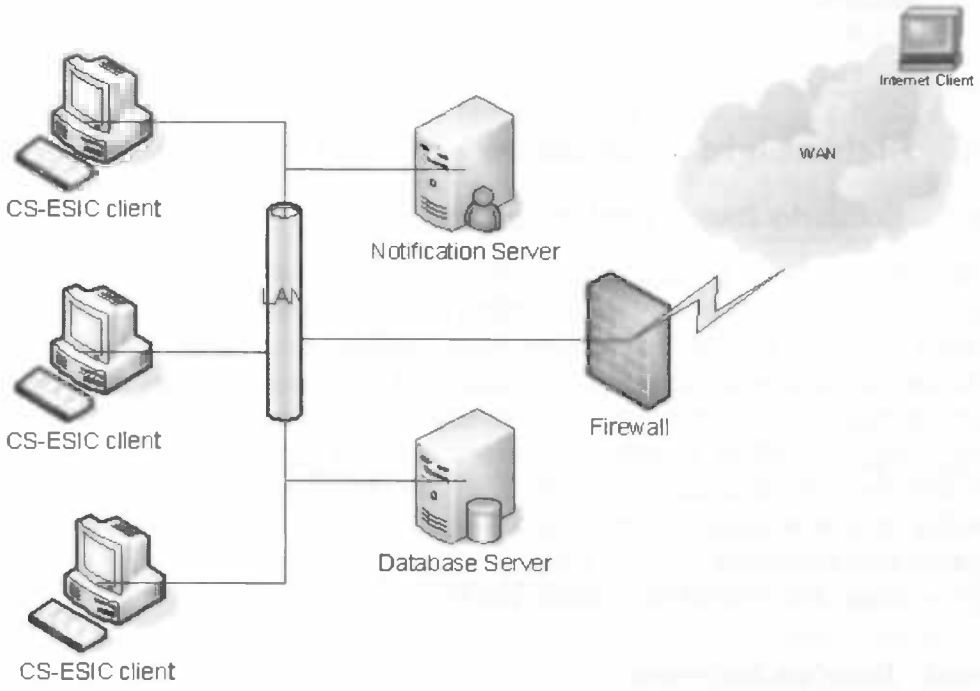


Figure 6.11: Deployment view

6.3 Notification and Content-based Filter Model

An event notification is a tuple of two typed attributes. The first attribute is a string. It describes the database wide unique identification number of the changed data record. The second attribute specifies the type of the change notification. For example, the following represents a change notification for an insertion operation.

```
Id = 1356947584
Reason = CS_Insert
```

A content-based event filter, or simply a filter, based on **xml** standard and specifies a set of attributes and constraints on the values of those attributes. Each constraint is a tuple that specifies the type, the name and the value of attributes. The operators provided by the filter expressions include all the common equality and ordering relations, operator "any" that matches any value, etc. The following is an example of a filter that matches articles in a table where "ExternNummer" is equal to "E001".

```
<FilterRoot>
  <FilterRoot Interface="{8DD51B1F-968D-11D3-AACD-0060979FF287}">
    <Type Value="0"/>
    <ChildElement Interface="{8DD51B1D-968D-11D3-AACD-0060979FF287}">
       <Veldnaam Value="ExternNummer"/>
       <Operator Value="GELIJK"/>
       <VeldType Value="1"/>
       <Values Value="E001"/>
    </ChildElement>
  </FilterRoot>
</FilterRoot>
```

6.4 Lightweight Evaluation of the Architecture

6.4.1 Scenario Based Evaluation

In this section I will present a lightweight evaluation of the designed architecture. The evaluation is conducted according to the Software Architecture Analysis Method (SAAM). SAAM is a scenario based method to evaluate certain quality attributes of software architecture. In practice SAAM has proven useful for quickly assessing many quality attributes such as modifiability, portability, extendability, integrability, as well as functional coverage. In this lightweight assessment, we will only focus on the most important quality attributes of the architecture: modifiability and extendability. The evaluation will show the weak or strong points, together with the points of where the architecture fails to meet the early mentioned requirements. The evaluation involves the following steps.

Step 1: Develop Scenarios

In this step the possible direct and indirect scenarios encountered by various classes of stakeholders are discovered. Scenarios fall into two categories: direct and indirect. A direct scenario is supported by the architecture because it is based on the requirements, which the system has been evolved from. Indirect scenarios require changes to one or more components of the architecture before they can be realized. In this evaluation we are only interested in direct scenarios because we want to see if they can be supported without requiring heavy modification of the architecture. The three stakeholder classifications are identified:

- Notification Service Developer (NSD) - Users who are responsible for designing/developing the notification server.
- Application Developer (AD) - Users who develop modules and services that use the notification service.
- Network Administrator (NA) - Users who are responsible for installing and maintaining the notification server.

We will concentrate on scenarios that primarily affect the notification service developers and network administrators because these are the major classes of the considered stakeholders in our evaluation. The table 6.1 shows the developed scenarios that are considered most likely to occur.

Scenario	Scenario Description	Classification	Affected Stakeholders
New event type	Extending the notification service with a new event type	Direct	NSD, NA
New database backend	Modifying the notification service to work with a new database server	Direct	AD, NSD
New notification type	Adding a new type of notification to the notification service	Direct	NSD, AD
Poll notifications	Modifying the notification service to implement poll notification mechanism	Direct	NSD, AD

Table 6.1: Scenarios

Step 2: Describe the Architecture

The detailed architecture description is available in previous chapters. We refer to the described architecture and we skip this step.

Step 3: Perform the Individual Scenario Evaluations

At this step we proceed with analysis of the impact that each direct scenario has on the architecture. This involves the identification of the components that are affected and the severity of the impact.

- Scenario 1: New event type
  Performing this scenario requires the following changes in the architecture.
  - Adding new event class inherited from TEventCatalogBase.
  - Adding new notification component that supports the new events.
  - Relinking and restarting the notification service.
- Scenario 2: New database backend
  Adding new database support requires relinking the notification service with the new CS-Base that supports the new database backend. Also, restarting of the service is required.
- Scenario 3: New notification type
  Performing this scenario involves the following modifications:
  - Developing a COM object that implements the new notification algorithm.
  - Developing a COM TCSNewNotificationEvent object that implements the required interface.

  - Developing a new notification component that implements the TCSNewNotificationEvent's interface.
  - Deploying the developed COM notification object in the Operating system.
  - Relinking and restarting the notification service.

- Scenario 4: Poll notification
  Performing this scenario involves the following modifications:
  - Developing a new class inherited from the abstract class AbstractStrategy that implements the new poll strategy.
  - Performing the scenario 3 with a appropriate new notification type.

The cost of the impact of a scenario that might have on a component is determined according to the table 6.2.

Scenario impact factor	Description
1	Make passive modifications (e.g. configuration file, etc.)
2	Make source code modifications
3	Make architectural modifications

Table 6.2: Cost of the impact of scenarios

The impacts of the scenarios on the different components of the system are given in the table 6.3.

Component/scenario	New event type	New database backend	New notification type	Poll notifications
Notification server	-	-	1	1
Notification service	2	-	2	2
Client	2	-	2	2

Table 6.3: Impact of scenarios

6.4.2 Prototype Evaluation

To better understand and validate the quality requirements such as performance, efficiency and security, an early evaluation of the notification service is carried out. The evolutionary prototype version of the notification service has been produced and it is used to validate the requirements. It also has been presented to users to obtain feedback that could be incorporated into the design. The evaluation is based on external metrics which has been collected while testing the running system. In order to gain information about architectural quality, the important quality attributes are refined into scenarios. The necessary functionality to perform these scenarios is implemented in the prototype. The executable prototype has been tested regarding quality attributes at runtime.

Test Environment

The test has been conducted at ChipSoft. There were 50 ECIS clients running on a terminal server. The notification service was running on a Pentium 2.5 Ghz Windows 2000 server with 1024Kb memory installed. The clients and the notification server were connected via a 100Mbit ethernet network.

Scenarios

The evaluation of the prototype focuses on evaluating the three quality attributes performance , efficiency and security. These three have been identified as important attributes of the system. In the following, the scenarios which are used during the evaluation of these quality attributes are described.

Performance

Performance is a measurement of the system response time for a functional requirement. Performance represents the responsiveness of the system, which can be measured by the time required to respond to events or by the number of events that are processed in a period of time. The performance scenarios require that the responsiveness of the notification service remains within the required limits.

- Scenario 1
  Each running client initiate 300 change event per minute stochastically. All clients are subscribed at the notification service in order to be notified about change events. The time needed for the notification service to notify the subscribers upon receiving a change event is measured.
- Scenario 2
  One running client initiate continual subscribe-unsubscribe requests to the notification service. The rate of served requests at the notification service is measured.

The results of the performance measurements are shown in the following table.

Scenario	Limit	Measured value
1	30 sec	< 1 sec
2	100 req/sec	100-150 req/sec

Table 6.4: Performance test results

Efficiency

The efficiency is about efficient utilization of resources such as processor, memory, network, etc. The efficiency scenarios require that the processor, memory and network resources utilized by the notification service at runtime remain within the required limits.

- Scenario 1
  The utilization of the processor during execution of performance scenarios is measured.
- Scenario 2
  The utilization of the installed memory during execution of performance scenarios is measured.
- Scenario 3
  The utilized bandwidth of the network interface at the notification server is measured.

The results of the efficiency measurements is shown in the following table.

Scenario	Limit	Measured value
1	80% of processor time	< 40% of processor time
2	80% of available memory	< 80% of available memory
3	100 Kb/sec	10-15 Kb/sec

Table 6.5: Efficiency test results

Security

The security is about the ability of the system to resist unauthorized attempts to access the system and unauthenticated use of notification services while still providing services to authorized and authenticated users. The security scenario requires correct handling of autorization and autentication of the communication participants by the notification service.

- Scenario 1
  A malicious client without proper credentials tries to subscribe at the notification service. The response of the notification service is observed at the client.
- Scenario 2
  A malicious clients without proper credentials tries to publish a data change event to the notification service. The response of the notification service is observed at the client.

The observations of the security test is shown in the table 6.6.

Conclusion

The following conclusions can be drawn from the evaluation conducted in this section. The considered scenarios reflect the typical situations that most anticipated to encounter in the future. The evaluation of these scenarios has showed that the cost of modifications in all cases limited to minor source code modifications and architectural modifications do not occur at all. The performance and efficiency related measured values are within required limits. The security of the

Scenario	Proper response	Observed response
1	access denied	Access denied attempting to launch a DCOM Server. The server is: GUID The user is Unavailable/Unavailable, SID=Unavailable
2	access denied	Access denied attempting to launch a DCOM Server. The server is: GUID The user is Unavailable/Unavailable, SID=Unavailable

Table 6.6: Security test observations

system works correctly granting access only authorized and authenticated users. From the evaluation it becomes clear that from an architectural perspective the designed architecture satisfies the considered quality attributes. Utilized design patterns in the architecture has facilitated the process of achieving these quality attributes.

6.5 Implementation Issues

6.5.1 Development Environment

During the project, the development is performed within Borland Delphi 7.0 environment. Delphi 7.0 is a mature development environment which consists of a number of tools such as code editor, GUI builder, TLB editor, debugger, etc. Delphi 7.0 has a rich, out of shelf object libraries that helps to make the development process fast and the program less buggy. As a programming language Object Pascal is used. Object Pascal is the successor of Imperative Pascal and it supports OO programming. However, lack of some language features such as namespaces, nested classes, class enumerations, static class variables, etc. makes it difficult to apply and use some OO programming techniques. It is somewhat hard to get clear design on modules level in Object Pascal. In Object Pascal the definition and the declaration of classes are contained in one compilation unit. This forces the developer to place dependent classes in the same module; otherwise you can easy get circular dependency references between modules, which results in huge and hard maintainable modules.

6.5.2 Integration into CS-ECIS

The challenge of this part of the project was to integrate the notification facility into the client application without making many changes in the existing code and architecture. Because of the rich and almost undocumented API of ECIS core libraries and many components, the integration was not as easy as it might look. Developers of modules at ChipSoft have developed own style and habits to program and use different components in different ways. It was impossible to find one solution to satisfy all those cases. The better solution was to make a new notification component and let the developers to seamlessly adapt the modules in order to use it. The notification component is developed as an independent component. The developers can drag and drop the component on desired GUI form and connect it to other components so that the connected components can

receive notifications (e.g. CSCollectionSource, etc.). The implementation of the notification component and the notification service is developed up to a prototype. However, in order to use it in a production environment (e.g. in hospitals), it should be extensively tested and developed further to make it more reliable if necessary.

Chapter 7

Discussion

This section will cover thoughts and reflections about the thesis and the work that I have done over the period of the project. During writing of this thesis I have tried not to go deep into implementation details but limit the scope of the paper mainly to the theory and research which I made during the project. This is because the implementations of the notification component and the notification service are domain dependent and cannot be applied to other domains and other cases. Moreover, they are tightly tied to the base libraries and other COM services of ChipSoft which are intellectual property of the company and are prohibited to be published. However, some important implementation issues and details have been discussed in order to give a complete picture of the system.

The project did not run without troubles and difficulties. The first difficulty that I have encountered was the lack of the documentation about the architecture of CS-ECIS and the existing code. The CS-ESIC has extensive and complex base libraries and because of little documentation I had to do much reverse engineering to understand the existing architecture and how the application works. This took lot of time and effort. Another difficulty was that most of the people which were less or more interested in the project have been working at headquarters of ChipSoft in Amsterdam. It was difficult to meet often and interview this people.

During the project it is much discussed about the suitability of using DCOM in the system. Most of developers have had a bad stereotypical image about DCOM and they distrust it. I think that the reason of this is the complexity of DCOM and little experience with it. They are partially right. I experienced working with DCOM as working with a black-box. The methodology behind DCOM is "COM on the wire". It means that where COM works should DCOM work too. But in many situations it is not the case. For example, if you have a mapped drive then COM objects that are registered on that drive work fine. When you try to use these objects from another computer then you get a vague error message that DCOM subsystem cannot start the DCOM server. Because it is not documented (or maybe somewhere documented but I couldn't find it) that the DCOM security manager prohibits the using of DCOM servers from mapped drives, you have to find it out by examining all the possible scenarios that can cause the error. There are many other catches to get DCOM working but once you get it working it works well.

During the project, I have presented a number of architectures of notification service. Not all of them were accepted by head developers of ChipSoft. Some

of architectures are thrown out because of constraints on network architecture or deployment. For example, an architecture that uses mailslots (a mailslot is a mechanism for one-way interprocess communications based on UDP protocol) to broadcast notifications within the system has been refused because the mailslots can broadcast messages only within one domain within a network. However, the network of some hospitals contains a number of interconnected domains. Another architecture was based on dynamic libraries that can be injected into the process space of the database server. These libraries can send notifications upon database table change event. However, the architecture was refused because a big chance to bring the whole database server down due to possible critical errors in these libraries.

Chapter 8

Related Work

This section compares the designed system with some existing notification solutions and mechanisms. The notification of a change made within a distributed database environment is not a new concept, as several systems have been implemented with notification mechanisms. There is also considerable research done in monitoring data changes at information sources and notifications of these changes [13], [17]. Different event detection and database notification techniques [1], [2], [15] have been studied. Compared with other change notification solutions, the designed system differs in a number of ways. First big difference is that data change monitoring and notification facility are separated from the database server. It delivers event publications efficiently to the clients via the publish/subscribe mechanism. This is a flexible approach and, as the number of clients grows, it scales better. In contrast, some of the proposed techniques [6] do not easily extends to scale up to distributed interoperable environments. The used publich/subscribe paradigm enables, in the future, to employ distributed database environments and ensures the scalability of the notification mechanism.
In some proposed techniques [8], [15], serious security concerns may arise if the clients are allowed to execute data definition statements on the database remotely. Because of the decoupling of the notification service from the database, it removes from the clients the need to know the location of the database. In addition, used DCOM technology is able to impersonate the users and the decision of public accessibility of data on the database to the users can be taken by database administrators during defining of domain models. This further improves the flexibility and the security of the system.
The designed architecture allows any number of clients to subscribe to an advertised event type (or a subset of it) by indicating attribute-filter expressions. The used attribute-filter technique is comparable with other proposed event detection techniques [4], [8], [17] which use, for example, the database tables to store user selection specifications. The attribute-filters are stored on notification service. They are practical and efficient representation of logical and arithmetical expressions. The user defined attribute-filters can be easily interpreted in terms of expressions and evaluated against data objects.
Another difference with existing solutions is that the designed system uses a combination of topic-based and content-based subscription models which provides flexible, efficient and scalable processing of notifications to handle large number of concurrently running clients with different attribute-filters on a variety of data

sources.

Chapter 9

Future Work

A prototype version of the notification service has been developed and tested. During the project, a number of interesting research topics have been emerged. Regarding the system correctness, it is desirable to investigate what other techniques can be used for efficiently evaluating validity of data collections in constantly changing concurrent database environment. For example, keeping the notification server informed about ongoing transactions on the database can help to avoid getting invalid collections by canceling the involved transaction. This can improve reliability of the system and further reduce the number of queries. In order to improve the performance of the system, it is desirable to design and implement a lazy notification mechanism, where the received notifications are propagated to data collections but refreshing of the changed data objects takes place only when these objects are actually accessed. It will be also useful to extend current event model to support composite events. By allowing more complex event patterns to be defined, a considerably larger set of situations of interest could be expressed. For example, in master-detail views, the subscription of the master collection will imply receiving events for details collections too.
The notification service designed and implemented during this project provides a foundation for further research and development.

Chapter 10

Conclusions

This paper describes the design and implementation of the prototype of the system wide notification service within CS-ECIS database environment. It uses an approach which provides a flexible, scalable and secure means for monitoring of information changes and notifying about them. This approach is based on the publish/subscribe communication model. The collaboration of the database, the notification server and the clients within this model provides a global event-based notification service. In the designed architecture, the client applications advertise the details of the changes to data to the notification service. The notification service publishes these events through a publish/subscribe system to the clients. The clients are able to subscribe to one or more events and can specify constraints on them by providing attribute-filters. As a consequence, they will be informed in a timely manner when a particular event occurs.

The publish/subscribe communication model used in this architecture allows database notifications to be disseminated without requiring clients to know the location of the change source involved. This allows to build more secure distributed data environment.

The designed and implemented notification service increases the flexibility and efficiency of CS-ECIS application running in distributed data environment and decreases the likelihood of stale data. It not only provides the clients with fresher data, but also decreases the amount of the database load and network bandwidth wasted on database polling. The designed notification infrastructure constitutes a good basis to be extended further which will allow CS-ECIS application to fully realize the benefits of globally aware distibuted data environments.

Chapter 11

Appendix

11.1 Screenshot of ECIS Client with Integrated Notification Facility

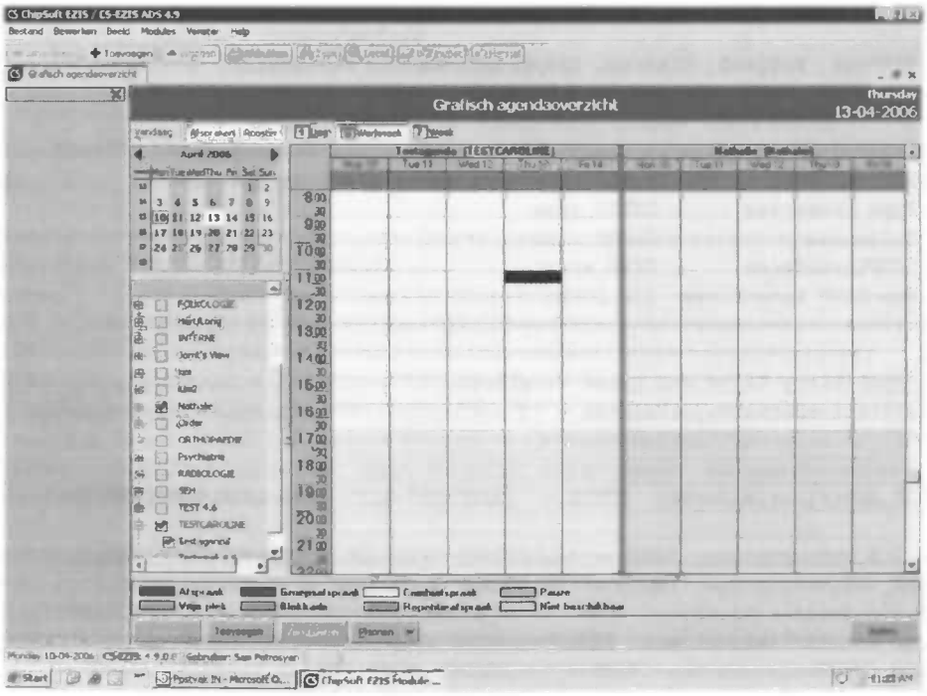


Figure 11.1: Screenshot of CS-ECIS client

11.2 Definition of Interfaces

```
unit NotificationServer_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or the
```

```
// 'Refresh' command of the Type Library Editor activated while editing the
// Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : 1.2
// File generated on 13-03-2006 11:23:31 AM from Type Library described below.

// ************************************************************************  //
// Type Lib: U:\NFS\Server\NotificationServer.tlb (1)
// LIBID: {B3E92297-4231-4236-A60E-9599F3B9DE4F}
// LCID: 0
// Helpfile:
// HelpString: NotificationServer Library
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\system32\stdole2.tlb)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}
{$VARPROPSETTER ON}
interface

uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;


// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  NotificationServerMajorVersion = 1;
  NotificationServerMinorVersion = 0;

  LIBID_NotificationServer: TGUID = '{B3E92297-4231-4236-A60E-9599F3B9DE4F}';

  IID_ICS_Subscription: TGUID = '{1D897168-7FD9-480C-B2CA-FF761E08A6E5}';
  CLASS_CSSubscription: TGUID = '{C1B4BDEE-2796-44F4-A6E2-8E78146343B1}';
  IID_ICS_NotificationEvent: TGUID = '{368EC1CC-3268-43FB-A11C-D86DE23A9F35}';
  CLASS_CSNotificationEvent: TGUID = '{F7A6D890-766A-4DBE-8E5E-304032CC1BCA}';
  CLASS_CSDBNotification: TGUID = '{708B27C3-F0FD-4C74-942A-1DE7C0375DB3}';
type

// ************************************************************************//
// Forward declaration of types defined in TypeLibrary
// ************************************************************************//
  ICS_Subscription = interface;
  ICS_SubscriptionDisp = dispinterface;
  ICS_NotificationEvent = interface;
  ICS_NotificationEventDisp = dispinterface;

// ************************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// ************************************************************************//
```

```
  CSSubscription = ICS_Subscription;
  CSNotificationEvent = ICS_NotificationEvent;
  CSDBNotification = IDispatch;


// ************************************************************************//
// Interface: ICS_Subscription
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {1D897168-7FD9-480C-B2CA-FF761E08A6E5}
// ************************************************************************//
  ICS_Subscription = interface(IDispatch)
    ['{1D897168-7FD9-480C-B2CA-FF761E08A6E5}']
    procedure Subscribe(const EventName: WideString; const EventInterface: WideString;
                        const Subscriber: IUnknown); safecall;
    function GetEventName: WideString; safecall;
  end;

// ************************************************************************//
// DispIntf:  ICS_SubscriptionDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {1D897168-7FD9-480C-B2CA-FF761E08A6E5}
// ************************************************************************//
  ICS_SubscriptionDisp = dispinterface
    ['{1D897168-7FD9-480C-B2CA-FF761E08A6E5}']
    procedure Subscribe(const EventName: WideString; const EventInterface: WideString;
                        const Subscriber: IUnknown); dispid 1610743808;
    function GetEventName: WideString; dispid 202;
  end;

// ************************************************************************//
// Interface: ICS_NotificationEvent
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {368EC1CC-3268-43FB-A11C-D86DE23A9F35}
// ************************************************************************//
  ICS_NotificationEvent = interface(IDispatch)
    ['{368EC1CC-3268-43FB-A11C-D86DE23A9F35}']
    function CreateEventClass(const EventName: WideString;
                              const ClassId: WideString): IDispatch; safecall;
  end;

// ************************************************************************//
// DispIntf:  ICS_NotificationEventDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {368EC1CC-3268-43FB-A11C-D86DE23A9F35}
// ************************************************************************//
  ICS_NotificationEventDisp = dispinterface
    ['{368EC1CC-3268-43FB-A11C-D86DE23A9F35}']
    function CreateEventClass(const EventName: WideString;
                              const ClassId: WideString): IDispatch; dispid 201;
  end;

// ************************************************************************//
// The Class CoCSSubscription provides a Create and CreateRemote method to
// create instances of the default interface ICS_Subscription exposed by
// the CoClass CSSubscription. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
// ************************************************************************//
  CoCSSubscription = class
```

```
    class function Create: ICS_Subscription;
    class function CreateRemote(const MachineName: string): ICS_Subscription;
  end;

// *********************************************************************//
// The Class CoCSNotificationEvent provides a Create and CreateRemote method to
// create instances of the default interface ICS_NotificationEvent exposed by
// the CoClass CSNotificationEvent. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
// *********************************************************************//
  CoCSNotificationEvent = class
    class function Create: ICS_NotificationEvent;
    class function CreateRemote(const MachineName: string): ICS_NotificationEvent;
  end;

// *********************************************************************//
// The Class CoCSDBNotification provides a Create and CreateRemote method to
// create instances of the default interface IDispatch exposed by
// the CoClass CSDBNotification. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
// *********************************************************************//
  CoCSDBNotification = class
    class function Create: IDispatch;
    class function CreateRemote(const MachineName: string): IDispatch;
  end;

implementation

uses ComObj;

class function CoCSSubscription.Create: ICS_Subscription;
begin
  Result := CreateComObject(CLASS_CSSubscription) as ICS_Subscription;
end;

class function CoCSSubscription.CreateRemote(const MachineName: string): ICS_Subscription;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_CSSubscription) as ICS_Subscription;
end;

class function CoCSNotificationEvent.Create: ICS_NotificationEvent;
begin
  Result := CreateComObject(CLASS_CSNotificationEvent) as ICS_NotificationEvent;
end;

class function CoCSNotificationEvent.CreateRemote(const MachineName: string): ICS_NotificationEvent;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_CSNotificationEvent) as ICS_NotificationEvent;
end;

class function CoCSDBNotification.Create: IDispatch;
begin
  Result := CreateComObject(CLASS_CSDBNotification) as IDispatch;
end;

class function CoCSDBNotification.CreateRemote(const MachineName: string): IDispatch;
begin
```

```
  Result := CreateRemoteComObject(MachineName, CLASS_CSDBNotification) as IDispatch;
end;

end.
```

RuG

Bibliography

[1] Christian Kleinerman Florian M. Waas Cesar A. Galindo-Legaria, Goetz Graefe. System and methods for database change notification. 2004. US 2004/0267741 A1.

[2] Wesley W. Chu. Database event detection and notification system using type abstraction hierarchy. 2002. US 6,427,146 B1.

[3] Microsoft Corporation. System and methods for requesting and receiving database change notifications. 2004. EP 1 462 958 A2.

[4] George A. Thomson Danil Brown, William Lance Easson. Rule based notification system. 2003. US 6,631,363 B1.

[5] Frank Stephen Serdy Donald James McNamara. Scalable notification delivery service. 2003. US 2003/0163533.

[6] David Nichols Douglas Terry, David Goldberg and Brian Oki. Continuous queries over append-only databases. Xerox Corporation.

[7] Jaswinder Pal Singh. Fengyun Cao. Efficient event routing in content-based publish-subscribe service networks. *IEEE INFOCOM*, 2004.

[8] Dieter Roller Frank Leymann. Subscription and notification with database technology. 2004. US 6,826,560 B1.

[9] Boris Gitlin Gus Rashid, Navruze Rashid. Rule based notification system. 2004. US 2004/0230661 A1.

[10] Opher Etzion Joris Mihaeli. Event database processing. IBM Haifa Research Lab.

[11] Sandeep Bhatia Subhash B. Tantry Kevin Hsiaohsu Tu, Peiwei Mi. Event notification system. 2002. US 2002/0154010 A1.

[12] Calton Pu Ling Liu and Wai Tang. Supporting internet applications beyond browsing: Trigger processing and change notifications. Georgia Institute of Technology.

[13] Donald R. Nelson. Method and apparatus for providing notification to a customer of a real-time notification system. 2002. US 6,496,568 B1.

[14] Shailendra Mishra Rajit Kambo, Namit Jain. Method and apparatus for automatic notification of database events. 2003. US 2003/0055829 A1.

[15] Adam W. Smith Pablo Castro Robert M. Howard, Michael J. Pizzo. Systems and methods for employing a trigger based mechanism to detect a database table change and registering to receive notification of the change. 2004. US 2004/0193653 A1.

[16] Mario Banville Roch Glitho. Implementation of notification capabilities in relational databases. 2000. US 6,041,327.

[17] Alexandre Edelman Ruben E. Ortega. Auction notification system. 2003. US 6,549,904 B1.

[18] Marie-Anne Neimat W. Kewin Wilkinson. Method of maintaining of cached data in a database system. 1993. US 5,261,069.

[19] Cesar Galindo-Legaria TorstenGrabs Christian Kleinerman Florian Waas. Database change notifications: Primitives for efficient database query result caching. Microsoft Corp. OneMicrosoft Way.

[20] David J. Wills, Fergus M. McTavish. Stateful push notifications. EP 1 555 793 A2, 2005.