MARKET SURVEILLANCE SYSTEM

Surveillance server

Author: Peter Bergenwald
Tutors: Vladimir Vlassov, KTH
Håkan Carlbom, OM
Johan Norén, OM

Master's thesis 2001 for the department of microelectronics and information technology at KTH and for OM

Abstract

This thesis analyses how to make the information in LogB files of OM's CLICK Exchange system available to other systems. The goal is to design and implement an interface from which a client application can obtain real-time and historical data about transactions in the exchange system. during a trading session on a CLICK exchange system the LogB file continuously gets data records appended reflecting events in the exchange system. The information is very detailed real-time data of all transactions taking place during market trading.

The second part of the task specified by OM was an analysis tool (i.e. surveillance client) based on the above interface to enable analysis or view of events in real-time or from stored data. This surveillance client is a part of a thesis written by Johan Örtenblad that will analyse the data from a financial mathematics point of view.

The CLICK system generates a large amount of data to the LogB file during one day of trading. The goal is to be able to have a real-time monitoring of the transactions in the CLICK system and therefore the LogB file. The data from the LogB file needs to be filtered and treated in an efficient way in order to create a scalable and flexible interface to client applications.

During the thesis project a prototype server was developed that implements an XML interface that a client application can connect to using a simple request – response protocol. The prototype supports subscriptions for real-time data and queries for historical data. It uses a database to store historical data used to respond to queries. Real-time data subscribed for by client applications is pushed out whenever they arrive from the CLICK Exchange system.

The performance of the prototype was analysed with data from the Jiway CLICK system. With a realistic configuration of the server it was able to handle around 2300 messages per second from the Exchange system on a NT 4.0 workstation computer with a 730 MHz CPU and 128 MB of memory. With some internal buffering in the server it would be able to handle about 2.5 Gigabytes of data per day. The performance aimed at was 3 Gigabytes per day.

Acknowledgements

I would like to thank my tutors for the help they have given me, Håkan Carlbom and Johan Norén at OM and Vladimir Vlassov at the department of microelectronics and information technology at KTH. I would also like to thank all the people at OM who helped me with different questions.

Table of contents

Introduction 5

1.1 Task definition 5

1.2 OM 7

1.3 Trading and Surveillance 8

1.4 Document outline 9

2 Background 10

2.1 Extensible Markup Language XML 10

2.2 Related master thesis reports 11

2.3 Smarts Surveillance system by Computershare 12

2.4 The CLICK system 13

3 Method 17

3.1 UML 17

3.2 Database design 19

3.3 Technologies used 20

4 Analysis of prerequisites 22

4.1 Information content 22

4.2 Conceptual data model 23

4.3 Surveillance functionality 25

4.4 Data source 25

4.5 Characteristics of data flow 26

4.6 XML interface 29

5 Design of server and interface 31

5.1 System design overview 31

5.2 Server 31

5.3 Interface 39

6 Implementation of server ____ *44*

6.1 Entity classes ____ **44**

6.2 Control classes ____ **45**

6.3 Dynamic description ____ **45**

7 Evaluation of server ____ *49*

7.1 Test method ____ **49**

7.2 Results ____ **49**

8 Conclusions ____ *57*

8.1 Summary ____ **57**

8.2 Performance ____ **57**

8.3 Interface ____ **58**

8.4 Problems encountered ____ **58**

8.5 Future work ____ **59**

9 References ____ *60*

Introduction

This degree project was done for the department of microelectronics and information technology at KTH and for OM. The project is one of two related projects at OM. The projects are closely related since the prototype implementations will communicate. Therefore they have a lot of the analysis and design issues in common. The other project concerns mathematical analysis and modelling of financial data and is written by Johan Örtenblad. The description of the projects follow below.

1.1 Task definition

The thesis work aims to analyse how to make the information in LogB files of OM's CLICK Exchange system available to other systems. The goal is to design and implement an interface from which client applications can obtain real-time and historical data. The logB file is a sequential log reflecting events in the exchange system (a binary log file). It contains very detailed real-time data of all transactions taking place during market trading. The data should be in a format that allows applications connected to the interface to display and analyse the trading on an exchange.

As a second part an analysis tool should be built based on the interface to enable analysis or view and events in real-time or from stored data. The analysis tool will aid in the performance and efficiency evaluation of the solution. The analysis tool will be a part of the parallel thesis written by Johan Örtenblad that will analyse the data from a financial mathematics point of view.

The CLICK system generates a large amount of data to the LogB file during one day of trading. Because of this, the data from the LogB file needs to be filtered and treated in an efficient way in order to create a scalable and flexible interface to client applications. To do this a thorough analysis of which data to be extracted and stored is needed. A support for delivering real-time data to multiple simultaneous clients and re-deliverance of historical data should be developed. The interface should implement a general API based on trading entities rather than exchange specific entities. Therefore an XML based solution will be considered. Performance issues also need to be evaluated because of the large amounts of data.

1.1.1 Functional requirements

- Solution should be divided in two functional parts, one that is an interface to the underlying LogB structure and a second part built on top of the first.
- Solution should address functionality common to surveillance system, so that when a commonly accepted function is not implemented it should be stated why.
- The large amounts of data in the LogB records must be filtered in a way that makes the solution efficient.
- An XML based solution for the interface should be investigated.
- The client application should use a SQL database to store data.
- The internal format of the LogB file is not directly understandable without additional information (for example some enumerates has a textual meaning). The XML interface needs to transform used information to readable texts.

- In the database in part 2 it needs to be decided what should be stored over a longer period. Typically after a given period of time only a subset of the information will be available in an aggregated form.
- It is a need that it is configurable which fields in the LogB file that is viewed through the XML interface. If it is not implemented a plan for how to do this has to be developed.

1.1.2 The OM design idea

The design in figure 1 was the starting point for the analysis work since it was the idea presented by OM.

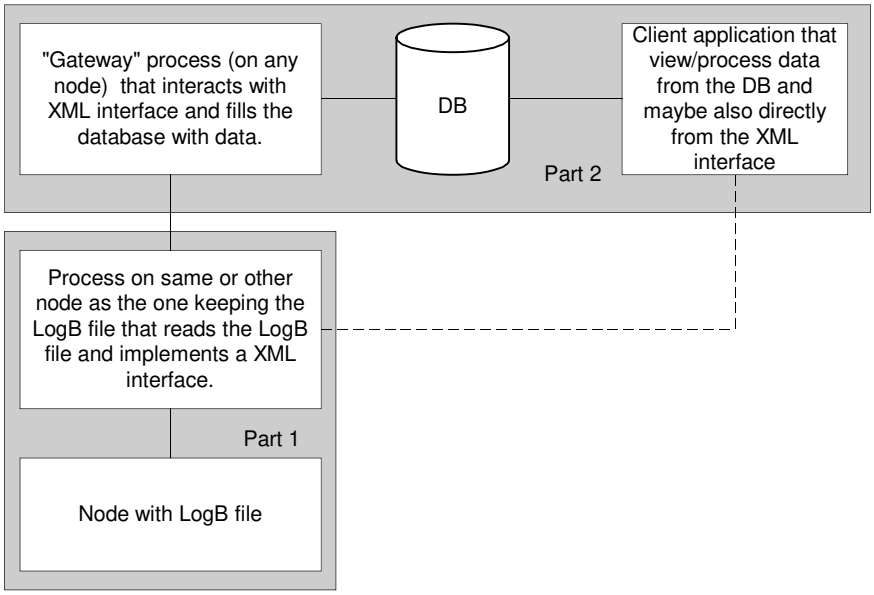


Figure 1, OM's system design idea

Depending on the solution for the XML interface, the Client application can connect directly to the XML interface or only read information from the database. For the XML interface 3 different types of implementations should be investigated (not necessarily implemented).

- Read only all data: Just by connecting to the XML interface all (filtered) data should flow from the LogB file out through the XML interface.
- Request needed information: This basically is an implementation of a classic client/server concept. In other words the connecting process to the XML interface needs to query data before receiving anything.
- Subscription based: The connecting client should post a subscription for the data it is interested in. From that point requested data should asynchronously flow from the XML interface.

Important is also that the client gateway (i.e. the process reading information from the XML interface) is treated as a separate entity since this gateway is likely to be adapted for different solutions (of part 2).

The design of the client and the database (part 2) will depend heavily on design decisions made in part 1. Still the database and client should be able to perform functions common in other

implementations available on the market. In other words part 2 should be treated as a separate product that could get its data from another solution than part 1.

1.1.3 Limitations

The purpose of this work is not to develop a full-fledged surveillance system, rather to develop prototype to illustrate our ideas. The focus is on developing a prototype that allows functionality to be extending, rather than to implement every detail of the whole system.

The GUI development will be limited to what is necessary to illustrate the idea of the system and the possible extensions in functionality.

The prototype will be developed with the Jiway CLICK system as target system and ignore trading messages only used by other CLICK systems.

1.1.4 Expected results

The minimum level is an analysis and design of a system solution together with a pilot prototype implementation of both parts 1 and 2.

The solution should be evaluated from the following standpoints: scalability of the design, extendibility with respect to new functions for trading analysis, the degree of compression of the data needed to perform the analysis, the response time to retrieve information from the LogB structure. Different types of XML interfaces should be discussed and each of these should be compared. Also, the suggested solution should be evaluated from a user point of view.

1.2 OM

OM is a provider of transaction technology, offering integrated and IT solutions to markets around the world. By transaction technology OM means electronic support for handling transaction at an exchange. OM was the world's first for-profit, privately owned derivatives exchange in 1985, and have become the partner and provider of exchange technology to over 24 international exchanges and clearing houses.

OM also owns and operates exchanges in Stockholm, Calgary and London. OM defines its core competence as transaction technology (see Figure 2, OM's business model) – which is derived from Exchange-, Technology-, and Market expertise.

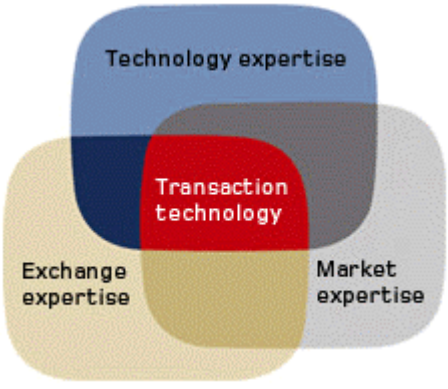


Figure 2, OM's business model

OM's business model involves three levels, and is based on a varying degree of involvement in projects. At the first level, OM develops and provides of solutions to marketplaces or other

interested actors. At the second level, OM is also responsible for the technical operation of an exchange. Finally, OM may take over all or part of the business risk through volume-based contracts, partnership or ownership of a marketplace. [1]

OM has developed two different main exchange systems. First there is the CLICK system, which has been sold to over 21 exchanges world-wide. The other is the recently developed the SAXESS system. It supports a wide variety of market models, e.g. continuous trading, call markets and quote driven markets and is currently used at the Stockholm stock exchange.

The thesis project will work with the JIWAY system, which is a development of the CLICK system to handle trading with instruments listed at different exchanges.

1.3 Trading and Surveillance

Trading in a modern electronic market place generally involves several different systems. A simple model this can be seen in Figure 3. The trading part is the front end application the traders use to receive information from the exchange and submit orders. The exchange system does the actual matching between orders according to the rules at the particular market. The clearing system takes information from all the trades and does the actual exchange. It holds all the accounts of the different actors.

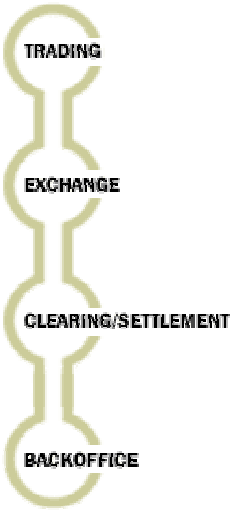


Figure 3, Trading system parts

The concept of market surveillance emerged in the 1980's in the US and Canada. The reason for market surveillance was among others to ensure that the market was fair and efficient. This can roughly be divided into two different categories.

1. Market manipulation
2. Insider trading

Both of the above are signified by heavy trading and price movement in the absence of new market information like press releases.

The general idea of market manipulation is to impose a price on the market by trading in a certain way to increase or decrease the price. By doing this the manipulating party wants to

[1] OM groups homepage, accessed 2001-09-12, available at http://www.omgroup.com

draw attention to the price and make the market think that the traded price is the right price. This changes the rules of supply and demand, which are cornerstones of an efficient market. Market manipulation is ”loud” in the sense that you are trying to convey a price to others. Normally this is done in instruments not heavily traded.

Insider trading on the other hand is trading based on information not publicly available. Therefore the market is not fair since not all parties have the same information. In this type of trading the idea is to trade as “quietly” as possible to be able to use the information as long as possible without others noticing. Typically the type of information can be news about take-overs or other reports that may increase or decrease the prices of an instrument.

Another reason for market surveillance was to preserve the trust and credibility of an exchange and its rules. [2]

1.4 Document outline

The document is structured as follows; chapter 2 describes the background with some related work either in the same area or in different areas but with interesting technical aspects. The basics of the CLICK trading system is also described as it is the target system for the prototype implementation in this work. Chapter 3 describes the method and technologies used during the work with the thesis. Chapter 4 contains an analysis of different aspects concerning the server such as data characteristics and client surveillance functionality to support. The next chapter describes the design of the server. For the full description of the client design see the report by Johan Örtenblad. Chapter 6 describes the implementation of the prototype whereas the following chapter evaluates the prototype. Conclusions and future work are found in chapter 8. Glossary is found at the end of the report just before the appendices.

[2] M J Aitken and J H Berry, July 1991, *Market surveillance at the Australian Stock Exchange*[article online], accessed 2001-09-18, available at http://www.smarts.com.au/surveill.html

2 Background

This chapter describes related work and a general outline of the CLICK trading system used as target system. The related work section introduces the basics of XML because a solution using XML will be considered as stated in the task definition by OM. Two other master's thesis reports are discussed. One focuses on XML as a communication protocol and contains an interesting survey of different initiatives for general XML protocols. The other uses a technical design interesting for this thesis even though it is aimed at a totally different target system. Finally the only encountered commercial market surveillance system is described. It is the only commercial product in the sense that it is sold as separate product. Most exchanges have some kind of surveillance systems but these are only used at the specific exchange.

2.1 Extensible Markup Language XML

XML is a text based markup language used to format and transfer data in a simple and consistent way. XML will here be described very briefly, focused on the areas interesting to this thesis.

XML is a subset of the much more complex Standard Generalised Markup Language SGML just as the better known HTML[3] is. In order to understand XML, an understanding of HTML is helpful. HTML is a markup language based on a set of predefined tags in the format:

<Header>This is some text</Header>

In the example the string "This is some text" is marked as the type "Header". There are two problems with HTML that constrains the use of it. First it is not extendable, you can only use the predefined tags if a web browser is supposed to understand the content. HTML also lacks possibilities to describe the type of data it transfers.

XML is an approach to solve these problems. XML is extensible in the meaning that anyone can define new tags to markup data in order to describe the type of data. XML is more focused on data description than presentation but can also be used to describe the presentation of data[4]. The tags used in XML can be any text but has to adhere to certain rules. There are certain character sequences that are reserved and rules governing how the tags are used. If a document follows these rules it is said to be well formed.

The basics are very simple, the document begins with a processing instruction saying that the data is an XML document. A tag defining the beginning of the XML document follows the processing instruction. Then the tags defining the data follow in the form of a start tag and end tag enclosing the data. Finally the document is ended with the document end tag. Below is an example of a very simple XML document:

[3] HTML, Hyper Text Markup Language

[4] World Wide Web Consortium (W3C), *XML in ten points,* web page accessed 2001-12-17, available at http://www.w3.org/XML/1999/XML-in-10-points.html

```
<?XML version="1.0" ?>
<OrderDoc>
        <Goods > ball </Goods>
        <Quantity> 4 </Quantity>
        <Customer> George </Customer>
</OrderDoc>
```

The data descriptions can be made to describe much more advanced data structures by nestling tags in different ways. There are an endless number of ways to form XML documents and they are said to be well formed as long as they follow the rules set up by the standard[5].

The question that arises is how will different applications interpret an XML document if anyone can create their own tags. The answer for XML is to define a way of describing how the data in an XML document is structured. There are two standards for this at the moment. The one that emerged first was the Document Type Definition (DTD). The DTD have a set of predefined words that are used to define the format of an XML document. It basically allows you to define what elements will be present in the document, their ordering and attributes. An element is something defined by a tag pair and an attribute describes an element.[6]

A more versatile and recent standard is the XML Schema[7]. In a schema, there are richer ways of describing the content of an XML document, but it is at the same time somewhat more complex. A document that follows a certain DTD or Schema is said to be valid in addition to being well formed.

For further information about XML study the homepage of the Word Wide Web Consortium (W3C) at www.w3c.org.

2.2 Related master thesis reports

The masters' thesis "A study of an XML Protocol for Manufacturing Systems using XML and Java"[8] investigates how to use XML to design a general protocol for communicating with manufacturing machines. The thesis report thoroughly discusses the use of XML and contains a survey of different open source efforts to create XML protocols.

A good metaphor for XML in the report is:

> "XML is like putting important items into a set of boxes, putting descriptive labels to them, and placing them in a nested logical pattern to make it easy for a box-searcher to find all the items and use them. The "set of boxes" is the XML document and the "box-searcher" is the program parsing and processing the content of it"

[5] World Wide Web Consortium (W3C), *The XML 1.0 Specification*, web page accessed 2001-12-17, available at http://www.w3.org/TR/2000/REC-xml-20001006,

[6] Steven Holzner, *XML Complete*, 1998, McGraw - Hill

[7] World Wide Web Consortium (W3C), *The XML 1.0 Specification*, web page accessed 2001-12-17, available at http://www.w3.org/TR/xmlschema-0/

[8] Astrid von Euler-Chelpin, *A study of an XML Protocol for Manufacturing Systems using XML and Java,* June 2001, IT department, KTH.

In the conclusion from the survey of XML protocol efforts the report states that the efforts still are rather immature. This may have changed since the report was written but the more important conclusion is that using the general approaches greatly increases the complexity of the XML format. The reason for this is that they are designed to be used in many situations. [9]

The thesis also compares two XML parsing approaches, SAX and DOM. It concludes that for small XML documents the DOM standard is easier to use. For large documents SAX can be the only alternative since the memory usage does not increase with the size of the document and parsing can be done before all of the document has been loaded. With the amounts of data in the CLICK system and considering that the information arrives continuously in real-time a SAX approach seems to be the better alternative for a surveillance system.

The masters' thesis "Call Traffic Monitoring system" describes a system that handles information about incoming phone calls to a telephone exchange. This is a very different target system from the CLICK Exchange system. There are however several technical and design similarities between the traffic logging unit in the thesis and the server implementing the XML interface in this work.

The traffic logging unit (TLU) in the Call Traffic Monitoring system is a multithreaded application handling transaction messages generated by events in other systems. It deals with concurrency and performance issues. It also uses a database to save information about transactions.

Concurrency in the TLU is handled through thread safe queues and the threads are run by thread objects encapsulation the thread functionality. The implementation there is heavily based on the windows platform and the Microsoft foundation classes.

Two main differences between the TLU and the server in this work is, first of all that OM has requested that the prototype is as independent of the Microsoft windows platform as possible.

The performance issues in the call monitoring system were also somewhat less than for a market surveillance system. The TLU performance of about 60 000 messages per day was judged to be more than enough. [10]

2.3 Smarts Surveillance system by Computershare

SMARTS (Securities Markets Automated Research Trading Surveillance) is a generic surveillance system designed to run on various platforms, and with various kinds of financial markets. It was originally developed by a group of researchers at the University of Sydney, but today it is developed and marketed by Computershare Limited. The system is made up of several different modules, each providing its own functionality. Among these are modules for benchmark visualisation, alerts triggering and post-trigger analysis, statistical reporting, real-time statistical monitoring and a market replay function.

The system was designed more like a general tool for financial market surveillance than with a specific exchange, or exchange system, in mind. Therefore, it is very versatile, at the same time as its complexity probably does demand some learning investments. When properly used, however, it can be very powerful according to the Computershare web page. Among other things, it features a purpose-defined definition language, allowing the user to define and use

[9] Astrid von Euler-Chelpin, *A study of an XML Protocol for Manufacturing Systems using XML and Java,* Chapter 3.5, June 2001, IT department, KTH.

[10] Andreas Oderstad, *Call Traffic Monitoring System*, March 2000, IT department, KTH

its own benchmarks and alerts by the use of a collection of predefined metrics. This solution allows the user on an individual exchange to interactively gain an understanding of the market prerequisites prevailing at that particular market, and fine-tune alerts that are triggered not too often, nor too seldom, and that produce relevant information.

Today, an installation of the SMARTS system is used, for example, on the Oslo stock exchange.[11]

2.4 The CLICK system

2.4.1 General

The CLICK system consists of two main parts, the actual exchange system and the front end trading application. This thesis only concerns the exchange system and therefore the trading application will be described only briefly.

The exchange system, which is the core of the CLICK system, is made up of several different processes that run on one or several computers. See Figure 4.

The main parts of interest in the CLICK system are:

OMNet	The network for the communication in the CLICK system
Information dissemination	This process disseminates information about events in the market place to all trading parties on the OMNet.
Database	Contains information about instruments and customers on the market place, and other information necessary to run the exchange.
Integrated Market Place	This is the process where orders are matched. This process receives orders and quotes and matches them to create trades. It calculates the current order book according to the market rules at all times. It also writes information about events on the exchange to a log file.
Surveillance module	This module is designed to allow market surveillance staff to retrieve information about the trading from a limited set of queries. The aim of this work and the parallel thesis by Johan Örtenblad is to develop a client server system that includes the surveillance functionalities found in this module and extend them.
Log file reader	This process reads from the log file extracting information according to queries from the surveillance module. It also implements a TCP interface that allows clients receive log data.
CLICK Trade	It is basically a GUI application that lets traders place orders and receive information from the market.

[11] SMARTS homepage, accessed 2001-09-17, available at http://www.smarts.com.au

The exchange system is based on orders. Orders to either sell or buy an asset traded at the exchange to a certain price and volume When the exchange is in a trading session traders submit orders to the. These orders are received and traded when a matching order is found in the order book (the list of not traded orders) or a new matching order comes in. An order is removed from the market when it has been traded fully (the remaining volume is zero) or if it is cancelled for some reason. The exchange also sends out information to the traders about the status of the market like the current best bid and offer.

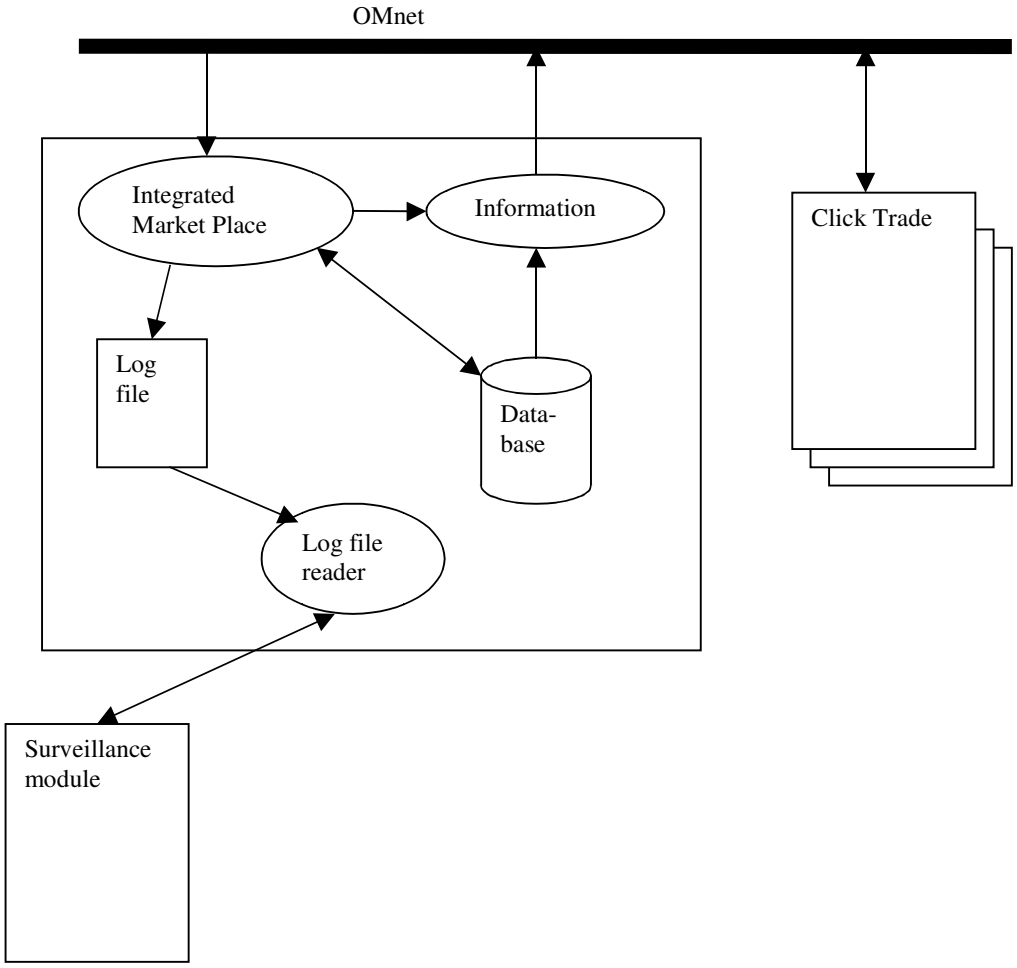


Figure 4, The CLICK system

2.4.2 Supervision in the CLICK system

LM (Local Module) is an integrated part of the CLICK system that, among other things, acts as a primitive market surveillance agent. It is strongly CLICK-specific, and is therefore only used on exchanges that run this exchange system (although not all such exchanges have incorporated LM into their CLICK system). LM checks for the following events, and produces an alert if one of them is detected:

- Internal trade – a member that trades with himself.

- Large trade – detects trades that are larger than normally. The definition of this event uses the normal-state (historical) variance.
- Trade price movements – a member that sells and buys in steps, to move the trade price in a certain direction (without the net ownership being altered in the end).
- Unintended crossing – deals that occur outside of the spread. This means that the buyer will pay more than the seller who will sell for the lowest price requires.

A surveillance system as simple as this one clearly has its limitations – one of the reasons why my and Johan Örtenblads's study was conducted in the first place. However, it represents a possibility to at least carry out rudimentary market surveillance. As it is tightly coupled with the CLICK system, it is not easily portable to other exchange systems.[12]

2.4.3 The CLICK transaction log

The matching process generates log information about transactions in the CLICK systems market place process. This log information is written to one or several logB files. The logB file is a sequential log reflecting events in the exchange system (a binary log file). It contains very detailed real-time data of all transactions taking place during market trading i.e. all orders, order changes and trades etc. The information available from the JIWAY target system will be discussed in detail in chapter 4.1 and 4.2. The trading information in the log file is in the form of binary C structures. The information in the C structures is in the internal format used by the CLICK system which often means numerical codes. These numerical codes have to be mapped to an understandable data format..

The amount of data written to the logB files can be very large. Some CLICK systems generate up to 10 Gigabytes of log data per day while 3 Gigabytes is a normal load in many systems[13]. However, the test data received were only about 100 MB. The size of the log messages vary between around 50 to 250 bytes in size. Due to the fact that most of the messages generated by the system are small and assuming that the test data is representative it turns out that the average message size is only slightly larger than 50 bytes. This mean that the system will generates a bit less than 20 000 messages per MB of data in the log file.

2.4.4 Restatement of task

The thesis work aims to analyse how to make the information in LogB files of OM's CLICK Exchange system available to other systems. The goal is to design and implement an interface from which client applications can obtain real-time and historical data. An XML based solution and three different interface types should be investigated (not necessarily implemented).

- Read only all data: Just by connecting to the XML interface all (filtered) data should flow from the LogB file out through the XML interface.
- Request needed information: This basically is an implementation of a classic client/server concept. In other words the connecting process to the XML interface needs to query data before receiving anything.

[12] PerÅke Sundström, Mats Danielsson and Håkan Carlbom, Interview 20010917 at OM

[13] Meeting with Johan Norén, OM, 10 September 2001.

- Subscription based: The connecting client should post a subscription for the data it is interested in. From that point requested data should asynchronously flow from the XML interface.

The solution should provide a way to configure and filter the data sent to clients in order to make the interface flexible and efficient.

3 Method

The methods described below were used during various parts of the design and analysis of the surveillance system. The Unified Modelling Language was used in the early design work.

The database design was used to make a model of the log data from the trading system and then translate this to a physical database. It also helped when analysing what was essential information and what could be discarded on general level and not only for the database.

3.1 UML

Unified Modelling Language (UML) is commonly used in object oriented software development. UML is a graphical language for visualising and specifying a software system. Below is a description of different domains that UML has semantics to describe. In the thesis work the UML usage has been limited to the use case model, the interaction model and the logical model since these are sufficient for this work. The dynamic model, the component model and the physical model were not used.[14]

3.1.1 Use case model

This model describes the functionality of the system. It uses the semantics of actors interacting with the system. It includes a general description of the use case as well as requirements and constraints concerning the use case. A system normally has several use cases that describe different functionality in the system. The use case model is used to make the system understandable on a high level and define how it will be used. For an example of a use case see the figure below.

Figure 5, UML use case example.

To describe the events of a use case the interaction or collaboration model can be used

3.1.2 Interaction / collaboration model

The interaction model describes how the different objects in the system interact to realise a use case. The model uses either sequence diagrams or collaboration diagrams to describe the flow of events and messages in chronological order in a use case. They are the base for the order of execution in the implementation. In the figure the below is an example of a sequence diagram, collaboration diagrams describe the same type of event flows. The example below illustrates the flow of messages when a client connects to a server.

[14] UML Tutorial – part 1, web page accessed 2001-09-14, available at www.sparxsystems.com.au/UML_Tutorial.htm

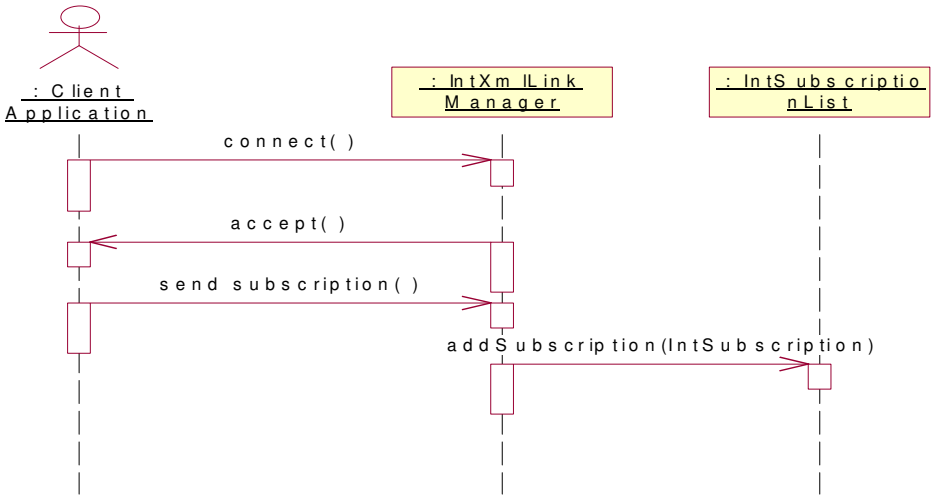


Figure 6, Sequence diagram example

3.1.3 Logical model

This model defines the classes in the design the objects are created from at runtime. The classes are described with the their attributes and their methods used to manipulate them. The classes are also categorised in different types. Entity classes represent some type of entity, i.e. they contain data defining the entity. Control classes govern the sequence of which events will take place in the system designed. This is the most classic part of object oriented design. The relationships between different classes are also described, for example aggregation of classes and classes derived from other classes. An example of a simple class diagram is shown below.

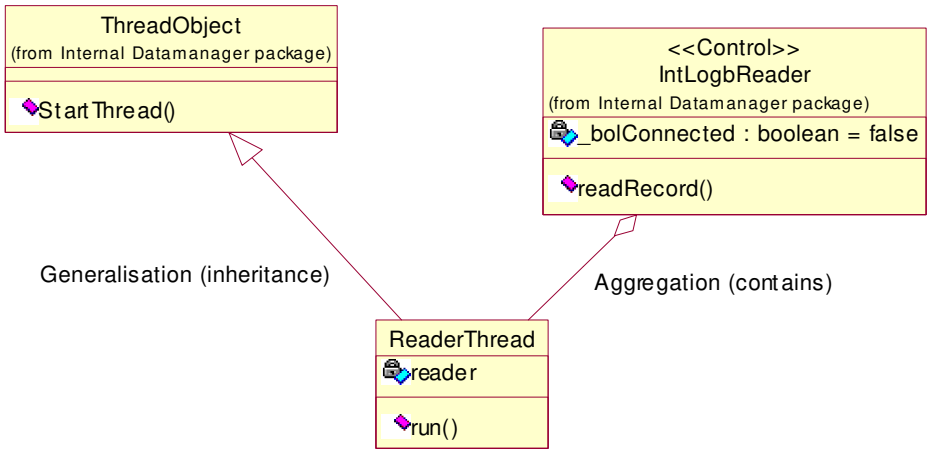


Figure 7, Class diagram example

3.2 Database design

The database design should in theory consist of three phases, conceptual design, logical design, and physical design[15]. The conceptual design should have no connection to the type of database used in the final application, it should only reflect the structure of the data. The logical design should take into account the database type used in the end but should still be focused on the structures of the data. The physical design is the implementation of the logical design in one type of database. Entity-relationship models and normalisation are used in both conceptual and logical design and here there is no difference made between conceptual design and logical design. The physical design describes the translation of a logical design to a relational database.

3.2.1 Entity – Relationship modelling

The basic parts of an Entity-relationship model are:

1. Entity types
   The concept of an entity type is an object that can exist independently. It may be a physical or abstract object that is fully defined by itself. For example ”address” may be an entity type.
2. Entity
   An entity is an instance of an entity type, in the example before a particular address.
3. Attributes

   Attributes define the particular properties of an entity, in the address case street name, zip code etc. One or a group of attributes defines the key for the entity, i.e. it uniquely identifies the entity.
4. Relationship types

   Relationship types define a relationship between entities. The person entity ”lives at” address. Here the relationship type is ”lives at”.

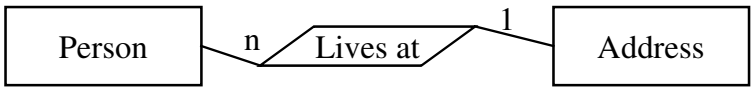


Figure 8, Example entity -relationship model

The numbers in the model define how many instances of each entity type can participate in the relationship. In this case several persons can live a one address but one person can not have more than one address.

3.2.2 Normalisation

Normalisation is a formal method for creating relations with desirable properties with respect to the requirement of the data they relate. Basically it aims at removing redundancies in the data and unwanted dependencies within the data. Normalisation is a method applied to relational database designs and is in part based on the concept of dependencies. For information about the definition of different dependencies see the book *Database Systems*[16] by Connolly and Begg. There are three normal forms that are the most commonly used:

[15] Connolly and Begg, 1998, *Database systems*, Addison-Wesley

[16] Connolly and Begg, 1998, *Database systems*, Addison-Wesley

1. First Normal Form
   All attributes are atomic values
2. Second Normal Form
   A relation is in second normal form if it is in first normal form and every non-primary key attribute if fully functionally dependent of the primary key.
3. Third Normal Form
   The relation is in second normal form and there are no transitive dependencies between primary key and no primary key attributes.

3.2.3 Physical design

In the physical design the logical design has been translated, in this case to a relational database design. In a relational database the tables containing data are called relations. Briefly this is done in 3 steps:

1. Convert entities to relations
   Every entity type is converted to a relation with key attributes as key in the relation.
2. Convert 1-n associations to foreign key attributes in the relations

   For every entity with a 1-n association add a foreign key attribute consisting of the primary key of the associated entity.

3. Convert n-m associations to separate relations

   For every n-m association add a relation containing the two primary keys of the entities. This should not be necessary if the Entity-Relationship model is well done.

An example is shown in Figure 9, Relations in a relational database.

After this step, the use of the database should be considered to see what indexes should be created and if it should be de-normalised in order to enhance performance.

Person
Person key
Attr.
Attr.
Attr.
Address key

Address
Address key
Attr.
Attr.
Attr.
Attr.

Figure 9, Relations in a relational database

3.3 Technologies used

During the development and implementation of this system several different technologies have been used. The reasons for using them have varied and will be stated below.

The server implementation is written in C++ with the Visual studio tool from Microsoft. C++ was chosen because a higher performance was likely to be achieved with C++ than for example Java. Since the CLICK system is written in C/C++ this choice also facilitates the incorporating of the work into the existing system.

Two different database management systems were used, in the beginning MS Access 97 was used, but it was quite slow. Instead MySQL was used in the final prototype. There are other systems that could be suitable but both MS Access and MySQL were available free of charge.

The server uses ODBC for database connection. As Microsoft windows is not the only platform used at OM it was requested that the prototype be implemented with a minimal set of windows specific code. STL[17] classes have been used to a large extent instead of the Microsoft Foundation classes equivalent. Exceptions were made in the use of sockets, database connection and thread handling.

The different applications in this work will use sockets to communicate since this is very standardised and works well between different platforms.

The client application was developed in Java using two different development environments. The Forte for Java environment from SUN was rejected after a while in favour of Jbuilder5 from Borland. The reason for this was the extreme amount of system resources used by Forte. The reason for using Java in the client were that the XML support is much better in Java and it seemed interesting to develop in Java as well as C/C++.

The database connection in the client is JDBC and the GUI uses swing components.

For the design of the system started out in the Rational Rose 98 tool, after a while this tool was used only for illustrating the high level design. The hassle of using the tool outweighed the benefits from it when it came to generating code from the design.

[17] Standard Template Library, A library of platform independent classes commonly used.

4 Analysis of prerequisites

This section analyses different aspects that impact on the design of mainly the server. Some are interesting also for the client application but Johan Örtenblad discusses these in his report. A large part of the analysis concerns the information from the trading system that the server has to handle. General content, quantity of data and characteristics are analysed. The client surveillance functionality is also discussed since it is important for what information the server has to transfer. Aspects of the interface are also analysed.

4.1 Information content

The information content and quantity from a trading system depends heavily on the type of market it is used on. In this case stocks are traded on the market. There are other markets where for example commodities like metals or derivatives are traded.

The Jiway CLICK system generates trade information messages that, to a large part correspond to the terminology used in trading at an exchange[18]. Information messages that may be valid also for other electronic market places are shown in Table 1.

Table 1, Information from the trading system

Message	Explanation
Orders	An order to buy or sell a quantity of a certain asset to a specific price.
Quotes	A quote is a type of order with both buy and sell side sent by the market maker who is responsible for making sure that there always are both buy and sell orders on the market.
Trades	A trade between two orders matched according to trading rules on the exchange.
Order book	The order book is a list of all the orders that have not yet been traded fully. In the CLICK system this list is given by information about order book changes.
Best Bid Offer (BBO)	Record with information about the currently best sell and buy price and their respective quantities. This information is also available from the order book.

The Jiway CLICK system also generates information messages that are quite specific to Jiway's specific business model. The Jiway market place only trades with assets also listed at other exchanges and the Jiway market guarantees that the price of an asset traded at Jiway is not worse than at any other exchange where the asset is listed. Additional information messages due to this are shown in Table 2.

[18] Meeting with Mats Danielsson, OM, 12 October 2001.

Table 2, Jiway specific information

Message	Explanation
External Best Bid Offer	BBO at the away market the asset is also listed.
Hedge Order	Order that is sent to be traded at the away market because the BBO at the away market is better than at Jiway.
Hedge Execution Report	Report of a successful hedge where the original hedge order has been traded at the away market in part or fully.
Price Improvement Quote	Record with information that a market maker will trade a certain volume of an asset at the market place to the price specified by the external BBO. This is done if the external BBO is better than the internal BBO but the market maker wants to trade rather than sending a hedge order to the away market.

There are other messages coming from the Jiway trading system that are not interesting for this work since the do not concern surveillance of the market. There are also messages not used by the CLICK system but used in other CLICK systems and are common in other trading systems as well that are not included here. An example of such information would be combination orders, they are interesting from a surveillance point of view but are not included in the scope of this project due to the limited time.

4.2 Conceptual data model

To make a model of the data in the Jiway CLICK system the Entity –relationship model described above was used. The model is influenced by the way trading is seen with an automatic matching process system. The data in the model will also reflect the somewhat special business model of Jiway. The idea is to start with the most general entity types and then to add on entity types and attributes specific to this system.

Basic entity types are presented in Table 3.

Table 3, Entity types

Entity type	Attributes
Order	• Price • Quantity • Asset • Entry time • Validity time • Type (bid or ask)
Trade	• Price • Quantity • Trade time
Order change	• Time stamp • Reason • Quantity remaining

Customer (places orders)	• Customer identity

Quotes in the table are treated as orders conceptually even though the information in the actual messages from the trading system differs somewhat. Other entity types more specific to the Jiway market are:

- Hedge Order
- Hedge Execution
- Price improvement quote

A simplified Entity – relationship model based on this is shown in the figure below.

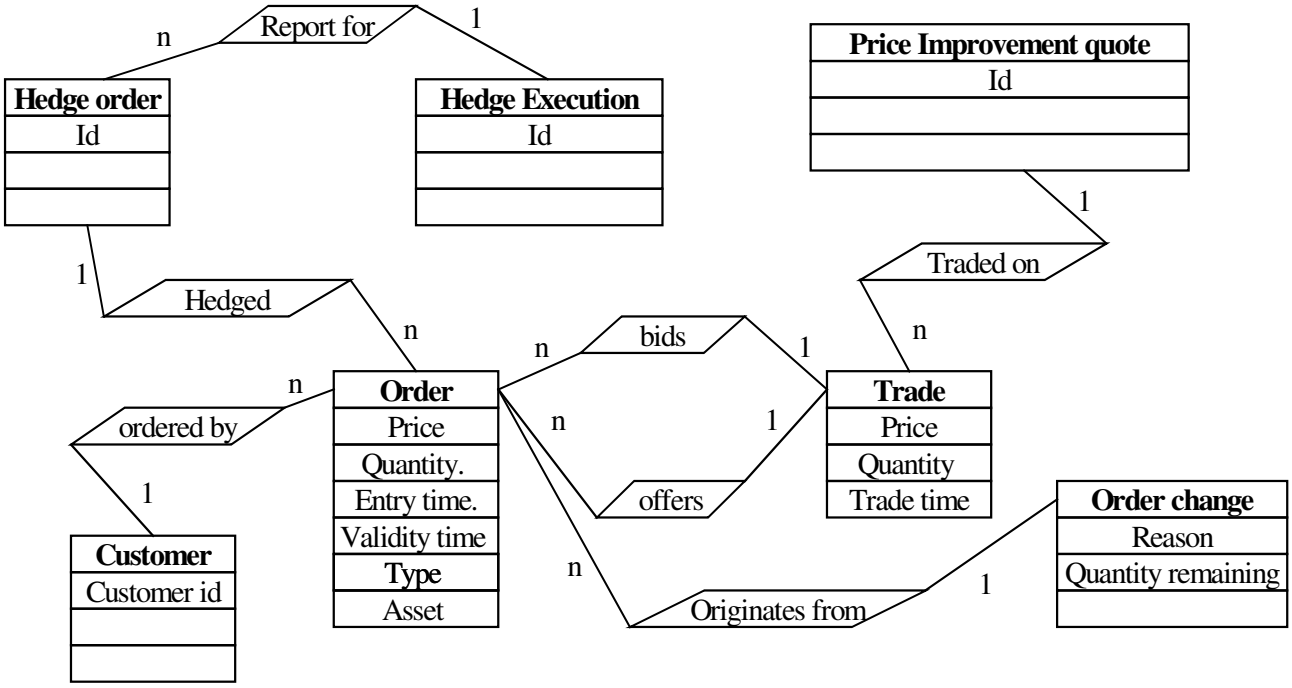


Figure 10, Entity-relationship model for trading data

In the final model there were changes made depending on the data available for the trading system and the structure of the trading system information. However, these changes do not make the model more understandable and are therefore not shown.

4.3 Surveillance functionality

The surveillance functionality of the system is more thoroughly described in the thesis by Johan Örtenblad. In short the client will monitor the variables in Table 4.

Table 4, Monitoring variables

Variable	Explanation
Volume	Monitors the difference between the volume traded in an asset during a certain T-length time period and the normal volumes.
Return	Monitors returns to see if the return on an asset suddenly leaps too high or too low as compared to the historic mean and standard deviation.
Order book	The watches whether the current order book has a "suspicious look" (not counting individual orders that might be suspicious in themselves) or not. In this, the current order book is compared to the current aggregated order book of historical data.
Spread	Measured either on the bid, ask or ask-bid In a similar way as for return and volume, the current spread is compared to the historic, and an alert is set off if the spread is too large.
Beneficial ownership	Each time a T-length time period comes to an end, the aggregated trades for each client and asset is calculated. If there has occurred trading at all (above some threshold value), the trigger is set off if the aggregation is significantly near zero (as indicated by the user). This technique captures not only trading with oneself, but also circulatory trading. As an add-on, a list of clients that have the same beneficial owner should be possible to provide to the system.
Suspicious orders/trades	New trades and orders are compared to the order book metric, to see if they lie too far off the BBO, in a way similar to the volume- or returns technique. Alerted orders might be suspected of price manipulation, trades of pre-agreed trading. The user may indicate a minimum volume for the order or trade for it to trigger an alert.
Immediately traded orders	Also detects pre-agreed-upon trading. Here the incoming trade is compared to a recent order book, and an alert is triggered if the trade was not represented in the order book (and is significantly large as indicated by the user).

4.4 Data source

The matching process generates the log information about transactions in the CLICK system. This information is available from a file or from a TCP Socket connection. To be able to access the information in the file the reading process has to run on the same node as the

matching process. If the information is accessed from the TCP socket instead the reading process can be run on a different machine without the risk of degrading the performance of the matching process since there already exists a process that writes logB information to a TCP socket.

The trading information in the log file is in the form of binary C structures. The data in the structures are written to the file sequentially. It is written and pipelined to clients in real-time. The client in this case would be the server in the thesis project.

If a solution where the server reads directly from the log file is chosen, the advantage is that the server may be slower than the actual rate of the trading information with no risk of losing information. The server just reads as fast as it can when reading the file sequentially. However, this may as mentioned degrade the performance of the matching process.

On the other hand, if the server reads information from the TCP Socket it has to be as fast as the rate of the trading information on average. Otherwise the send queue in the socket connection will overflow. The advantage is that the server may run on a separate node. However, if the system is supposed to provide more or less real-time data from the CLICK system, it has to be able handle the at least the average information rate of the matching process.

This is given by classic queuing theory for a stationary system. If λ is defined as the average rate of arrivals to the system and $1/\mu$ is the average processing time in the system[19] then the equation that has to be valid for a stationary system is given by:

$$\frac{\lambda}{\mu} \leq 1$$

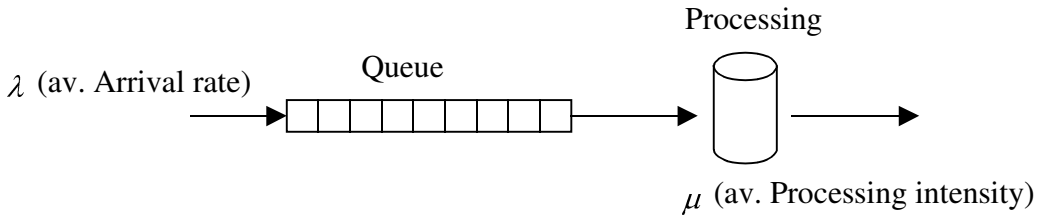


Figure 11, Simple queuing system example

If this relationship does not hold, the queue size will run toward infinity and the time spent in the queue for each message will also go toward infinity.

The server will depend on how data arrives and how it is processed, i.e. what functions λ and μ follow. These functions (or estimates of these functions) will give hints about how to design internal queues in the system to be able to handle variation in the momentaneous arrival rate of information.

4.5 Characteristics of data flow

The data flow from an exchange does not follow any simple arrival process. It is natural to assume that the intensity of arrivals depend on several different parameters of which some are difficult approximate. Some variables likely to have an influence include :

[19] Jan Grandell, *Köteori 2:a utgåvan*, 1997 , Institutionen för Matematik, KTH

- The number of instruments traded on the market
- The time of the trading session
- The announcement of news

The announcement of news regarding a company that might affect the activity on a market place and thus the amount of information generated by the exchange system. The effect and timing of news affecting the market are naturally difficult to model. However, it should be possible to create a model with the average amount of information generated by the exchange system depending on the number of instruments on the market and the time of the day through historical analysis. This is assuming that the trading pattern does not differ too much between trading days.

Simple queuing systems are often assumed to be M/M/1 systems which means that the arrivals and the processing in the system follow exponential functions and that there is only one processing unit. For M/M/1 systems it is possible to calculate several aspects of the system when it has reached a stationary state. In this work the average queue size and waiting time will be interesting. However, a simple assumption about exponential arrivals will be misleading because of the reasons stated above. Instead a way of approximating a M/M/1 system during certain periods will be used. The general idea of the analysis for the buffering needed by the server is:

- Assume exponential processing time for all messages with intensity μ
- Assume exponential arrival for every time interval, intensity λ
- Divide into intervals where the arrivals are approximately exponential and calculate their respective intensities.

The average intensity of arrivals for a whole day is given by:

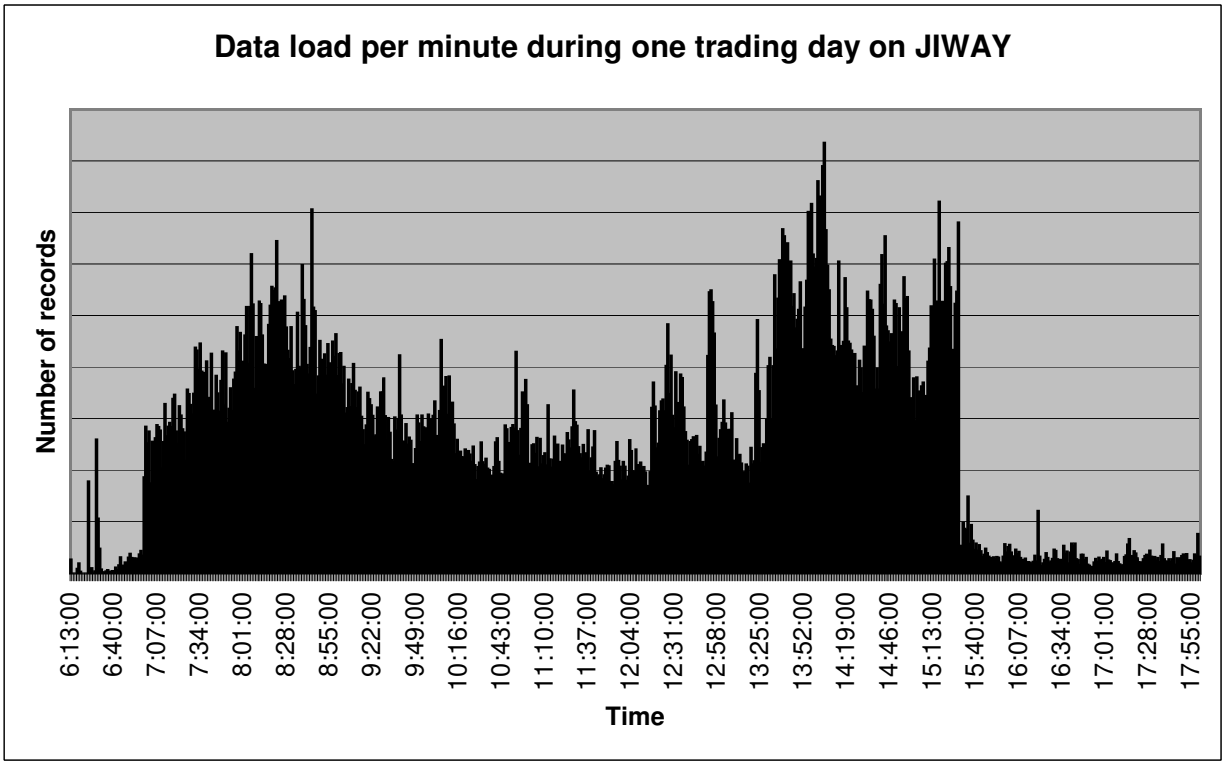
With a load of 3 Gigabytes per day and an 8 h trading day means an average rate of about 100 Kb of data/sec and an average message rate of around 1900 messages/ sec.

With a load of 10 Gigabytes per day and an 8 h trading day means an average rate of about 350 Kb of data/sec and an average message rate of around 6300 messages/ sec.

This gives that the average processing intensity, μ, has to be > 1900 messages/sec or > 100 Kb/sec for a 3 Gigabyte load and >6300 message/sec or >350 Kb/sec for a 10 Gigabyte load for the equation shown above to hold.

These values for the processing intensity, μ, will be used in the evaluation as lower and upper bounds for the server. This will help to show if the prototype implementation of the server is a feasible solution.

To get a better idea about the intensity of transaction messages during a trading session, data from a sample trading session have been used. The intensity of arrivals of transaction messages has first been aggregated into minute intervals to give a detailed view of the arrivals. The result is graphed below. The data available to us in the thesis work was less than 3 Gigabytes. However, it is assumed that the data is representative for higher intensities as well. The assumption is that the relative change of intensity levels will be the same also for higher actual levels of intensity.



The data used to create the graph was 100 MB in size and contained 1,87 million messages. The trading session time was 12 hours. This gives an average rate of 2Kb/sec or 43 messages/sec. However, when inspecting the graph the intensities at the tails are much lower and therefore make the average over the whole trading session less useful. If data in the period from 7:00 to 15:30 only is used the tails are cut off. Then there are 1,80 million messages and a session time of 8 h and 30 min, which gives an average rate of 59 messages/sec. The maximum number of messages during 1 minute was 8348 which gives a peak rate of 139 messages/sec

Table 5, Transaction message rates, full day

	Number of messages	Time (s)	Rate (msg./sec)
Average 12h	1868437	43200	43,25086
Average 7 – 15:30	1803849	30600	58,94931
Peak	8348	60	139,1333

This gives a ratio of 2,4 between peak rate and average rate. The overall shape of the graph is also very interesting. By inspecting the graph it is possible to deduce a period in the beginning and one in the end of two hours with significantly higher rates. The result of splitting data into the periods 7 – 9, 9 – 13:30, and 13:30 – 15:30 is shown in Table 6, Transaction message rate, periods.

Table 6, Transaction message rate, periods

	Number of messages	Time (s)	Rate (msg./sec)
Average 7-9	475028	7200	65,97611
Average 9-13:30	730017	16200	45,06278
Average 13:30-15:30	598804	7200	83,16722
Average 7-15:30	1803849	30600	58,94931

Assuming that this data is representative a 3-Gigabyte log file would give a scale factor of 30 to the calculated rates. For a that amount of data the result would be:

Table 7, Transaction message rates, scale factor 30

	Number of messages	Time (s)	Rate (msg./sec)
Average 7-9	14250840	7200	1979,283
Average 9-13:30	21900510	16200	1351,883
Average 13:30-15:30	17964120	7200	2495,017
Average 7-15:30	54115470	30600	1768,479

The analysis of the buffering necessary in the server will be done in the evaluation chapter where the actual performance of the server is analysed.

4.6 XML interface

It is reasonable to assume that the conversion of the data generated by the trading system to an XML format will increase the size of data. There are several reasons why the conversion will increase the size of data:

- XML is a textual format and thus not always as efficient with regards to memory as binary integer representations when representing numbers for example.
- XML also marks up all of its content with tags that will increase the size of data also. The data from the trading system is in the form of structures where a piece of data is marked by its exact position in the structure.
- The C structures from the trading system many small pieces of data in number formats.

Because of this, it has been necessary to use as short XML tags as possible in the XML interface. An example calculation illustrates this:

Assume that a long integer in a C structure defines the for example the price of some asset. A long integer takes up 4 bytes of memory and no memory to be identified. The maximum value of a long integer takes up 10 characters in base 10 representation, which equals 10 bytes of memory with normal text encoding of one byte per character. If the tags <price>, </price> are used to mark up the data that adds on another 15 characters. In total that means that the 4-byte data may use 25 bytes in XML representation and never less than 16 bytes.

This would mean an increase in size from 4 to 6 times for this data. If this ratio is assumed to valid for all data in the C structures it means that the information from a 10 Gigabytes log file

also could expand 4 to 6 times when expressed in an XML format. That amount of data may be difficult or at least costly to handle.

If the tag size is restricted to two characters instead the size increase ratio is somewhat more favourable. Two character tags also keeps a lot of flexibility with 26 * 26 combinations possible if the ordinary English alphabet is used and case is ignored.

It still means up to 10 characters for the value but the tags will be only 9 characters.

This gives a size increase ratio of 2,5 to 4,75.

This is still a significant increase in size, but compressing the XML data before sending it to the client could reduce the size further. However, compression only helps if it is the speed of the connection that is a problem. If on the other hand the conversion to XML is slow and CPU demanding, compression will only increase the workload for the processor.

It could be interesting to use a general XML protocol for the communication, for example XML protocol by the World Wide Web Consortium that uses SOAP 1.2[20]. SOAP stand for Simple Object Access Protocol and defines a standard for accessing remote object over a network. The problem with general approaches is that they quickly become very complex and the message overhead becomes large. In this work both time to implement and the amount of data are critical. Because of this a simpler protocol developed from scratch will be used including only what is necessary for the prototype.

To define the format of the XML interface there are two options either a Document Type Definition (DTD) can be used or an XML schema. The DTD is simpler but does not have the same descriptive capabilities as the XML schema. The aim here is to develop a simple interface to start with and therefore a DTD will be used.

[20] World Wide Web Consortium, *XML protocol*[web page], accessed 2001-12-20, available at www.w3c.org/2000/xp/

5 Design of server and interface

5.1 System design overview

The exchange surveillance system is designed as a three-tier client-server system with two alternative data sources, one server and multiple clients, see Figure 12. Basically the main task of the server is to provide real-time or historical data from the exchange in a XML format to client applications. Real-time data is pipelined directly from the CLICK log system and historical data from a database with the information about transactions on the exchange. It is the client applications that does the actual surveillance based on the data from the server. A client connects to the server through an XML interface. For communicating a client application uses a simple request-response protocol. Request – response protocol in the sense that the client posts requests for data from the server either as subscriptions for future data or queries for historical data then the server respond with the appropriate data and finally the connection is closed. Even though the response may take some time there is no notion of a session[21] in the communication.

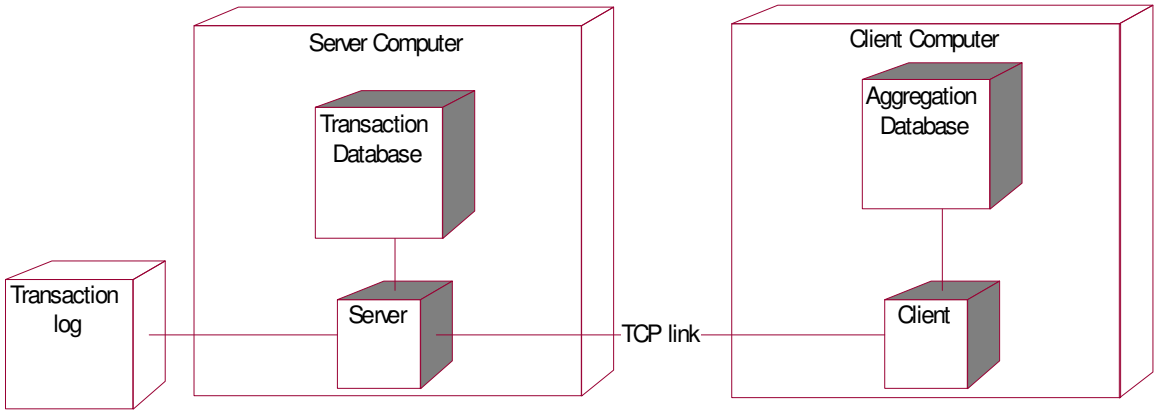


Figure 12, System overview

5.2 Server

The server has four main functions, reading trading data from either a file or a TCP connection, loading selected data into a database, sending trading information in real-time to clients with a subscription, receiving requests and handling queries for historical trading data saved in the database. The functions of the server were identified from the use cases that correspond to the functional requirements of the server stated by OM and they are illustrated in Figure 13 below.

[21] Session, communication where several commands and responses are exchanged between the server and the client on the same connection.

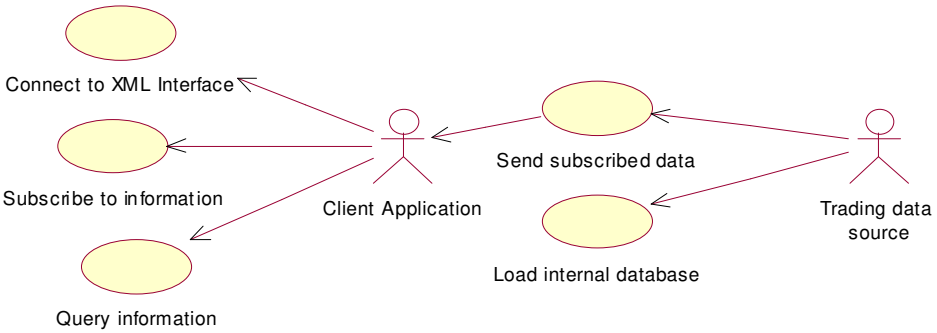


Figure 13, Server use cases

To be able to do these tasks, the server has been designed to be multithreaded, where there are at least three threads running at all times including the main thread. The server functionality of sending real-time data and loading the database is carried out in the main thread. The main thread also spawns two threads when it starts, one thread that reads the trading data from the CLICK system and puts it in a queue; the other thread starts listening for client connections on a TCP port. When a client connects to the server the listener thread starts a new thread that reads from the socket and handles the request. The client can send only one request per connection. If the request is a subscription it is put in a subscription list and if it is a query the thread reading the query handles it. Figure 14 illustrates the thread model of the server.

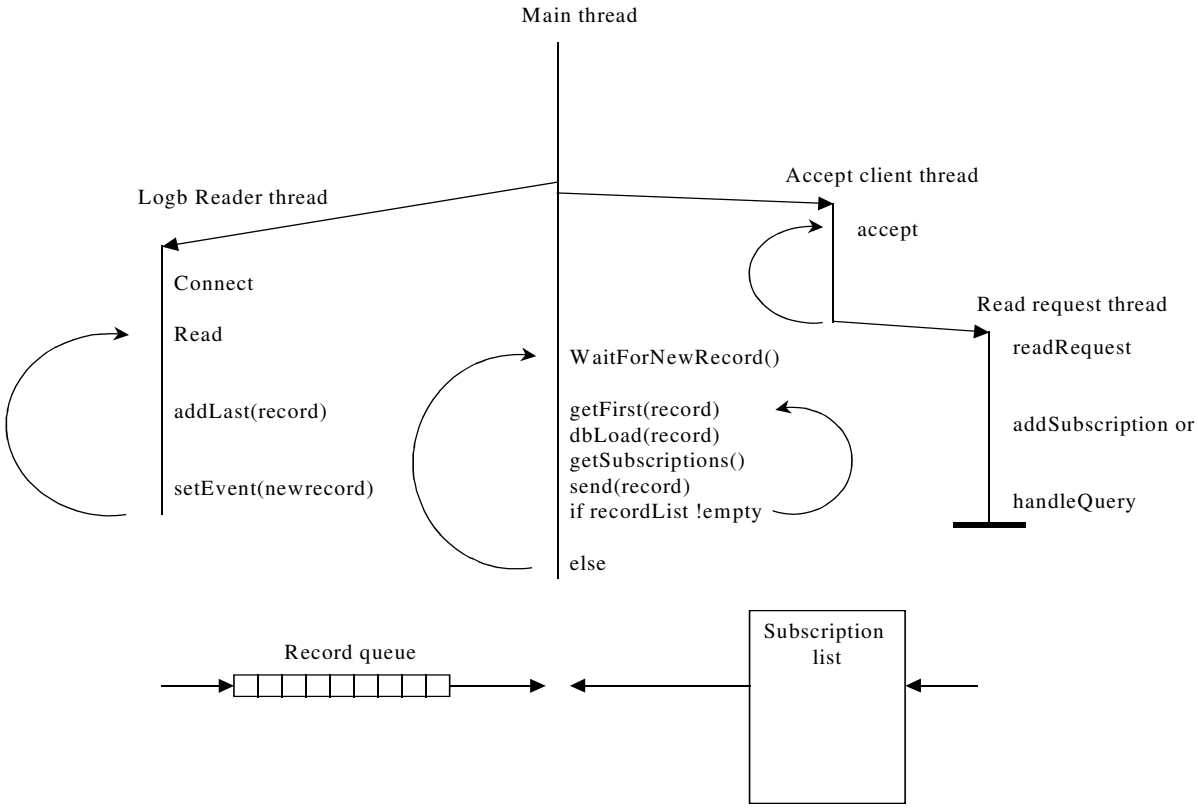


Figure 14, Thread model for server

The main thread of the server basically runs through a loop as long as trading data keeps coming. It waits for the queue to be filled by the reader thread and then removes the first information object in the queue. It takes the information object and loads it into the database and sends it to client who has subscribed for that data. This is done as long as the queue contains information objects. When the queue is empty the main thread waits again for the reader thread to fill the queue.

Based on the use cases the server functionality has been divided into different control objects. These objects are more or less responsible for one task. The system also contains a number of entity objects that map closely to the information messages generated by the market place. More information about the entity object is found in appendix C. The logical design of the server with the control objects is shown in the figure below.

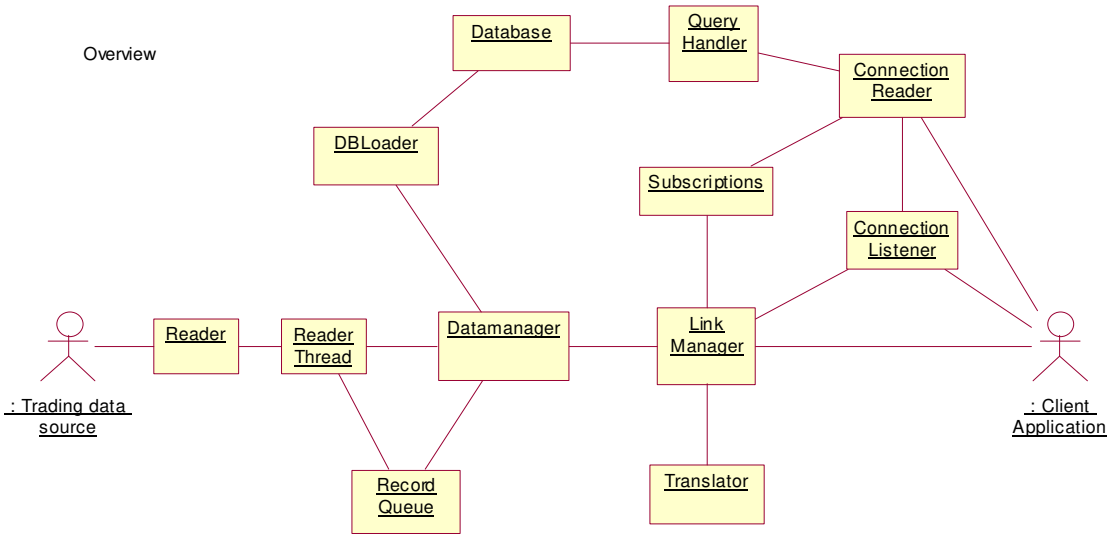


Figure 15, Control class overview of server

The rest of the sections in this chapter are more thorough description of the different parts of the server with sequence diagrams to describe the execution of the different tasks.

5.2.1 Concurrency in the server

As stated above there are at all times 2 threads running plus the main thread. On top of this, there are a variable number of threads depending on the number of clients in the process of connecting to the server. This leads to a number of synchronisation issues in order to ensure that data is persistent and correct when accessed.

The handling of concurrency is done through the use of different types of shared resources. First of all there is the queue of objects between the reader thread and the main thread. Access to the queue is synchronised by locking the queue before accessing it. Queue overflow is avoided by a test for maximum queue size and an event signalling to the reader that the queue is not empty. The main thread is notified of objects in the queue by the queue itself.

The second shared resource is the list of subscriptions. The data access is synchronised by a locking mechanism. It is assumed that there is very little risk of overflow in the subscription because it is not very likely to have for example more than a 100 clients with 100 subscriptions each. Even this would only generates a list of 10000 subscriptions which is not difficult to store in memory. Hence list overflow is not handled, however, to many subscriptions would definitely give performance problems.

The last resource is the database with detailed information about transactions. Here concurrency is handled by built in function in the database.

5.2.2 Reading trading information

This is (as mentioned above) done in a separate thread. The reason for doing this in a separate thread is to be able to handle variations in the intensity with which trading data arrives. At times, like at the opening of a trading day, the information intensity may be much higher than

normally and higher than the server can handle. Then the queue between the reading thread and the main thread serves as a buffer to even out the intensity.

The reading functionality is contained within a reader object that may read from a file or a TCP connection depending on how it is initiated. The reader object also has a set of variables allowing for configuration of which messages to be read from the trading data. The reader thread also shares an instance of a queue with the main thread. The queue is designed to be thread safe by locking the function used. The reader reads one record at a time every time its read record function is called and creates an information object and adds it last in the shared queue. The reader object is a member of the reader thread, see Figure 16.

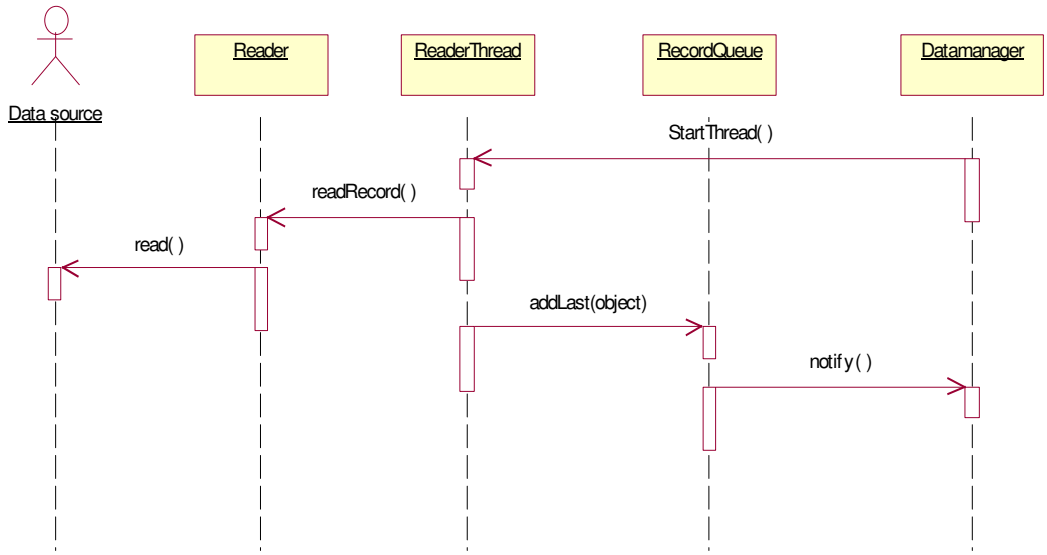


Figure 16, Reader thread sequence diagram

5.2.3 Saving historical data

The main thread saves data to the database by using its data loader object and calling its load data function. The loader function takes the information object to be loaded as a parameter. It checks the type of the information object and loads its into the appropriate database table. In the same way as the log reader object the loader object has a set of variables that configure which information objects are stored in the database. The flow of execution is shown in Figure 17, Send and save data sequence diagram.

The database is designed to maximise the insert speed and therefore it maps closely to the different trading data messages. The database was normalised to third normal form (see the discussion about normalisation in the method section) to remove redundant data. Then it was de-normalised again to gain in query performance. The database contains the following tables:

- OrderEntry
- Quote
- OBChange (Order book change)
- Trade
- ExternalBBO
- InternalBBO
- Hedge
- HedgeExecution
- ImprovementQuote
- InstrumentStatus

See appendix B for an exact description of the physical design.

5.2.4 Sending data to subscribers

The reader notifies the main thread when data arrives from the CLICK system. The data manager then removes the first object in the queue and calls the link manager objects send method with the information object as parameter. The Linkmanager asks the subscription list for the destinations that has subscribed to that type of information and sends it to those destinations. The information is translated to an XML format by the XML translator and is consequently sent in that format. The main thread keeps removing the first object in the queue and sending it in a loop until the queue is empty. Then the data manager enters into a wait state until it is notified again of objects in the queue. See the sequence diagram in Figure 17.

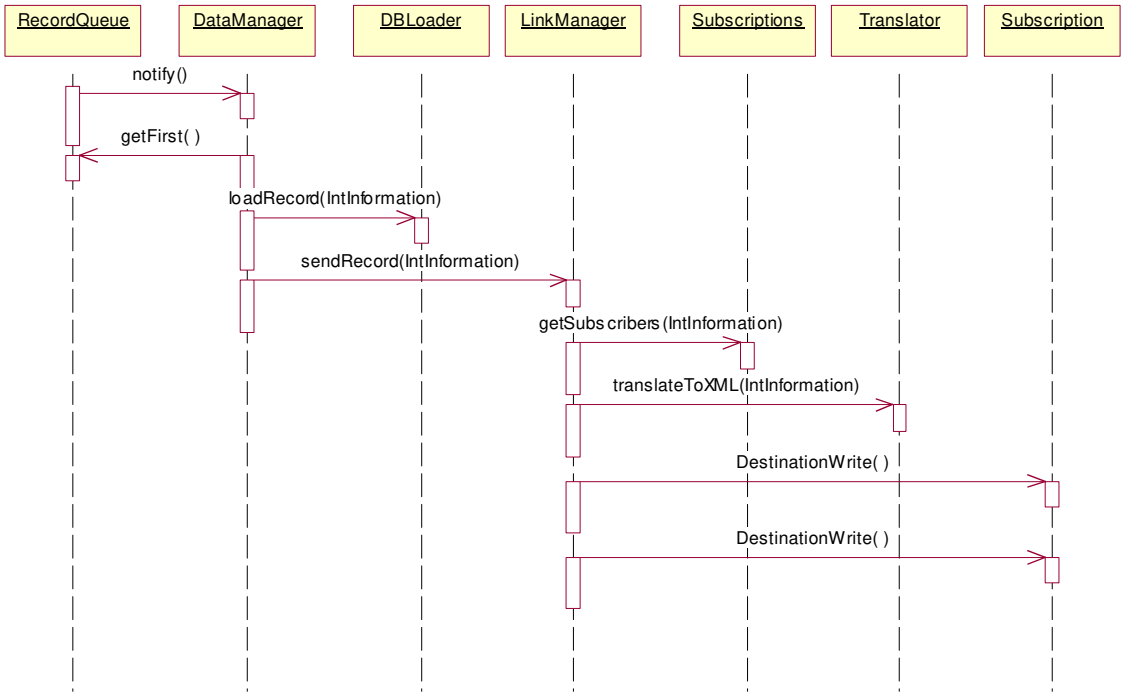


Figure 17, Send and save data sequence diagram

5.2.5 Handling requests for data

When a client connects to the server the thread listening to the server socket spawns a new thread that waits for a request to come in. The request is in an XML format described in the next section. The handler thread listening to the connection reads the request when it arrives and then translates it to an internal request object containing all the parameters defining the request and the socket associated with it. The handler thread then checks the type of the request and if it is a subscription it adds the request to the subscription list and then exits, see Figure 18, Subscription handling sequence diagram.

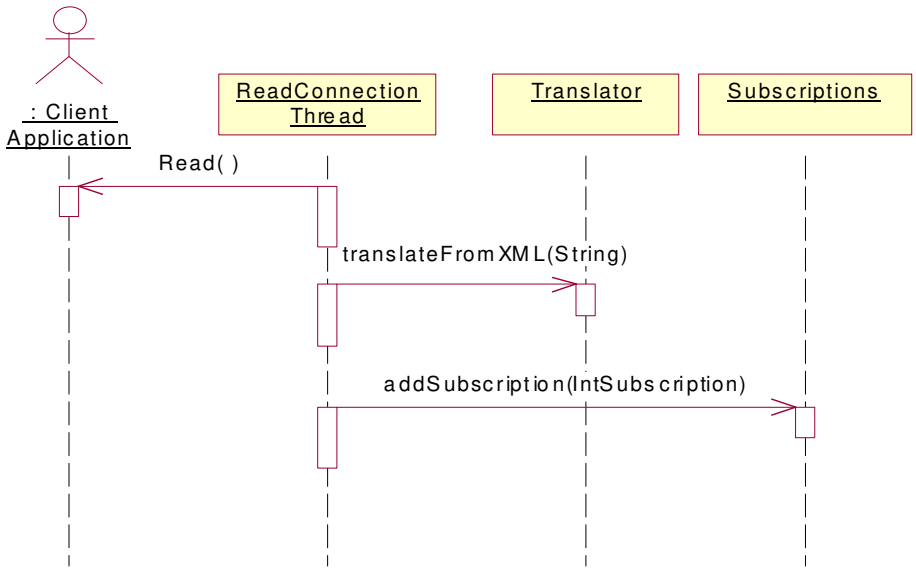


Figure 18, Subscription handling sequence diagram

If the request is a query the thread that reads the request handles it. The request is translated from XML to a request object. The handler thread uses its query handler object to handle the request. The query handler translates the request into an SQL statement and queries the database. The response is translated into information objects and then translated to XML by an XML translator. The result is sent to the destination of the request. Once all the information has been sent the handler thread dies. The flow of execution is shown in the figure below.

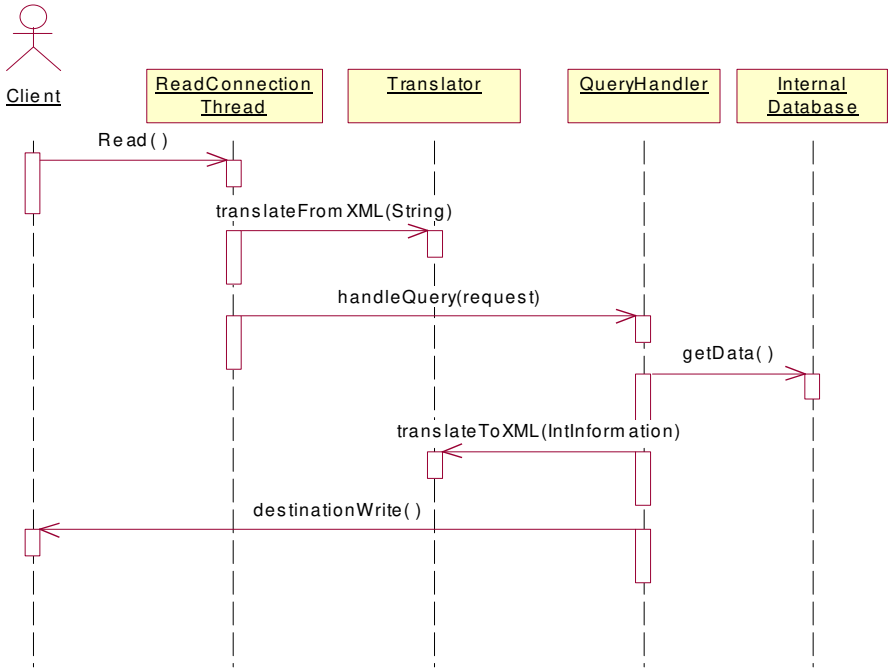


Figure 19, Query handling sequence diagram

5.3 Interface

The interface is governed by a simple request – response type protocol. A client sends in a request and the server responds with appropriate transaction data. The request can be either in the form of a subscription for future trade data or a query for historical data. Both requests and responses in the protocol are sent over the interface in an XML format. The Document Type Definition (DTD) specifying the format is found in appendix A. The request is the only command defined by the protocol and the client can only send one request per connection. After the server has sent all of the data corresponding to the request in response the connection is closed down by the server.

5.3.1 Protocol

As mentioned before in the analysis section the choice was made to develop a simple protocol from scratch rather than trying to use some existing general framework. It was decided that the interface should be able to handle 2 types of requests. One is subscriptions for future data of a type specified by the request. The request is time limited to a specified stop time, the start time is set to the time when the request enters the subscription list. The second is a query for historical data entered in the database. The protocol also requires the client to precede every request with the size of the message in bytes in a textual format. This is a temporary solution to let the server read all off the request from the socket before starting to parse it.

A useful functionality would be to have subscriptions that start in the past and continue into the future. This would allow clients to connect to the server at any time during the day and still monitor the trading from the beginning of the session. With the current design this would however create difficult performance and synchronisation problems. Assuming that the

server normally runs under high load the performance would plunge if late clients were allowed to recover earlier data all at once. Recovery of data could instead be allowed as a batch job after the trading session.

The server responds to the request with an XML document. Both query and subscription responses follow the same XML format. In the case of a query, all of the information is sent once it has been retrieved from the database. The signal to end the communication is the end tag of the document. If the database does not contain any of the data asked for an empty XML document is sent as a response.

If the request is a subscription, the beginning of the XML document is sent to the client when the subscription is added to the subscription list. The beginning of the XML document works as a signal to the client that the subscription is accepted. Then as long as the subscription remains valid, i.e. as long as specified in the request, subscribed information is sent as XML to the client when it is received from the CLICK system. The response forms one valid XML document being sent successively when information arrives. When the subscription expires the server ends the communication by sending the document end tag. The document is not a valid XML document until the end tag has been sent. The figure below illustrates the basic flow of messages between a client and a server.

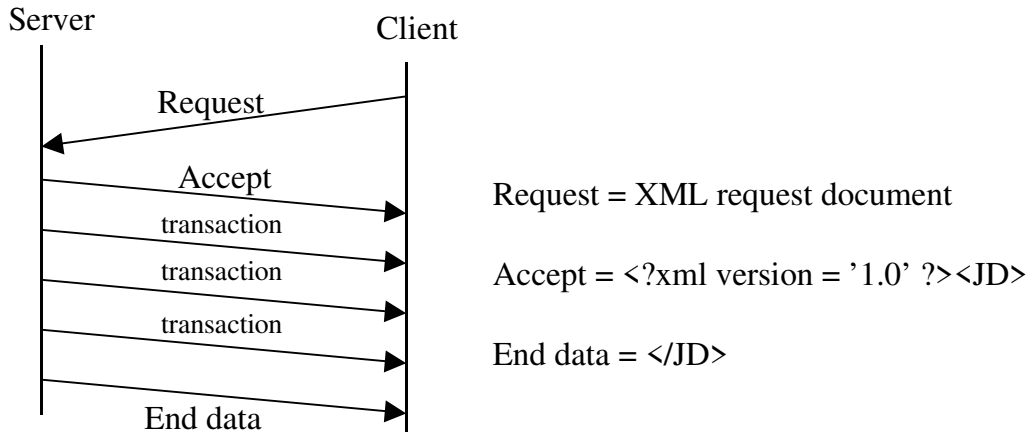


Figure 20, Communication flow client - server

Below is a description of the information in a request and the type of transaction messages the XML interface defines. For a strict DTD definition of the format and an explanation of the meaning of the different tags used see appendix A. The tags used are not in full text since that would increase the amount of information to be passed over the connection too much to be a viable solution at heavy load like 10 Gigabytes of raw information from the trading system.

5.3.2 Request

The requests that the interface can handle are predefined. The format of the request has ten different parameters that define the query, where some of the parameters also have a predefined set of values that are valid.

The parameters in a request are shown in

Table 8.

Table 8, Request parameters

Parameters	Predefined values	Explanation
Request type	Subscription	Future data.
	SubscriptionLight	Future data where some fields in the transaction messages are suppressed.
	Query	Stored data.
	QueryLight	Stored data where some fields in the transaction messages are suppressed.
Request name	All	This value is only valid for subscriptions in order to avoid queries for a too large amount of information.
	Surveillancedata	This is value returns the minimal subset of the available information necessary for the client to perform its surveillance functionality. For our implementation of the client the necessary information is quotes, orders, trades and order book changes.
	Transaction name	The name of a specific transaction message, one of the following: • Trade • Order • ExtBBO • IntBBO • Hedge • HedgeExecution • Quote • PriceImprovementQuote • OBChange
Instrument	All	Returns information for all instruments traded at the exchange. This only a valid value for subscriptions also to avoid large queries
	Instrument name	The name of an instrument should be in a readable form and not some numerical code. A name could for example be Volvo A, SSE. This is not implemented in the prototype due to the time it would take to develop. Instead the internal numerical representations are used.
StartTime	YYYY-MM-DD hh:mm:ss	The field is only considered for queries. Subscriptions are assumed to start when they are received.
StopTime	YYYY-MM-DD hh:mm:ss	The value is used in both queries and subscriptions
Customer	All	
	Empty string	

	Customer name	The name of a customer to the exchange, i.e. a broker firm.
User	All	
	Empty string	
	User name	The name of a user account at a brokerage firm, i.e. a specific broker.
Client	All	
	Empty string	
	Client name	The end client, this information is not available in the trading system today.
Order number	Empty string	
	Number	If this field is used the request is assumed to be a request, and the return is all the information about that order number found in the database.

5.3.3 Trading information

The format of the trading information sent as XML to the client correspond closely to the format of the information from the trading system itself. As discussed earlier, the Jiway CLICK system generates a number of messages that are more or less applicable to many electronic exchanges but also others that are quite specific to the Jiway business model. The more general transaction messages that the XML interface defines are:

Message	**Explanation**
Order	This is a transaction message with information about a new order in the system.
Quote	This message contains information from a market maker about an instrument.
Trade	Information about a trade with references to the two matched orders.
Order book change	Information about any change in the order book and it is generated after trades, order entries and quotes for example.
Internal Best Bid Offer	Information about the current best bid and best offer for a certain asset at the market place.

The transaction messages more specific to the Jiway business model defined in the XML interface are:

Message	**Explanation**
External Best Bid Offer	BBO at the away market of the asset.
Hedge Order	An order that is sent to be traded at the away market because the BBO at the away market is better than at Jiway.

Hedge Execution Report	Report of a successful hedge where the original hedge order has been traded at the away market in part or fully.
Price improvement quote	Record with information that a market maker will trade a certain volume of an asset at the market place to the price specified by the external BBO.

The XML interface should be configurable and flexible. Configuration of the XML format is done through a set of boolean variables which indicate whether or not a specific piece of data should be included in the transaction messages. This allows for reconfiguration of the XML interface at runtime.

6 Implementation of server

This chapter describes the implementation of the server with the different classes and the execution loop in the different parts of the server. The aim is to provide both a static and a dynamic view of the prototype implementation focusing on the actual in code. The implementation language was C++ and the development tool Visual studio from Microsoft. As mentioned before Windows NT 4 was used as platform to run the server.

In total the server is made up of 86 different classes but around 60 of them are entity classes of different levels containing transaction data from the exchange.

6.1 Entity classes

The entity classes are the classes containing the information handled by the system as discussed earlier in the method chapter. All classes containing transaction data are derived from the same base class ***IntInformation***. The class ***IntInformation*** is never instantiated in itself, it only contains data fields common to all sub classes and provides a way to use one pointer type for all sub classes. The final sub classes represent events at the exchange. These entity classes are instantiated during runtime and they are presented in appendix C.

The server uses other entity classes acting as shared resources in the server to allow communication between the threads, shown in Figure 21.

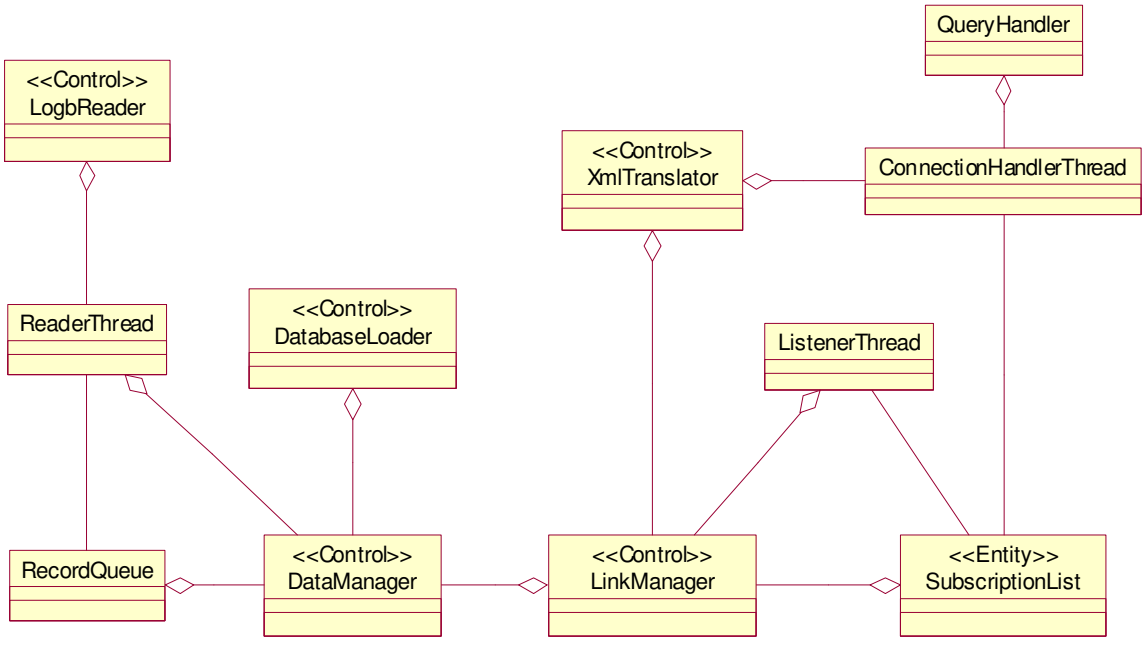


Figure 21, Basic class diagram for the server

The ***RecordQueue*** class is a thread safe queue acting as a buffer between the ***ReaderThread*** and the ***DataManager***. The queue is implemented by a *dequeue* object from the standard template library (STL). It uses a lock object wrapping Microsoft's critical sections for thread

synchronisation. The queue sets an event shared with the main thread when an object is added to the queue. ***SubscriptionList*** contains a vector of pointers to received subscriptions and functions to add and retrieve subscriptions. The list manages the removal of old subscriptions by itself. The list uses the same type of lock objects as the ***RecordQueue*** to synchronise access.

6.2 Control classes

The control classes contain different functionality according to their tasks. They are described briefly here and shown graphically in Figure 21, Basic class diagram for the server:

LogbReader, the main function is the read record function that reads a record from the initiated data source and creates internal information objects. The data source can be a TCP socket or a file, however, the TCP reader has not been tested. The socket class is a wrapper for windows specific socket functions. When reading from a file the STL object *fstream* is used.

ReaderThread, controls the reading functionality of reading records and adding objects to the queue as long as the data source is open. It inherits its basic API from the ThreadObject class, which is the startThread function.

The ***DataManager*** control the main execution loop of saving information and sending data to subscribers.

DatabaseLoader performs the actual insertion in the database by creating the actual SQL insert statements and sending them to the database via a database connection class wrapping the MFC *CDatabase* class used.

The ***LinkManager*** governs the sending of subscribed information by using the SubscriptionList to retrieve the subscriptions and the XMLTranslator to translate the information the an XML format. It also starts listening for connection requests implemented in the connection listener thread.

The ***ConnectionHandlerThread*** handles the requests either via the query handle or by adding a subscription to the ***SubscriptionList***.

Class ***QueryHandler*** uses MFC specific *CRecordset* classes to retrieve information and sending it on a socket specified by a request object.

6.3 Dynamic description

This section will describe the execution flow in the 4 different threads that can be active. A graphic representation is shown below in Figure 22, Thread model for server.

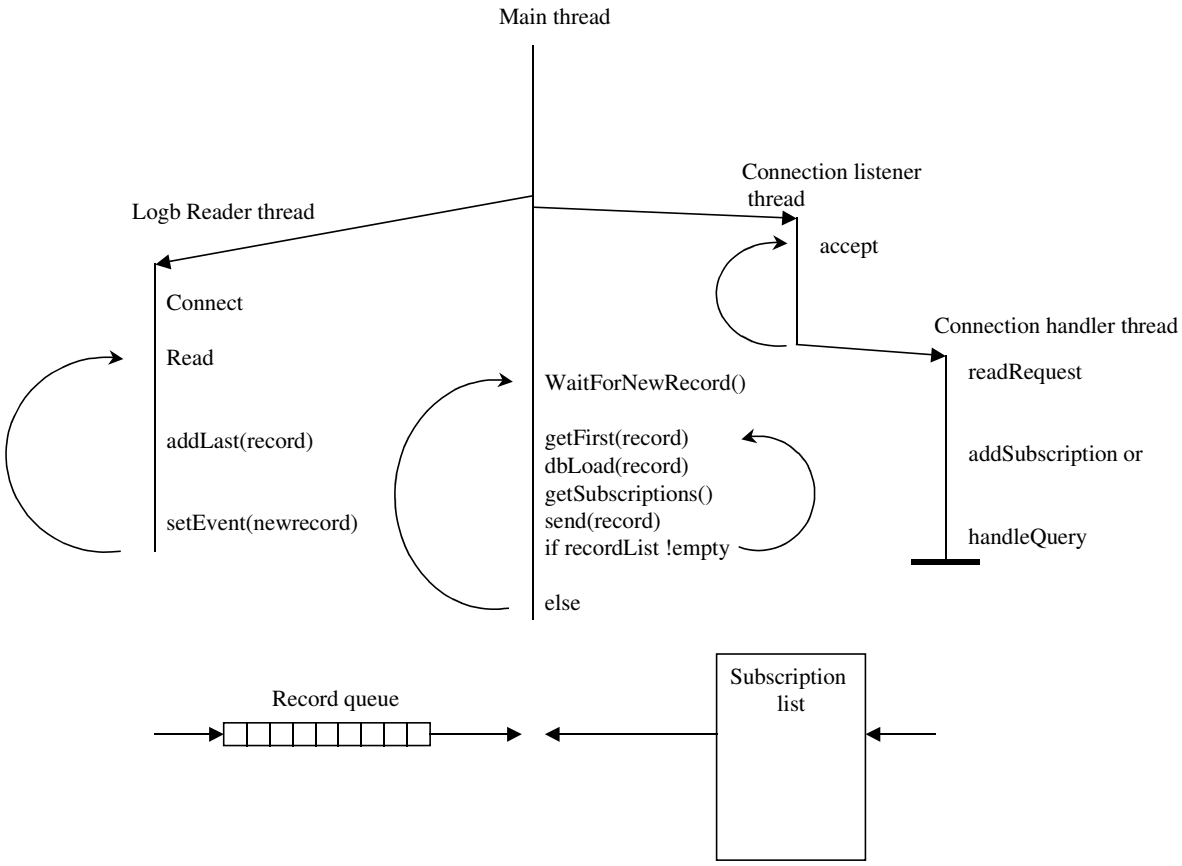


Figure 22, Thread model for server

6.3.1 Thread descriptions

The main thread starts by creating the datamanager object and then calling its go() function. The datamanager then initialises the objects it uses, the logbReader, the recordQueue and the readerThread where the reader thread takes the logbReader and the recordQueue as arguments in its constructor. It also creates the linkmanager object and dataLoader object.

The execution starts when the datamanager calls the linkmanager's listenForConnections function that starts the connection listener thread and the readerThread's startThread function. Then the datamanager goes into a wait state. It waits for events from the recordQueue that the queue contains objects or from the readerThread that the data source is closed.

In the case when there are records in the queue the datamanger enters a loop as long as there are records in the queue. In the loop it removes the first object in the queue. Calls the dataLoader's loadInformation function with the retrieved object as a parameter. The dataLoader checks the type of the object, extracts the information from it in a text format and creates SQL statement that are loaded into the database. Then the datamanager calls the linkmanager's sendInformation function with the same object as parameter. The linkmanager checks the type and calls the subscriptionList's getSubscriptions function. The subscription list matches the information in the object to the parameters of the subscriptions in the list and

creates a vector with matching subscriptions. If the vector returned contain any subscriptions the XMLTranslator's translate function is called and the resulting XML string is written to the socket in the subscription object. In case the data source has been closed the datamanger goes though the same type of loop as above to empty the queue and then exits.

Table 9, Log reader thread description

Thread	**Log reader**	
Goal in Context	Read log reacords from a TCP link or log file and transform them into internal transaction objects.	
Preconditions	Log data source initialised in LogbReader	
Trigger	Read log data request comes in	
DESCRIPTION	Step	Action
	1	Check if the log data source is open
	2	Call LogbReader's readRecord function
	3	Check queue size
	4	Enqueue object returned by readRecord
	5	Queue notifies main thread of object in queue
	6	Return to step 1
EXTENSIONS	Step	Branching Action
	1a	Log data source not initialised: 1a1. End thread
	3a	Queue full: 3a1. Wait for queue to empty then resume step 4.

Table 10, Connection thread description

Thread	**ConnectionListener**	
Goal in Context	Wait for and accept incoming TCP connections	
Preconditions	The server socket is initialised	
Trigger	Call to LinkManagers listenFor Connections function	
DESCRIPTION	Step	Action
	1	Blocking socket accept call
	2	Initialise ConnectionHandlerThread with socket and subscriptionlist
	3	Start ConnectionHandlerThread
	4	Return to step 1

Table 11, Connection handler thread description

Thread	**Connection handler**	
Goal in Context	Read request from a TCP connection and handle the subscription or query	
Preconditions	Socket is connected and the subscription list is initialised	
Trigger	A connection has been accepted	
DESCRIPTION	Step	Action
	1	Blocking read from socket until first "<" character
	2	Parse data to message size
	3	Blocking read of message size
	4	Translate data to request object by XMLTranslators translateFromXML function.
	5	Check request type
	6	Handle request
	7	End thread
EXTENSIONS	Step	Branching Action
	6a	Request is subscription: 6a1. Add request to subscriptionlist 6a1. Resume step 7
	6b	Request is query: 6b1. Call QueryHandler 6b2. QueryHandler parses request to SQL 6b3. QueryHandler executes SQL query 6b4. Translate returned recordsets to objects 6b5. Send data to client 6b6. Resume step 7
Comment	When the client request is received, the handler thread reads until the first "<" character and parses the first characters to a long integer that defines the size of the request. This is the temporary solution to allow the server to read all of the message before parsing the XML content.	

7 Evaluation of server

The evaluation focuses on testing the speed of the server by using statistics from different test runs. It is not concerned with the users perceived performance. The evaluation is divided into a number of categories.

- Server read performance
- Database insert performance
- String generation performance
- Overall system performance

Performance here is defined as the number of transaction messages per second from the exchange the server can handle.

7.1 Test method

The server has been tested with several different configurations to evaluate the design and implementation. Each test results in a runtime for the server handling a certain amount of data. The server runtime was timed with the different configurations starting from a very clean configuration where the server does nothing else but read from the logb file. This was done in order to provide a lower threshold for the runtime on this platform. The platform used limits the speed of execution in the server and a clean configuration gives the highest possible speed. Then different configurations of functionalities have been added on to see the effect they have on the runtime. Changes in configuration were made by commenting parts of the code and the recompiling it. During the tests, the data source were log files on the local hard drive.
Not all parameters influencing the results have been taken into consideration but these tests serve more as guidelines for how performance might be enhanced rather than a strict performance test.
The general parameter setting throughout during the test are depicted in Table 12.

Table 12, General parameter settings

Parameter	Setting
Platform	• CPU – Intel Pentium III , 730 MHz • Memory – 128 Mb • Operating system – Windows NT 4.0 workstation
Data source	Log files on local hard drive
Records in log file	1893936
Size of log file	100117 Kb
Buffer size in server	6000 objects
DBMS	MySQL

7.2 Results

A general observation from the test is that the average CPU usage stayed very close to 100% during most configurations. Only during the configuration where the server does nothing but read the log data the average CPU usage was around 95%. This is an indication that the

implementation may not be very CPU efficient. One would assume that the speed of the server ought to be restricted by disk I/O operations but here it seems to be processing speed.

7.2.1 Server read performance

The first test was to run the server with all functionality removed except for the reading of the log file. All log records were read as they would during a normal run but no information objects were created and no other tasks were performed. The main thread waited for objects to be added to the queue the entire run. This gave (as shown in the table Table 13) a runtime of one and a half-minute. This provides a limit for the performance of the server.

Table 13, Read only performance

Create objects	Messages read	Log file size (Kb)	runtime
No	1893936	100117	00:01:31

Then the server was run in a configuration where internal information objects were created, added and removed from the queue and finally deleted. This was done in two configurations one were the database loader and linkmanager were never called. In the other configuration the database loader was called and the database insert statements were generated but not executed. The LinkManager was also called but without any subscriptions present, therefore in this implementation no XML strings were created.

Table 14, Performance with objects created

DB loader	DB insert	Request	Linkmanager	Objects created	runtime
no	No	no	No	1872443	00:06:24
Normal	No	no	yes	1872443	00:07:16

Table 14 shows that the creation of the internal information objects is costly in processing time and runtime. The runtime increases more than four times.

7.2.2 Database insertion performance

The database insertion performance was tested in a number of different configurations. First, a number of different versions of indexes were tested. Multi-valued inserts into the database were used in these tests where one insert statement contained up to 200 rows of data. BBO messages where not inserted which decreased the load on the database significantly. A first observation is that database inserts increase the runtime to with up to 50% compared to the results above.

The choice of index did make a difference in performance, the smallest index was the fastest but this version is not very useful for when doing select statements. It only indexes the sequence number of the messages but it will always be indexed since it is the primary key for many messages. Version 5 also indexes the asset specified in the record, which may be a very useful for queries.

The shortest runtime reached gives an insertion speed of 1296 rows per second. If the quantity of data is increased 30 or 100 times, this may effect the performance negatively if the processing needed to create indexes is not linear to the number of rows of data.

Table 15, Database insert performance

DB (rec/stat)	DB rows	Data size (Mb)	Index size (Mb)	Index version	Runtime (min:sec)
200	740 283	30	16,7	Index v1	10:03
200	740 283	30	13,1	Index v2	12:37
200	740 283	30	8,51	Index v3	09:31
200	740 283	30	19,6	Index v4	10:36
200	740 283	30	10,9	Index v5	09:55
100	740 283	30	10,9	Index v5	09:46
50	740 283	30	10,9	Index v5	10:04
300	740 283	30	10,9	Index v5	09:45
400	740 283	30	10,9	Index v5	10:17
100	1 872 443	67	27,6	Index v5	16:57

According to the MySQL manual, significant performance gains can be achieved by using multi-valued inserts[22], i.e. inserting more than one row per SQL statement. Therefore the maximum number of rows in one statement was varied to investigate the effect on the runtime. The results in the table show that the best result was reached with an insert size of up to 300 rows per statement. It is somewhat surprising that larger insert sizes diminish the performance. A possible reason for this may be that the concatenation of the long strings needed for the insert statements is not efficient. The implementation uses the STL[23] string class for character data and string concatenation, which dynamically allocates memory and this may be as fast as using for example pre-allocated memory of a fixed size.

Finally a test was run where all objects where inserted into the database. No clients were connected and the code was compiled without the LinkManager. Then an insertion performance of about 1800 messages / second was reached. An interesting note is that the CPU load during this run was on average a bit below 90% and there seemed to be a lot of disk swapping. Maybe performance could be improved by changing memory parameters for the MySQL database.

[22] MySQL homepage, MySQL handbook, Chapter 5 MySQL optimization, 2001-12-12, http://www.mysql.com/documentation/mysql/bychapter/manual_MySQL_Optimisation.html#MySQL_Optimisation

[23] STL, Standard Template Library for C++.

7.2.3 String generation performance

An effort to test the performance effect of the concatenation of strings was made. The server was run with a client, so that the XML strings were created. XML data was never written to the socket though, in order to make sure that the client did not hamper the server's performance. Strings with insert statements for the database were also generated up to a size of 200 values per statement. One test run the data was never inserted into the database, in the second data was inserted. Compared to when the server is run without database loader and XML generation the extra processing needed increases the runtime with about 60%, and when the insert statements are executed with almost 90 %.

Table 16, String concatenation performance

DB (rec/stat)	Client	Generate XML	XML size (Kb)	Objects created	runtime
no	Yes	Yes	342590	1872443	00:10:11
200	Yes	yes	342590	1872443	00:12:02

With the configuration where the server generates all string data but never inserts or sends it, i.e. without dependencies of the TCP link or the database gives a server performance using the data in the table above of:

$$\frac{\text{No of messages}}{\text{Time in second}} = \frac{1872443}{(10*60+11)} \approx 3100 \text{ messages/ second}$$

A runtime of 12 minutes and 2 seconds gives an average processing speed of:

$$\frac{\text{No of messages}}{\text{Time in second}} = \frac{1872443}{(12*60+2)} \approx 2600 \text{ messages/ second}$$

Here the number of objects created is used as a message count since this better reflects the performance than using the number of records in the log file.

Assuming that a client is not allowed to receive data in a slower pace that the server sends it, the only extra time needed in real operation would be to write the data to the socket. However, in this implementation the server waits for the client, so that by being slow, the client slows down the whole system.

7.2.4 Interface evaluation

The test runs above also show that size of the XML data generated by the server was 342 MB from the 100 Mb of raw log data. The result is a size increase ratio of:

$$\frac{343}{100} \approx 3{,}4$$

This is in line with the earlier estimation of a size increase ratio of 2,5 to 4,75 with respect to the size of the original data. With a fast connection between the server and the client, the amounts of data does not seem to pose a problem with one client, since it only requires a speed

below 500 Kb/sec over the connection. With 20 different clients though, the speed of the connection becomes more crucial.

7.2.5 Overall system performance

In order to test the overall performance of the server, it was run with a client application that read the XML data sent over the socket connection. To test the server performance rather than the client performance the client was configured to be as fast as possible. The test client here did no calculation on the data received. It only read the XML data and transformed it to information objects.

Table 17, Overall performance

DB insert	Client Request	Objects created	Write XML	XML size	DB rows	Runtime
100	All	1872443	Socket	342 590	740 283	00:13:41
100	All	1872443	Socket	342 590	1 872 443	00:19:28

The first run gives a processing speed of:

$$\frac{\text{No of messages}}{\text{Time in second}} = \frac{1872443}{(13*60+41)} \approx 2300 \text{ messages/ second}$$

During this test run the server was configured with all functionality including the ability to query the database for historical information. The effect of this functionality was not tested properly. The client never posed queries to the server, which most likely would have increased the runtime of the server. It is somewhat difficult to test the effect of queries in a meaningful fashion since it depends heavily on how many queries are sent and how much data that has to be retrieved from the database. In the first run only a subset of the data was inserted into the database, BBO records were not inserted and they constitute almost two thirds of the records in the test data.

In the second run, all objects created were inserted into the database. This increased the runtime, but it gives a more accurate picture of the speed of the server. This is due to the fact that one can not assume that two thirds of the messages does not need to be inserted into the database for other types of CLICK target systems. The processing speed of the server then turns out to be:

$$\frac{\text{No of messages}}{\text{Time in second}} = \frac{1872443}{(19*60+28)} \approx 1600 \text{ messages/ second}$$

Queries were tested informally, the client sent a query for a 20-minute interval concerning one asset. The information asked for was quotes, orders, trades and order book changes which was assumed to be the most interesting subset of the information available for surveillance. Query responses were perceived to be more or less instantaneous and did not seem to slow down the server noticeably. However, the test was done using the test data of 100 MB, the effect of a data quantity increase of 30 to 100 times is difficult to predict.

One aspect of the server where performance needs to be improved is the database insertion speed. An organisation called The Transaction Processing Performance council has published

a list of results for a database benchmark that they have defined. This benchmark use different types of transactions and the comparison between those results and the test made here may not be all together valid. Their definition of a transaction may be very different the number of rows inserted per second. However, the results give a hint about the performance possible. They measure the number of transactions per minutes and the top result is 709 220 transactions/minute which means around 11800 transactions/second.[24] This is an improvement of almost a factor 7 of the current insert performance. The database itself could probably be improved also if one takes into account the characteristics of the data. Since we now that data arrive sequentially with respect to time and that queries are concerning a time period, it should be possible to optimise the database configuration and the creation of indexes for this.

Another area that could probably be improved is the generation of character data (as mentioned before), which seems to take up a lot of processing time. In this implementation the STL string class is used. The use of pre-allocated character arrays might be faster than the dynamic allocation in STL strings.

A third area of improvement is how the internal information objects are created. The design used is very object oriented using an object for each field of data in the transaction objects. Because of this a large number of C++ *new* commands are issued for each transaction object created. Another solution would be to use the C structures read from the log data internally. For each record only a new C structure would be created and used internally in the server. To convert the data in the C structures to other formats the server would use template functions instead of object member functions.

Another way to improve performance would be to filter away more record types in the log data. The creation and destruction of objects take a considerable amount of time and decreasing the number of internal object would definitely have a positive effect on performance.

7.2.6 Internal buffering evaluation

The server needs buffering to handle variations in the arrival intensity. This section attempts to evaluate the buffering requirements for a server with the above described performance for a given amount of data generated during the trading session.

As stated in the analysis chapter arrival and processing intensities are assumed to be exponential for time periods T as in M/M/1 systems. Here λ is the average rate of arrivals to the server and μ is the average processing rate of the server. The following is assumed:

- Arrivals follow Exp(λ)
- Processing intensity follow Exp(μ)

Then the traffic intensity ρ is defined by:

$$\rho = \frac{\lambda}{\mu}$$

[24] Transaction Processing Performance Council, Top Ten TPC-C by Performance, Version 5 Results, 2001-12-13, http://www.tpc.org/tpcc/results/tpcc_perf_results.asp

and ρ < 1 for stationary systems. Then the average number of messages (lq) queuing in the system is defined by:

$lq = E[Xq] = \frac{\rho^2}{1-\rho}$ where X is the number of messages queuing in the system at time t.[25]

The testing above gives that the server has a processing intensity of about μ = 2300 msg./sec with one client with the configuration of the first test of the overall system performance. Assuming exponential arrival during the last 2 h period and 3 Gigabytes of data gives a λ = 2500 msg./ sec. A 7200 second period with an average of 200 more messages arriving than being processed gives a buffer of:

$buffer = 7200 * 200 = 1440000$ transaction message objects

almost 1,5 million messages. This is too large a buffer for most platforms. The conclusion is that the server implementation on the current platform would not be able to handle a load of 3 Gigabytes per day.

Assuming a load of 2.5 Gigabytes instead gives a ratio of 25 and the message rates shown below in the table:

	Number of messages	Time (s)	Rate (msg./sec)
Average 7-9	11875700	7200	1649,403
Average 9-13:30	18250425	16200	1126,569
Average 13:30-15:30	14970100	7200	2079,181
Average 7-15:30	45096225	30600	1473,733

Here the average queue size to during the afternoon period would be:

$$\lambda \approx 2100, \mu \approx 2300 \Rightarrow \rho \approx \frac{2100}{2300}$$

$$\rho^2 \approx \left(\frac{2100}{2300}\right)^2$$

$$lq = E[Xq] = \frac{\rho^2}{1-\rho} \approx 10$$

This is the average queue size but there are peak rates much higher during shorter periods. The peak rate measured with the test data scaled up 25 times is 140*25 = 3500 msg./sec

To handle this one-minute peak the buffer would have to be at least:

(3500 – 2300) * 60 = 72 000 messages

[25] Jan Grandell, *Köteori 2:a utgåvan*, 1997 , Institutionen för Matematik, KTH

A buffer of 72 000 messages would be possible with more memory on the existing platform. An estimate of the average waiting period during the afternoon high level of trading is given by:

$$\omega = E[S] = \frac{1}{(1-\rho)\mu} \approx 0{,}005\,\text{sec}$$

This gives that the average delay would be very low i.e. the system is very close to real-time performance with this load. At peak rates the delay could be a lot higher, the time for the server to process a full queue would be around:

$$\frac{QueueSize}{Av.\,\text{Pr}\,oces\,\text{sin}\,gRate} = \frac{72000}{2300} \approx 31\,\text{sec}\,onds$$

The delay would never be longer than this, but if the arrival rate is to high the server will slow down and possibly loose information.

This analysis gives a hint about the buffering in the server and the load possible to handle but is by no means complete nor exact but this was not the purpose either.

8 Conclusions

8.1 Summary

The results of this project are two different XML interface versions where the second version is based on experience gained from the pilot implementation of the first. The second version of the interface is described shortly below and more thoroughly in appendix B.

The implementation of the first interface resulted in a server type application that allow client applications to send requests for data and the server responds with either real-time or historical data. The three different interface types stated in the task definition are incorporated in the pilot server. An evaluation of the implementation was made with respect to message processing speed.

8.2 Performance

As an interesting note one can compare the performance of this pilot implementation to the SMARTS surveillance system product. At the SMARTS home page claim that the system has the following performance:

> "The minimum memory requirements depends on the transaction volume of the market(s) being studied. For a medium sized market (1,000 securities and around 50,000 trades per day), 256 Megabytes for the server would be appropriate, however, an additional 128 Megabytes should be added for each additional user under the central server model.
>
> Recent tests showed that *SMARTS* was able to process alerts for over 1,200,000 order-level transactions including 500,000 trades in 17 minutes on a 1995 HP server with 512 Megabytes of memory and 7 minutes on a 1996 DEC Alpha server also with 512 Mb memory. This equates to 2,860 transactions per second. Depending upon the number of transactions which arise for a given trade this translates to between 500 and 1200 trades per second. This was with a full *SMARTS* surveillance environment running. Both computers were single-processor machines." [26]

This performance quite close to the performance of around 2300 transactions per second reached by the pilot server implementation in this work. As discussed in the thesis by Johan Örtenblad the client application should be able to handle about the same amount of messages per second as the server. With the current implementation of the client there is a problem with the client database which limit the speed of the client. For a full fledged surveillance system there would of course be several other variables used to evaluate the system but these are outside the scope of this thesis.

The performance of the server does not seem to be enough to handle a mature market were a CLICK exchange system could generate 3 Gigabytes of data and more. Especially if one considers the extra processing needed to perform tasks not included in the pilot implementation. Such tasks would for example be the translation of codes internal to the CLICK system to meaningful character data. There are however several roads open for

[26] SMARTS homepage, *Features*, accessed 2001-09-17, available at http://www.smarts.com.au/12ai_Features.html#reqper

performance improvement ranging from just buying more powerful hardware to detailed code optimisation.

8.3 Interface

During the work with the monitoring system it was realised that the design of the interface was not as general as we had hoped. To much target system specific data was included in the interface. This was due to the fact that the functionality of the surveillance client and its information requirements were not fully known in the beginning of the design work. The interface could not be appropriately investigated until the was a prototype client developed that used XML formatted information. Therefore a prototype interface was developed and used. The main drawbacks of the first interface design are:

- Uses to much Jiway and CLICK specific entities
- Difficult to extend

To remedy this it was decided to redesign the interface with more focus on surveillance needs and general trading entities. Exchange surveillance is concerned with the actions taken by its participants and not internal exchange information. The surveillance client needs to know what was done at the exchange and who did it. These ideas lead to a design were information sent over the interface is defined as either an *order entry* or an *order change*. Both these entities have fields that are common and fields that are specific to the type order or type of change. The table below gives an overview of the fields in the interface. It is the type and the reason attribute that contain the data specific to different types of transactions.

Table 18, Interface fields version 2.0

Order entry	Order change
Order identity	Event time
Instrument	Bid order identity
Event time	Ask order identity
Bid or Ask	Reason
Broker firm	
Dealer	
Type	

This interface design was not implemented in the pilot server but a description of how this could be done and a detailed description of the interface is found in appendix D.

8.4 Problems encountered

The overshadowing problem during this thesis has been the performance of the system. The solution implemented can not handle the rate of information in mature markets on the platform used. A number of improvement ideas have been presented but none have been tested in an implementation. There have been other minor problems during the work but more in the area of understanding the information from the exchange system and different technologies used.

8.5 Future work

My opinion is that the first thing to do is to one more time investigate the design of the interface in order to find a good generalisation of the transaction data generated at exchanges and find a good way to make the interface extendable. The second version of the XML interface is an attempt to do this but it has not been tested in an implementation.

The overall performance needs to be improved if the system is to be run on more mature markets than Jiway.

On a less abstract level a mapping between internal codes used in the CLICK system to readable text in the XML interface needs to be developed. A way to reconfigure the XML interface without changing the source code would also be desirable as well as ways to configure the server from a user interface.

9 References

9.1.1 Printed material

1. Andreas Oderstad, *Call Traffic Monitoring System* [masters thesis], March 2000, IT department, KTH
2. Astrid von Euler-Chelpin, *A study of an XML Protocol for Manufacturing Systems using XML and Java* [masters thesis], June 2001, IT department, KTH.
3. Connolly and Begg, 1998, *Database systems*, Addison-Wesley
4. Jan Grandell, *Köteori 2:a utgåvan*, 1997 , Institutionen för Matematik, KTH
5. Steven Holzner, *XML Complete*, 1998, McGraw - Hill

9.1.2 Online material

6. Andrew Conrad, July 26 2001, *A Survey of Microsoft SQL Server 2000 XML Features,* Microsoft Corporation available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml07162001.asp
7. M J Aitken and J H Berry, July 1991, *Market surveillance at the Australian Stock Exchange*[article online], accessed 2001-09-18, available at http://www.smarts.com.au/surveill.html
8. OM groups homepage, accessed 2001-09-12, available at http://www.omgroup.com
9. SMARTS homepage, accessed 2001-09-17, available at http://www.smarts.com.au
10. Transaction Processing Performance Council, Top Ten TPC-C by Performance, Version 5 Results, 2001-12-13, http://www.tpc.org/tpcc/results/tpcc_perf_results.asp
11. *The UML tutorial part 1*, accessed 2001-09-16, available at http://www.sparxsystems.com.au/UML_Tutorial.htm
12. World Wide Web Consortium (W3C), *XML in ten points,* web page accessed 2001-12-17, available at http://www.w3.org/XML/1999/XML-in-10-points.html
13. World Wide Web Consortium (W3C), *The XML 1.0 Specification*, web page accessed 2001-12-17, available at http://www.w3.org/TR/xmlschema-0/
14. World Wide Web Consortium, *XML protocol*[web page], accessed 2001-12-20, available at www.w3c.org/2000/xp/

9.1.3 Meetings / interviews

15. PerÅke Sundström, Mats Danielsson and Håkan Carlbom, Interview 20010917 at OM
16. Mats Danielsson, Interview 20011012 at OM.
17. Johan Norén, Interview 20010910 at OM.