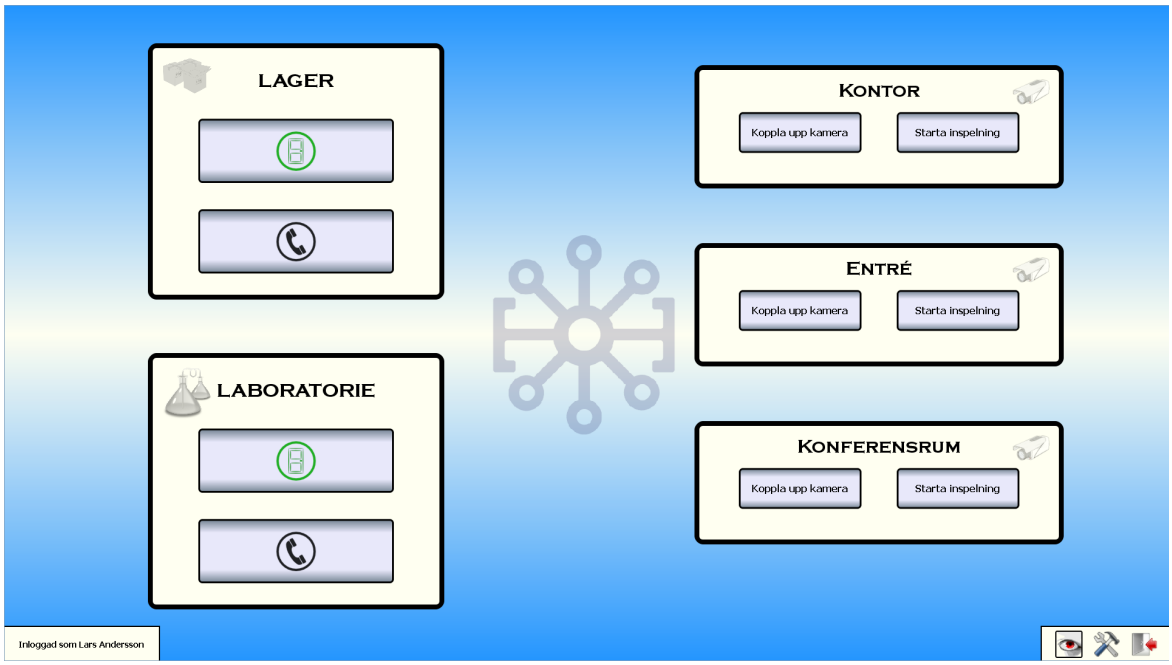
CHALMERS



Developing an Integrated Surveillance Interface

Master of Science Thesis

LARS ANDERSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, February 2011

Developing an Integrated Surveillance Interface

Lars Andersson

Examiner: Graham Kemp

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
Screenshot of a surveillance system control panel developed in this project.

Department of Computer Science and Engineering
Göteborg, Sweden February 2011

Preface

I would like to extend my thanks and appreciation to Galder Security AB who have allowed me to do this project at their company. Also many thanks to my examiner Graham Kemp, my supervisor Jens Kjärsgaard and my family and friends.

Abstract

This report goes through the design that was created and implemented to produce the foundation for a modern, fast and reliable surveillance interface that builds on the existing total security system OnGuard from the company Lenel. The solution is divided into two programs, a server application which handles the communication with the OnGuard system and other third party programs that the program should be able to communicate with, and a client application which the user uses to interact with the server. The server application is developed using C# and is compiled to run under the .NET virtual machine. The client application is developed in C++ using the Qt and Qt Quick framework and can be compiled to run under many different operating systems.

This project was never intended to, nor does it, deliver any finished product but more so investigates the possibilities of building a surveillance interface that builds upon OnGuard and utilizes Qt Quick to produce a modern and fast user interface.

Contents

List of Abbreviations

ANSI	American National Standards Institute
API	Application Programming Interface
CIL	Common Intermediate Language
DTD	Document Type Definition
GUI	Graphical User Interface
HWND	Handle to a Window
INI	Initialization
MOC	Meta-Object Compiler
QML	Qt Modeling Language
RDBMS	Relational Database Management System
SDK	Software Development Kit
SGML	Standard Generalized Markup Language
T-SQL	Transact-SQL
UUID	Universally Unique Identifier
VMS	Video Management System
VRM	Video Recording Manager
WMI	Windows Management Instrumenation
XML	eXtensible Markup Language
XSchema	XML Schema

1 Introduction

1.1 Background

In society today we see an increasing demand for security to prevent everything from fires to burglary and vandalism. A high security facility today often consists of multiple systems that takes care of many different tasks. These systems carry out their duties on their own but it can be hard for an operator, who has the task of monitoring the security of the whole facility, to get a good overview and to control parameters in the different systems. Therefore it is often desirable to connect all the systems to a superior surveillance system which lets the operator interact with all the systems through one single graphical user interface.

Galder Security AB is a distributor of the total security system OnGuard from the US based company Lenel. OnGuard communicates with all the different systems in a secure facility and lets the operator interact with these through their own graphical user interface Alarm Monitoring.

1.2 Purpose

Galder Security AB feels that the user interface, Alarm Monitoring, that is integrated with OnGuard is lacking a lot in graphical appeal and also in its intuitivity. With this masters project they want to investigate the possibilities of developing their own program as a replacement to Alarm Monitoring that builds on and utilizes the OnGuard system. But with a modern graphical user interface and many improvements and missing functions implemented.

Galder Security AB have many features that they want to have incorporated into the product, below are a few of them outlined.

- The system must communicate with existing integration platform brand Lenel. Through this platform most of the communication with the different hardware is done.
- The system must present events such as burglar alarms, intercom calls, portable panic alarms etc. both graphically and in text.
- The system must be able to send signals to open doors, establish calls, set the emergency sections, control the security cameras, etc.

- The system must be able to show vector graphics in which the operator can zoom in and out. In the images it should be possible to track the location of the emergency sections and their status. It should also be possible to navigate between these images.
- The system must be able to play video streams from cameras.
- Video streams should be able to be linked up automatically when alarm events and control systems occur but also manually by the operator. Mobile cameras are to be controlled via "click image" and/or via a graphical joystick.
- The system shall be able to be put into configuration mode where the authorized user using the login can define system parameters, such as operator accounts, windows settings, etc. Operator Accounts should be able to determine what the individual user can and cannot do within the system.

Implementing all of the outlined features is outside the scope given the time frame of this masters project, but the resulting solution from the project should have some of the features implemented and serve as good foundation for future work to build upon.

1.3 Method

The new surveillance interface will communicate with OnGuard to get information about and control systems that OnGuard supports. The chosen design splits the solution into two different applications, a server application which communicates with the OnGuard system and a client application which communicates with the server. This way it is possible to connect multiple clients to one server and let them show and control different things in the OnGuard system. The server is developed in C# and is compiled to run under the .NET virtual machine and the client is developed in C++ using the Qt and Qt Quick framework.

The server communicates with OnGuard over WMI. Communication with third party software, Bosch VMS, is implemented using Bosch VMS SDK's API, a library written for C# .NET by Bosch. A communication protocol based on XML was developed to communicate between the server and the client and the query language XPath is used to query things in the XML data in both the client and the server. In the client a small settings file, following the INI file format, was designed to store some client specific data but most of the client data is stored in the server database. The server uses

SQL to query things in the database and a subset of the format called WQL to query things in WMI.

2 Background

2.1 OnGuard

OnGuard is a suite of programs and services developed by the company Lenel. Together with a Microsoft SQL Server database they form a complete overarching surveillance system which can communicate with a wide variety of systems and put these at the operator's fingertips through the graphical user interface Alarm Monitoring.

OnGuard has support for communication with third party software through Windows Management Instrumentation (WMI).

2.1.1 Windows Management Instrumentation

Windows Management Instrumentation is a set of extensions to the Windows Driver Model that provides an interface through which applications can communicate with each other [19].

The communication is done using a provider-consumer model where the provider manages some object on the computer [19], in our case it manages the OnGuard system. The consumer can then query data from the provider using WQL, which is a subset of the American National Standards Institute Structured Query Language (SQL) [16]. It is also possible to subscribe to different events asynchronously, so that the consumer always will be up to date with new events. Most of the data and functions available in OnGuard are exposed in this way through a WMI provider.

2.1.2 Microsoft SQL Server

Microsoft SQL Server is a relational model database server produced by Microsoft [12]. The latest versions of it are Microsoft SQL Server 2008 and Microsoft SQL Server 2008 R2 of which Lenel uses the former of the two. The two primary query languages for the database are T-SQL and ANSI SQL [17].

2.2 Bosch VMS

Bosch VMS is a system for management of digital video, audio and data in-

tended for use in a video surveillance environment. The environment usually consists of some surveillance cameras which records its material onto certain storage devices. Which storage devices the cameras should use is managed by a VRM. The VRM also fetches the video data fragments when a playback of the video data is required, since the video data can be fragmented over many different storage devices. The Bosch VMS controls the cameras and the VRM and can show recorded and live video data from the VRM and the cameras. It also manages different alarms and events that occurs in the system [1][3].

2.3 C#

C# is a object oriented programming language developed by Microsoft. It is similar to C++ but introduces many new things to make the programming easier on the programmer. For example foreach loops, very nice event programming, array bounds checking, detection of attempts to use uninitialized variables and more [18]. C# is usually compiled to run under the .NET virtual machine. When doing this it is compiled into a Common Intermediate Language (CIL) which is then assembled into bytecode which the .NET virtual machine executes.

Important when writing long running applications is to handle memory management well and at all cost avoid memory leaks. When letting the code run as managed under .NET this is greatly simplified by the use of a garbage collector and a managed heap [10]. Using these, deallocation of most resources in C# is handled automatically. But when working with unmanaged resources the programmer must remember to clean up accordingly. Unmanaged resources are for example window handles (HWND), database connections, file I/O, etc [13]. The easiest way to handle the cleanup of unmanaged resources is to implement Dispose and Finalize methods [13], similar to the use of destructors in other programming languages like C++.

2.3.1 .NET

.NET is a programming framework developed by Microsoft. .NET includes both a large library of coded solutions to common programming problems and also a virtual machine which runs the code in a managed environment [15]. .NET supports several programming languages which allows language interopability and the .NET library is available to all these programming languages that are supported.

2.4 C++

C++ is a object oriented programming language developed by Bjarne Stroustrop. It is regarded as a middle-level programming language as it has both low-level and high-level language features [20].

C++ is most often compiled to run directly on the computer hardware, in contrast to so called managed code which executes in a virtual machine. With this comes more responsibilities to the programmer, since there is no garbage collector or managed heap it is very easy to create memory leaks and this must of course be avoided at all costs. But since we don't have the overhead that an automatic garbage collector brings we get a performance boost in applications written in C++ compared to for example C# running in the .NET virtual machine.

2.4.1 Qt

Qt is a cross-platform application development framework which is widely used for creating GUI-applications. Qt uses standard C++ with a preprocessor, called the Meta Object Compiler (MOC), to enrich the language. These enrichments are, to name a few, foreach loops, event handling systems with signals and slots, run-time type information, and a dynamic property system. In the Qt framework are also a lot of libraries for many different solutions to different coding problems. These features in the Qt libraries include SQL database access, XML parsing, thread management, network support, a unified cross-platform API for file handling and more.

To implement this enrichment of the language the meta-object compiler reads the source files and if it encounters the Q_OBJECT macro in a class it produces the C++ source files containing the meta-object code for those classes. In these classes it is possible to declare many different macros to make the meta-object compiler include different things in the generated source files. One very important thing, at least when we are working with Qt Quick, is that we can add properties to the dynamic property system using the Q_PROPERTY() macro and also add enumerations using the Q_ENUMS() macro. It is also possible to add special functions called signals and slots using the Q_SIGNAL and Q_SLOT macro. These special functions are used for Qt's event system and can be connected together using a special connect function. One signal can be connected to multiple slots and one slot can be connected to multiple signals [8]. They are also available under the keywords "signals" and "slots" for declaring multiple signals and slots.

Listing 1 shows a simple C++ header file which uses the Q_OBJECT macro at the top of the file, indicating to the Meta-Object Compiler that it should process this class. The Meta-Object Compiler will process the file and produce a new file containing the final C++ code where all the MOC macros have been exchanged for C++ code. The Q_PROPERTY macro adds a boolean variable called "pushed" to the dynamic property system with a function for reading, a function for writing and a signal for notifying when the value of the variable has changed. The signals macro makes the pushedChanged function available as a signal in Qt's event handling system, and it can be connected to as many slots or signals as desirable which will then be called when the signal is emitted using the "emit" keyword. The slots macro makes the connectMeToSomeEvent function available as a slot in Qt's event handling system, and it can be connected to as many signals as desirable and will be called when these signals are emitted using the "emit" keyword.

```
1       class Button : public QObject
2       {
3         Q_OBJECT
4
5         Q_PROPERTY(bool pushed READ pushed WRITE setPushed
              NOTIFY pushedChanged)
6
7       public:
8           explicit Button(QObject *parent = 0);
9
10          bool pushed() const { return m_pushed; }
11          void setPushed(const bool p_newPushed);
12
13      signals:
14        void pushedChanged(const bool p_newPushed);
15
16      slots:
17        bool connectMeToSomeEvent();
18
19      private:
20        bool m_pushed;
21      };
```

Listing 1: C++ header file which utilizes some of the Qt Meta-Object Compiler macros.

One important feature in Qt which is used extensively in combination with Qt Quick is the List Models. The List Models are classes which can be populated with items and then connected to views in QML. A popular approach is to create a model by inheriting QAbstractListModel, adding all the needed functionality to this class and then instantiate it and populate it with items. Another class is then created by inheriting QSortFilterProxyModel, adding some functionality and instantiating it and assigning the

former model as a source model to this class. The first class that we created is then the source model and the second is a proxy model that uses the first as a source. The proxy model does not create any new items, it is simply used for sorting and filtering by redirecting indexes and hiding indexes. It is this proxy model that is connected to the view and therefore it decides which items a user should view. This flow from QAbstracListModel through QSortFilterProxyModel and finally to a view in Qt Quick is shown in Figure 1. In this example all items with a index higher than three are filtered out in the QSortFilterProxyModel and the remaining items are sorted in ascending order.

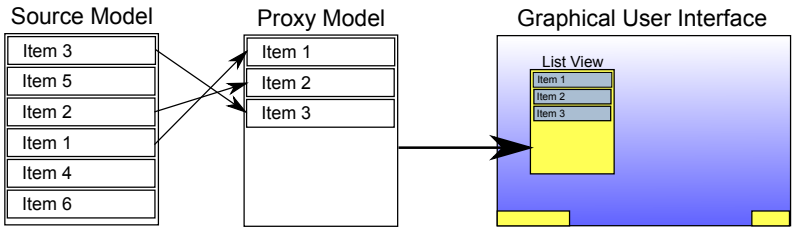


Figure 1: The relationship from source model, through a proxy model and into a view.

2.4.2 Qt Quick and QML

Qt Quick is a brand new technique for developing GUIs that has been invented as a response to the demand of fluid and highly polished GUIs that are used in todays mobile phones.

Qt Quick uses a declarative approach where the designer or programmer designs the GUI in the QML language, which is a script-like language based on javascript. The QML code can be interpreted to create the GUI, either using a standalone interpreter like the QML Viewer, or utilizing the Qt Quick framework classes in an application written in C++. With a standalone interpreter it is easy to produce a design and also to test it and evaluate it extensively, which speeds up the development process a lot and makes sure that the end result will be well polished and tested. But to make the most of your application it is often the case that you want to create the main application in C++ and then run the QML code on top of this program to produce the GUI, this is what's accomplished with the Qt Quick framework classes. Using the Qt Quick framework classes it is also possible to extend the QML language, expose properties and different models through declarative contexts and make connections between the QML code and the C++ program using signals and slots [7].

Four classes stand out among the Qt Quick framework classes.

- QDeclarativeEngine
- QDeclarativeComponent
- QDeclarativeContext
- QDeclarativeItem.

These four classes are the classes that are necessary to interpret the QML code and build up the GUI. QDeclarativeEngine is the actual interpreter which instantiates our QML components. QDeclarativeComponent encapsulates a QML component definition, the actual QML code. QDeclarativeContext defines a context within the QML engine through which it is possible to expose data to the QML components instantiated by the engine. QDeclarativeItem is the most basic of all visual items in QML, all visual items in Qt Declarative inherit from this class. The simplest way to set up a QML declarative environment using these four classes is to first create an instance of the QDeclarativeEngine. Then create a QDeclarativeComponent with the root QML item and assign it to the instantiated engine. Next instantiate a new QDeclarativeContext using the engine's root context and then create the root QDeclarativeItem using the recently instantiated QDeclarativeComponent. Through a declarative context it is possible to reach properties, enumerations, slots and signals that have been added to Qt's dynamic property system from the QML code.

QML provides mechanisms to declaratively build an object tree using QML elements. These QML elements, together with their internal properties, specifies how the GUI should look like and how it should behave.

```
1       import Qt 4.7
2
3       Rectangle {
4         id: background
5         width: 640; height: 480
6         color: "white"
7
8         Text {
9           id: textLabel1
10          anchors.centerIn: parent
11          text: "Hello World!"
12        }
13      }
```

Listing 2: Hello World in QML code.

In Listing 2 we see a simple Hello World example coded in QML. On the first line we have an "import" statement that imports the Qt module which contains all of the standard QML elements, without this statement the Rectangle and Text element wouldn't be available. After this we see an object tree with two QML elements, a Rectangle element as the root element and a Text element as its child. Both of these QML elements have some default properties that can be set. First in both of the elements we set the property named id, this is an unique identifier which makes it possible to reference between different elements in the code. The width and height in the Rectangle element are simply the width and height that the rectangle should occupy and the color property is the color that it should have. There are some predefined color strings but the colors can also be specified using color codes. In the Text element we set the anchors.centerIn property to be the center of its parent, effectively putting the Text element in the center of the parent Rectangle element. Last we set the text that the Text element should display with the text property.

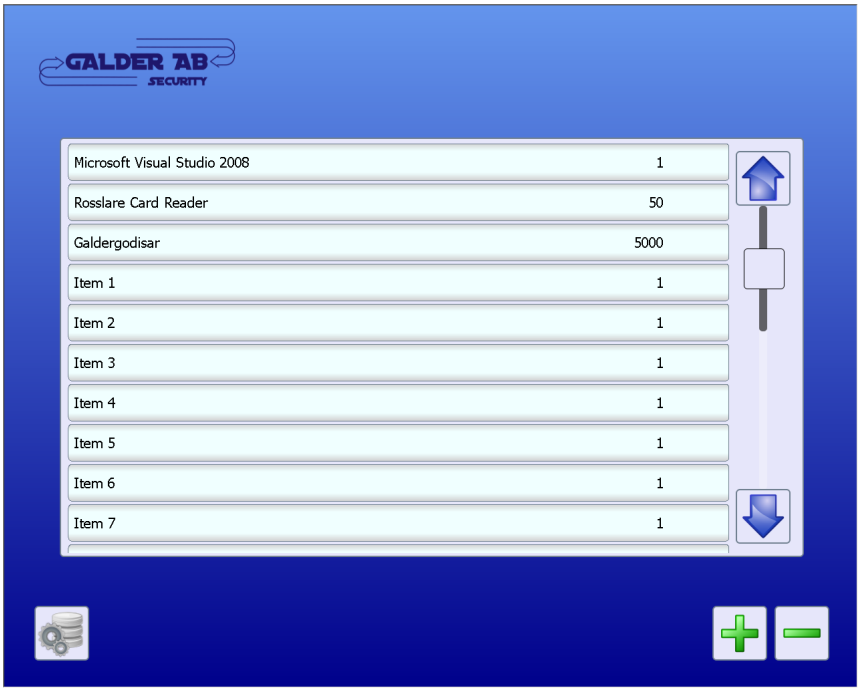


Figure 2: An inventory management program that was developed during the evaluation of Qt Quick (list view).

All declarative QML code is built up in the same way with a single item as the root and all the other components branching from this single root item. Other features that are available in the QML code are states to structure up property changes, transitions to animate the property changes, signal and

slot functions and support for writing functions using javascript.

Figure 3: An inventory management program that was developed during the evaluation of Qt Quick (list item expanded).

To give the reader a better understanding of how a GUI developed using the Qt Quick framework can look like, two screenshots from an inventory management program that was developed during the testing of Qt Quick and QML are shown in Figure 2 and Figure 3. In the first picture we see a simple list showing the items in the inventory with buttons below it to add items to the inventory, remove items from the inventory and a button to the left which is used to access the database settings and exit the program. When an item in the list is clicked on, the list item is expanded with a smooth animation and the item in its expanded form is what can be seen in the second image. Here some information is shown about the specific item and there are two buttons, one to edit the information and one to close down the view and return to the view in the first image.

2.5 Event-Driven programming

Event-Driven programming is a programming paradigm in where the programs flow is determined by different events. In its most basic form a program written using event-driven programming uses an event loop which is

clearly divided into two sections, event selection and event handling.

```
1   public class EventProducer
2   {
3     public delegate void MyEventHandlerFunc();
4
5     public event MyEventHandler MyEvent;
6
7     public void Trigger()
8     {
9       OnMyEvent();
10    }
11
12    protected void OnMyEvent()
13    {
14      if (MyEvent != null)
15      {
16        MyEvent();
17      }
18    }
19  }
20
21  public class EventConsumer
22  {
23    public void MyEventHandler()
24    {
25      // Handle event
26    }
27  }
28
29  public class TestApplication
30  {
31    static void Main(string[] args)
32    {
33      EventProducer eventProducer = new EventProducer();
34      EventConsumer eventConsumer = new EventConsumer();
35
36      eventConsumer.MyEvent += new EventProducer.
            MyEventHandler(eventConsumer.MyEventHandler);
37
38      eventConsumer.Trigger();
39    }
40  }
```

Listing 3: Example of event-driven programming in C#.

Event selection is where the thread checks if any event has occured and event handling is when it calls the functions that should handle the different events that have occured. Many events can be connected to one handler and many handlers can be connected to a single event. Both C# and C++ with Qt have support for event driven programming.

In Listing 3 we see a simple example of event-driven programming in C#. The example is divided into three different classes. The TestApplication class which contains the entry point for the program, the EventProducer class which generates the events in our example and the EventConsumer class which handles the events that arise in the EventProducer. In the EventProducer we first have a delegate which describes how a function which should be able to subscribe to the event should look like. After this we have the actual event declared. The Trigger function is just a function that is called from the main function to be able to generate an event. The OnMyEvent function is called from inside the class when we want to generate an event, the function checks if there are any event handlers subscribing to the event and if there are it calls them. In the EventConsumer class there is only one function and that is the event handler. It is structured in the same way as the MyEventHandlerFunc delegate in the EventProducer, the same return value and the same number of parameters with the same types. In the TestApplication class we have only the main function of the program, here we create one instance of the EventProducer and one of the EventConsumer, we connect them together and then we emit an event by calling the Trigger function. By using the "+=" operator it is possible to add multiple functions that should act as event handlers to the event and we make sure that the function is able to handle the event by creating an instance of our earlier declared delegate and passing the function as a parameter to this.

Listing 4 shows the same example that we saw in C# but now with Qt's signals and slots system. We have two classes and one main function. As in the C# example we have the EventProducer which produces the events and the EventConsumer which handles the events. We see that all the classes have the Q_OBJECT macro so that the Meta-Object Compiler will process them and produce the C++ code that makes the signals and slots work. In the EventProducer class we first have a signal declared. This signal effectively is our event. Then we have a trigger function so that we can make the class emit this event. Important to note here is the emit keyword which is not native C++ syntax but also a macro that the Meta-Object Compiler processes and produces C++ code for. The macro is simply used to emit an existing signal. In the EventConsumer we only have a slot which can be connected to signals. In the main function we simply create instances of the EventProducer and the EventConsumer, connect the signal in the EventProducer with the slot in the EventConsumer and call the trigger function to get an example of a triggered event.

Of course these are very simple examples of event-driven programming in the two different settings and the trigger function is only there for demonstration purposes. Usually you have something external triggering the different events.

```
1   class EventProducer
2   {
3     Q_OBJECT
4
5   signals:
6     void myEvent();
7
8   public:
9     void trigger()
10    {
11      emit myEvent();
12    }
13  }
14
15  class EventConsumer
16  {
17    Q_OBJECT
18
19  public slots:
20    public void myEventHandler()
21    {
22      // Handle event
23    }
24  }
25
26  int main(int argc, char *argv[])
27  {
28    EventProducer eventProducer;
29    EventConsumer eventConsumer;
30
31    QObject::connect(&eventProducer, SIGNAL(myEvent()), &
          eventConsumer, SLOT(myEventHandler));
32
33    eventProducer.trigger();
34  }
```

Listing 4: Example of event-driven programming in C++ with Qt.

2.6 XML

XML, which stands for eXtensible Markup Language, is a set of rules for encoding documents in machine-readable form. It is a subset of the Standard Generalized Markup Language (SGML). With XML it is possible to produce documents which can be processed quick and easily by a machine and still be human readable [4].

2.6.1 XPath

The XML Path Language is a query language which primary purpose is to address parts of an XML document. Beside this it also provides basic functions for manipulation of strings, numbers and booleans [6] [2].

2.7 INI File format

The INI file format is a standard for configuration files. It is simply a text file with some basic structure, built in such a way so that it should be easy for a human to open the file, find and edit things. The INI file format is commonly associated with Microsoft Windows but has been deprecated there in favor of the registry and XML.

3 Results

The result of the masters project is a surveillance interface with a backend that communicates with the total security system OnGuard from the US based security company Lenel. The backend is a server application running on the same machine as the OnGuard service and program suite. The frontend is the client application written in C++ using the Qt framework. The Qt Quick framework was chosen after extensive testing and it delivers what it promises, great looking programs that are responsive and quick to develop. The clients can be compiled to run on any of todays popular operating systems, Linux, Windows or Mac OS. The server application is designed to be able to handle multiple clients connecting and communicating with it.

Figure 4 shows a general overview of the project solution. In the server computer we have the OnGuard system communicating with hardware that it supports. Also present in the server computer is the server application developed in the masters project which communicates with the OnGuard system over WMI. The server application communicates directly with other hardware that OnGuard doesn't support, as of the end of this project this is only Bosch VMS. The clients connect to the server over TCP/IP using an XML based communication protocol and it is then possible to interact with all the hardware through the clients.

The protocol between the server and the client is, as mentioned earlier, a protocol based on XML that was developed during this master thesis. Galder Security AB hasn't decided if they want this protocol to be open to the public or not, seeing that an open protocol could create some security risks. Therefore it is sadly not discussed especially thoroughly in this thesis. In both the server and the client, XPath is used to parse the incoming XML packages. In the server this is done using the XmlDocument class that is available in the .NET framework and in the client it is done using the QXmlQuery class available in the Qt framework. All the XPath queries and the structure of the XML documents have been written manually without the use of techniques like DTD or XSchema. As of the writing of this report there are 60 different elements that are used to build up the XML documents that are passed back and forth between the client and the server. A design where everything is built up using XML elements, without any XML attributes is used. In the authors opinion this improves the human readability of the XML document.

The total solution implements the following functionality partly and the design aims to support the addition of the other functions, mentioned in the introduction, in the future.

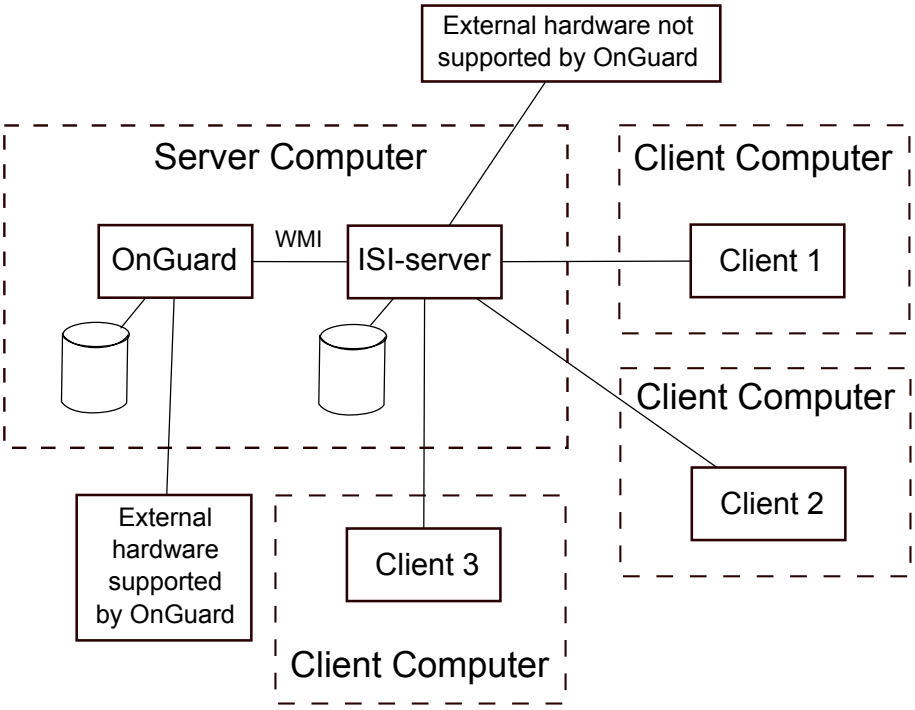


Figure 4: An overview of the project solution.

- The system must communicate with existing integration platform brand Lenel. Through this platform most of the communication with the different hardware is done.
- The system must present events such as burglar alarms, intercom calls, portable panic alarms etc. both graphically and in text.
- The system must be able to send signals to open doors, establish calls, set the emergency sections, control the security cameras, etc.
- The system shall be able to be put into configuration mode where the authorized user using the login can define system parameters, such as operator accounts, windows settings, etc. Operator Accounts should be able to determine what the individual user can and cannot do within the system.

3.1 The design of the server

Since the server should be designed to run on the same computer as the OnGuard system it felt natural to design the server closely knit to the things that the OnGuard system is closely knit to. Lenel uses Microsoft developed

products like WMI and Microsoft SQL Server and they are also a Microsoft Gold Certified Partner [11], hence the server should be developed using Microsoft developed products and techniques to make the interfacing against WMI and Microsoft SQL Server easy. Also when looking at the market of devices that we in the future probably want to communicate with using the server, we see a majority of developers using the .NET platform and developing interfacing libraries in C# or Visual Basic. Therefore the server was designed for and implemented with the C# language, utilizing many of the prebuilt functions in the .NET library and running in the .NET virtual machine. Also since the server will be an application that should run for a very long time without restarting, it is very nice that it is running in a virtual machine with garbage collection.

One big feature of the C# language is the event system where it is easy to connect several event handlers to one event and vice versa. This is used extensively in the design of the server to produce a nice design where it is easy to add new things without disrupting the flow in the application in any way. For example if we want to do something more with an incoming data package from the client it is easy to add another event handler which listens to the "New input data ready event" which is shown in Figure 5.

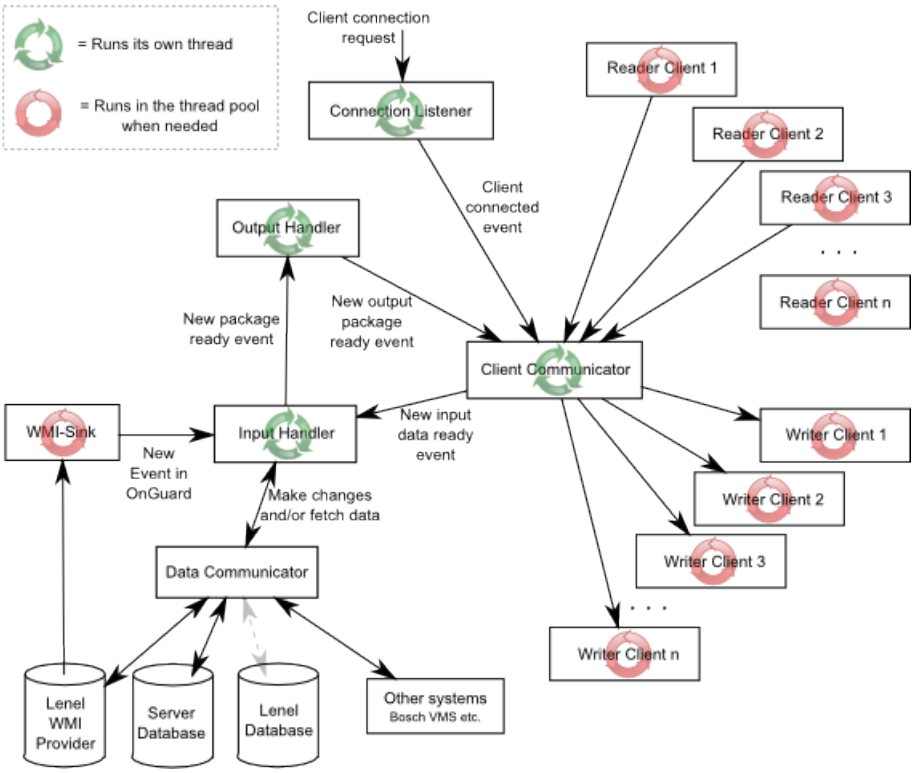


Figure 5: The general design of the server.

The chosen design of the server is shown in Figure 5, we see here that we have four permanent threads running and then a lot of code running in the .NET thread pool. The .NET thread pool is what handles asynchronous operations in .NET, "The .NET Framework uses thread pool threads for many purposes, including asynchronous I/O completion, timer callbacks, registered wait operations, asynchronous method calls using delegates, and System.Net socket connections." [14].

The design approach in Figure 5 is leaning towards the "one thread per client" approach, which is a fast approach with a few tens of clients but can get bad quickly when the client count goes up [5]. But since the thread pool is used for communication with the clients this should be more effective than an ordinary "one thread per client" solution [9].

The number of threads adds some complexity to the solution with the handling of shared resources and thread synchronization, but to make the server responsive and quick for many clients at the same time it should be a good solution.

As of the end of this project the server makes changes to and collects data from mainly two different sources, the OnGuard WMI provider and the server database. It is also possible to connect the server to Bosch VMS clients and control some parameters in these systems. In the future it could be necessary to also make a connection directly against the OnGuard database to be able to implement some functionality against OnGuard, but this should be avoided as far as possible since WMI is the recommended way for a third party program to get into the OnGuard system. Communication with the server database is done using SQL queries. Communication with the WMI provider is divided into two parts. First there is one connection where WQL queries are executed and methods can be called on retrieved objects. Second there is also a WMI sink that subscribes to certain events from the WMI provider which are then delivered asynchronously.

Currently there are only three external events that the server reacts to.

- Connection attempts from clients.
- Packets received from connected clients.
- Events that are received in the WMI sink from the OnGuard WMI provider.

The following text describes the event and data flow in the server design shown in Figure 5 for the three external events that it reacts to.

When the server receives a connection attempt the Connection Listener raises a "Client Connected" event with the newly established connection. The Client Communicator listens to this event and adds this connection to its internal list of connections and starts an asynchronous read on the connection. If the server doesn't receive a correctly formulated handshake request with a client ID that is present in the server database within a certain time interval the connection will be discarded.

The second external event, when a packet is received from a client, triggers the Client Reader to send out the newly received package in a "Package Received" event. The Client Communicator receives this event and then resends it as a "New Input Data Ready" event which the Input Handler picks up and adds the package to its internal list of packages. The Input Handler is somewhat the brain of the server and it takes the incoming package, parses the information to a server command, executes the command and then creates a package that should be sent back to the client. The execution of the command could mean changes in the server database, changes in the OnGuard WMI provider or retrieving of data from either of the two, data which should potentially be sent back to the client. After the Input Handler has finished with its tasks regarding a certain packet it raises a "New Package Ready" event containing the newly created return package. The Output Handler takes this return package and adds it to its internal list of outgoing packages. When it has sent out all packages that were waiting before the new package it calls the send function in the Client Communicator with the given package which starts an asynchronous send to the correct client. Which client should receive the return package is decided by comparing the Client ID in the message and that which is registered on the client from the handshake.

When the third external event, receiving an event in our WMI sink from the OnGuard WMI provider happens, the WMI Sink will create an event package using the information received. After this it will raise a "New Event In OnGuard" event containing the newly created event package. The Input Handler will react to this event and put the event package in its internal package queue. When it's time for the Input Handler to process the event from the package queue it checks what event it is, looks in the database if any client subscribes to this event and creates an event package for each client and raises a "New Package Ready" event with each package. The output handler reacts as in the second case and sends these packages out in the same manner.

Heartbeat using Ping and Pong messages is implemented in the server and is handled internally by the Client Communicator, without packages being passed to the Input Handler. If a client doesn't respond to one of the ping

messages that is sent out from the server, it will be disconnected.

3.2 The design of the database

The server database is a Microsoft SQL Server relational database management system. Microsoft SQL Server is used because this is the RDBMS that Lenel uses for its OnGuard system, and since we are working closely with OnGuard we use the same.

The current design of the database is shown in Appendix A. The core of this design is the Clients and Users tables. When a client connects to the server it has to perform a handshake procedure providing a UUID that is registered on a client in the Clients table, else the connection will be discarded. When a user tries to log in on a connected client the username and password combination has to be present in the Clients table, else the log in will fail. The server is responsible for updating the LoggedOnUser column and the Connected column in the Clients table, so that it always corresponds to the current state.

The LnlEvents table contains the events from OnGuard that are available for the clients to subscribe to, which they do in the ClientsSubscribesToLnlEvents table.

The Bosch VMS clients that the server should be able to send functions to are registered in the BoschClients table.

The other tables are for functions that have not yet been implemented.

3.3 The design of the client

Figure 6 shows the design of the client. The client is developed with a "Model View Control" design pattern in mind, but with QML and the declarative environment acting as both View and Control. The only threads running are two event handler threads. One which handles all the input from the user and updates the GUI and one which is used to communicate with the server and to decouple these tasks from the main thread so that the GUI doesn't freeze.

At startup of the client it loads some settings from a local INI file into the main model, it then uses these settings to make a connection attempt against the server. If the connection was successful it will send a handshake

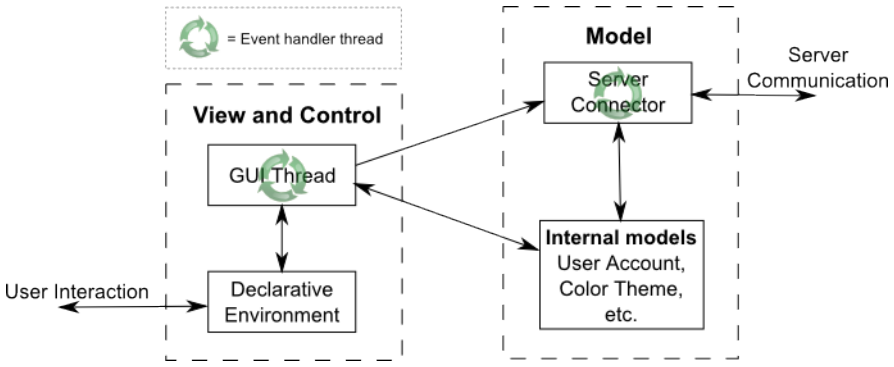


Figure 6: The general design of the client.

command to the server and wait for a response. In the response will be embedded all the client specific data that needs to be loaded from the server at client startup, implemented so far are only the input signals and output signals that the client subscribes to. In the future things like fonts, colors, sizes and how the workspace should be customized should probably also be loaded from the server at handshake. The input and output signals that are loaded from the server are used to populate the main input signals model and the main output signals model. These are classes that inherit QAbstractListModel, which makes them possible to connect to views in QML or to proxy models to be able and sort and filter them. After the handshake has been done the client is officially connected and a user can now log on to the server. After login the user is presented with the workspace that is customized for that particular client and, depending on the users permissions, will be able to enter the administration menu to make configuration changes in both the client and the server.

Currently the design for each individual client workspace is specified in QML code. This could be solved in many different ways but by specifying it only using QML code the doors are kept open to use Qt Creators development interface for Qt Quick applications for this task. Instead of developing some own development interface for specifying how the workspace for a client should look and which components the user should be able to interact with. This development interface is not yet released but will be soon.

In the workspace that is specified using QML code it is possible to connect different events and relays in OnGuard to different buttons and animations in the declarative environment. It is also possible to control different Bosch VMS clients that are registered in the database from the declarative environment.

3.4 Usage example

With the solution, as it is at the end of this project, it is possible to implement control panels. A control panel is simply some buttons and indicators on a touchscreen which makes it possible for a user to control different things in the environment that is being surveyed. When Galder Security create this type of control panel today they use Alarm Monitoring on a separate client. From now on they could use the solution presented in this master thesis instead, producing the same functionality but with much more graphical appeal and also saving money on licensing costs.

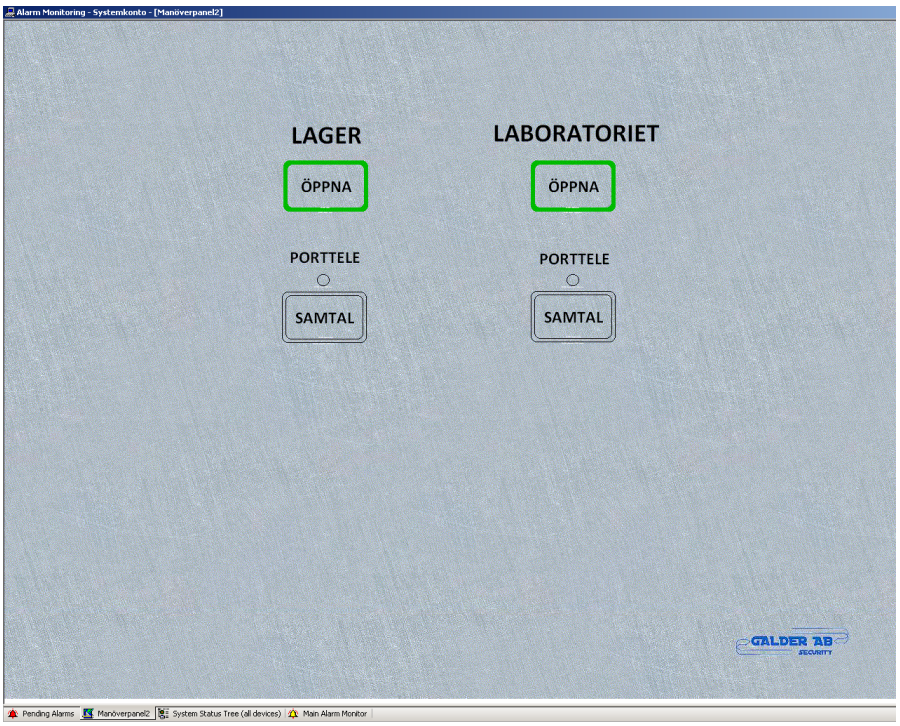


Figure 7: A screenshot of a simple control panel implemented in Alarm Monitoring.

In Figure 7 we see a simple control panel implemented in Alarm Monitoring. The top left button opens the door to the inventory. The button also has a color indication on its outer edge, green if the lock is locked and red if it is unlocked. The top right button functions in the same way as the top left button, but for the door into the laboratory. The bottom left button connects an intercom call between the intercom telephone at the client and the intercom telephone outside the inventory. The button also has a color indication on its outer edge, red color indicating that an intercom call is connected. Above the button is a small dot that changes color to yellow

when there is an incoming call. The bottom right button has the same functionality as the bottom left button but with the intercom telephone outside the laboratory.

In Figure 8 we see a simple control panel implemented in the workspace for a client that has been developed in this project. The two panels on the left side implements the exact same functionality as the control panel that was shown in Figure 7 and described above. The top left panel implements the same functionality as the two buttons to the left in Figure 7 and the bottom left panel implements the same functionality as the two buttons on the right in Figure 7. Instead of text on the buttons, as in Figure 7, the functionality of the button is instead described with a small icon in the middle of the button. A door indicates that the button opens a door when pressed and a telephone indicates that the button connects an intercom call when pressed. The color indication on the outer edge of the buttons in Figure 7 has been exchanged here with a color change of the icon in the middle of the button. The little dot that changes color to yellow in Figure 7 when there is an incoming call has been exchanged with an animation of the icon in the middle of the button. On the right side we have three panels that, instead of communicating through OnGuard as the panels on the left side, communicates directly with a Bosch VMS client. Each of the three panels implement the same functionality but for different cameras. The left button connects the specific camera on a big monitor and the right button starts recording of the stream from the camera onto a hard drive.

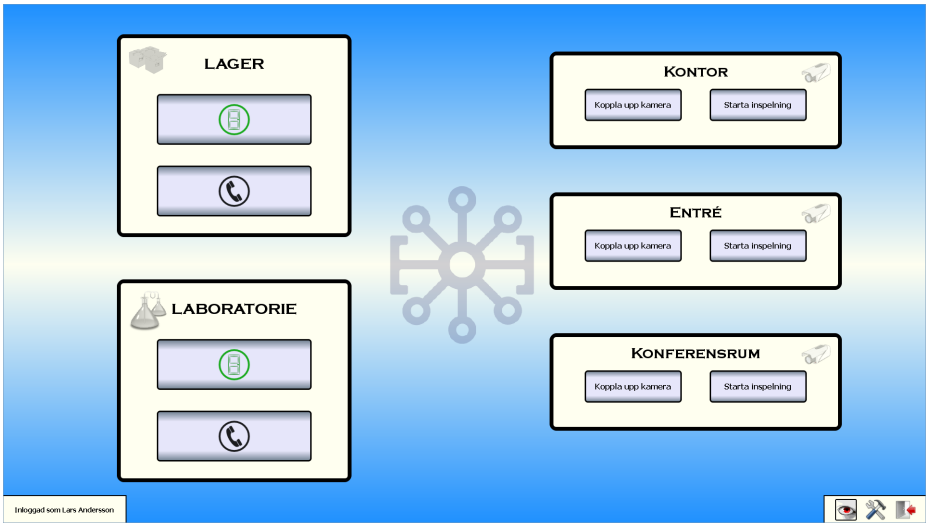


Figure 8: A screenshot of a simple control panel implemented in the client workspace.

4 Discussion

Due to the use of a development framework that the developer did not have any previous experience in, the design approach on this project has been very iterative. In the beginning of the project an inventory management program was developed, both due to the fact that the company needed an application for this but also to be able to learn and evaluate the Qt framework. During the development of the client four iterations of the QML code for the GUI have been developed before the final iteration was settled upon.

No special project management method was used due to the fact that there was only one developer. Looking back it would have been nice to use for example Scrum and some Agile Software Development techniques, since these effectively force the development team to have very good communication and follow up with the management. If the development will continue on this project these techniques should probably be adopted.

When investigating what method to use to connect against a database using C#, many different methods came up. The approach that looked like the best was Linq to SQL, which is a method where objects that represent the different tables in the database are created automatically to be used in the code. But due to the fact that only Microsoft Visual Studio Express was available for use and using Linq to SQL with this development environment seemed less than straight forward, the final solution uses the SqlConnection class in the .NET library instead with which it is possible to send SQL queries to the database and iterate over the data that is returned.

5 Future work

Before continuing development on a solution that implements all of the desired functionality that is found in Alarm Monitoring, the integration with OnGuard needs to be investigated further. It is still uncertain if functions like acknowledge alarms and intercom communication easily can be communicated through the OnGuard system, and these are functionality that are crucial for a full-fledged surveillance system like this.

Another thing that should be considered before further development is if the administrative interface should be put in its own program that communicates directly with the server database. Including this logic in the server and the client means a lot of extra work and complexity for very little profit.

In the future a layer of security also needs to be added to the solution, but seeing that these security systems often resides in closed networks something like encryption of the communication should probably be sufficient.

6 Conclusions

In this project a solution that implements some of the functionality that Galder Security AB requested has been produced. The solution uses a client-server model and allows for multiple clients connecting at the same time to the server. From the client and through the server it is possible to communicate with OnGuard and the different subsystems that OnGuard communicates with. It is also possible to communicate directly with different Bosch VMS clients from the client and through the server. The solution uses new and exciting techniques in Qt Quick to produce the graphical user interface in the client that the user interacts with.

With the solution in its current state it is possible to produce tailor-made control panels. The design of the control panels is specified using QML code and different signals and slots make it possible to interact with OnGuard and Bosch VMS from this code through the client and the server. With these control panels it is possible to replace existing control panels that are usually implemented in Alarm Monitoring, making it possible for Galder Security AB to potentially sell copies of the product in a relatively early stage of its development and in this way support the continuing development of the product.

Development has also started on a simple inventory management system for the company to use. This was a by-product from the beginning of the project when evaluating and learning Qt and Qt Quick.

References

[1] Galder Security AB. Bosch VMS. Accessible at: http://www.galdersecurity.com/cctv/bosch-vms.html, year = 2010, note = [retrieved 2010-01-12].

[2] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernndez, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPath) 2.0*, 2007. Accessible at: http://www.w3.org/TR/xpath20/.

[3] Bosch. Bosch VMS. Accessible at: http://www.boschsecurity.se/content/language1/html/1671_SVE_XHTML.asp, 2011. [retrieved 2010-01-12].

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008. Accessible at: http://www.w3.org/TR/2008/REC-xml-20081126/.

[5] Francesco Callari. Operating Systems Course Material. Accessible at: http://www.cim.mcgill.ca/~franco/OpSys-304-427/lecture-notes/node68.html, 1995. [retrieved 2011-01-12].

[6] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*, 1999. Accessible at: http://www.w3.org/TR/xpath/.

[7] Nokia Corporation. *Qt Quick*, 2010. Available in the Qt Assistant.

[8] Nokia Corporation. *Using the Meta-Object Compiler (moc)*, 2010. Available in the Qt Assistant.

[9] Ian Griffith. Doing Work Without Threads. Accessible at: http://www.interact-sw.co.uk/iangblog/2004/09/23/threadless, 2004. [retrieved 2011-01-12].

[10] James Kovacs. Identify And Prevent Memory Leaks In Managed Code. *MSDN Magazine*, 2007. Accessible at: http://msdn.microsoft.com/en-us/magazine/cc163491.aspx.

[11] Lenel. Lenel Alliances. Accessible at: http://www.lenel.com/alliances, 2010. [retrieved 2010-11-30].

[12] Microsoft. BI Resources - SQL Server 2008. Accessible at: http://www.microsoft.com/bi/products/sql-server-2008.aspx, 2009. [retrieved 2010-02-10].

[13] Microsoft. Implementing Finalize and Dispose to Clean Up Unmanaged Resources. Accessible at: http://msdn.microsoft.com/en-us/library/b1yfkh5e(v=VS.90).aspx, 2010. [retrieved 2010-12-07].

[14] Microsoft. The Managed Thread Pool. Accessible at: http://msdn.microsoft.com/en-us/library/0ka9477y.aspx, 2010. [retrieved 2010-12-01].

[15] Microsoft. The .NET Framework. Accessible at: http://www.microsoft.com/net/, 2010. [retrieved 2010-12-07].

[16] Microsoft. WQL (SQL for WMI). Accessible at: http://msdn.microsoft.com/en-us/library/aa394606(v=VS.85).aspx, 2010. [retrieved 2010-12-07].

[17] Microsoft. Microsoft SQL Server 2008 - Overview. Accessible at: http://www.microsoft.com/sqlserver/2008/en/us/programmability.aspx, 2011. [retrieved 2010-02-10].

[18] Microsoft. The C# Language. Accessible at: http://msdn.microsoft.com/en-us/vcsharp/aa336809, 2011. [retrieved 2010-02-10].

[19] Walter Oney. *Programming the Microsoft Windows Driver Model, 507-510.* Microsoft Press, 2002.

[20] Bjarne Stroustrup. The C++ Programming Language. Accessible at: http://www2.research.att.com/~bs/C++.html, 2010. [retrieved 2010-02-10].

Appendix A

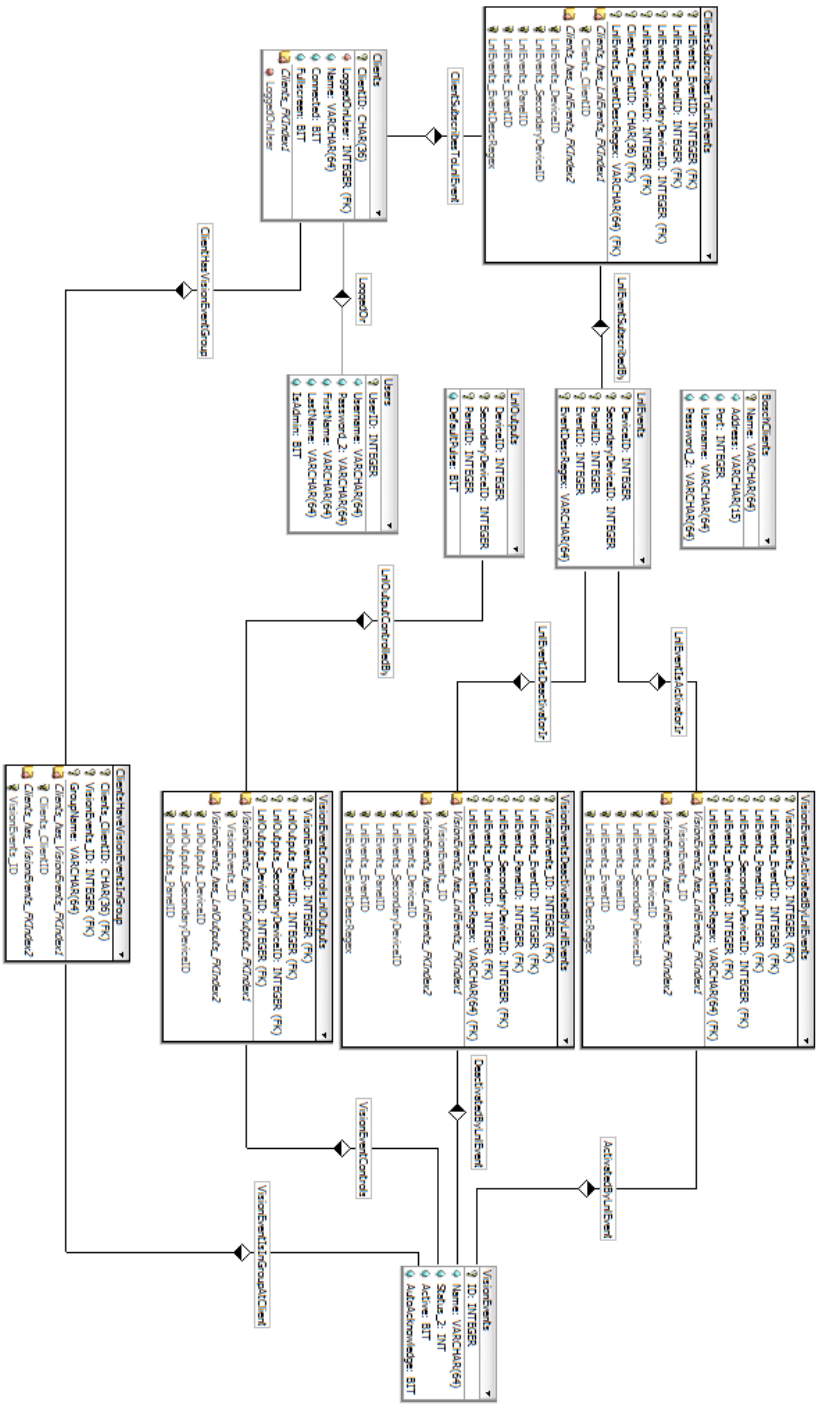


Figure 9: The current design of the database.