API Analysis of Financial Exchange Systems

Jill Rhoads

TRITA-NA-E03126
API Analysis of Financial Exchange Systems

Jill Rhoads

TRITA-NA-E03126

Master’s Thesis in Computer Science (20 credits) at the School of Computer Science and Engineering, Royal Institute of Technology year 2003
Supervisor at Nada was Joel Brynildsson
Examiner was Stefan Arnborg
Abstract

The purpose of this Master's Thesis is to analyze the electronic exchange phenomenon and specifically discusses the two software platforms SAXESS and CLICK. Both trading platforms are described from a client application developer's point of view. Historical background, as well as a basic overview of options trading is introduced. Financial interface standards are broadly covered as well as XML's introduction into financial application development. Suggestions for improving the CLICK OMnet API in order to facilitate better developer understanding for future development are discussed.
Analys av gränsnitt för applikationsutveckling i börssystem

Sammanfattning

Foreword

This report is the documentation of a Master’s Thesis in Computer Science submitted to the Royal Institute of Technology’s Numerical Analysis and Computer Science (NADA) college in Stockholm, Sweden. The thesis partially fulfills the requirements for a Master’s of Science in Computer Science and yields 30 credits under the European Credit Transfer System (20 Swedish credits) and represents 20 weeks of full time research. For more information about the master’s project in general see http://www.nada.kth.se/utbildning/grukth/exjobb/index.html.en.

This thesis was commissioned by OM Technology AB from February through July, 2003.

CLICK and SAXESS are registered trademarks owned by OM Technology AB.

Acknowledgments

I would like to thank a few people for their input regarding this Master’s Thesis. First, I would like to thank Göran Altius, M.Sc. in Computer Engineering, from OM Technology AB for his guidance while working on this report. Thank also to Joel Brynielsson, M.Sc. in Computer Engineering and Ph.D. candidate, for his help as my Supervisor at NADA. My endless gratitude also goes to my husband, Björn Rhoads, for his help with LaTeX and his unending patience throughout this process.
Thesis Outline

Chapter 1 will look at the background for this Master's Thesis as well as discuss the chosen methodology. An introduction is also provided on the background of financial exchange markets and how technology has been an inherent part of their growth. An overview of how options trading works in practice and theory will also be discussed.

Chapter 2 will introduce basic concepts like client/server programming as well as other topics of interest like event handling and middleware.

Chapter 3 will discuss the CLICK exchange system and its components in general.

Chapter 4 will give a quick overview of OM’s other financial exchange platform SAXESS in order to give another example of exchange market mechanisms.

Chapter 5 describes the CLICK external API in more detail in order to give the reader insight for later discussion.

Chapter 6 will look at financial information standards.

Chapter 7 will discuss strengths, possible improvements, and limitations regarding the CLICK external API.

Finally, chapter 8 is an executive summary where the CLICK API improvement conclusions will be summarized.
Contents

1 Introduction
1.1 Purpose ............................................... 1
1.2 Methodology ........................................... 2
1.3 Brief Historical Background Information ................. 2
  1.3.1 A Brief History of Financial Markets ............. 3
  1.3.2 A Brief History of the Options Market ........... 6
1.4 History of OM ........................................... 6
  1.4.1 OM Systems ........................................ 7
1.5 Options .................................................. 8
  1.5.1 From Futures to Options ......................... 8
  1.5.2 The Gory Details .................................. 8
  1.5.3 Electronic Trading in Options ................... 9

2 Client/server programming .................................. 13
  2.1 N-Tier Architecture .................................. 13
  2.2 Event Handling ....................................... 15
    2.2.1 Typed Events .................................... 15
    2.2.2 Push and Pull Models ............................ 15
    2.2.3 Event Channels .................................. 16
    2.2.4 Event Filters .................................... 16
    2.2.5 Non Functional Constraints .................... 16
    2.2.6 Formal Event Models ............................ 16
  2.3 Synchronous Versus Asynchronous Programming ......... 17
  2.4 Endian Programming .................................. 18
  2.4.1 In Combination with ANSI C ..................... 19

3 Overview of the CLICK Trading Platform ................. 21
  3.1 Marketplace Software ................................ 22
    3.1.1 Marketplace (MP) ................................ 22
    3.1.2 Information Dissemination (IN) ................ 23
  3.2 Base Software ........................................ 24
  3.3 Clearing Software .................................... 24
  3.4 Architecture .......................................... 24
    3.4.1 RTR ............................................. 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>OMnet</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Quick Overview of SAXESS</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>History</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>System Architecture</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Exchange Transaction Protocol (XTP)</td>
<td>28</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Transaction Flows</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Exchange Message Protocol (XMP)</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Exchange Message API (XMAPI)</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>SAXESS Trade Client/Server</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>Development</td>
<td>31</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Change Management</td>
<td>31</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Documentation</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>The Click OMnet API</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Interface Design</td>
<td>35</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Transaction Headers</td>
<td>36</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Reliable Transaction Router</td>
<td>36</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Tunneling and Compression</td>
<td>36</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Structs</td>
<td>37</td>
</tr>
<tr>
<td>5.1.5</td>
<td>C Data Types</td>
<td>37</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Batch Queries</td>
<td>37</td>
</tr>
<tr>
<td>5.1.7</td>
<td>Broadcasts</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Development</td>
<td>39</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Documentation</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2</td>
<td>OM Courses</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>Interface Standards</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Straight-Through Processing (STP)</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>Messaging Protocols</td>
<td>41</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Non-XML Protocols</td>
<td>41</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Extensible Markup Language (XML) Protocols</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Discussion on XML Protocols</td>
<td>44</td>
</tr>
<tr>
<td>6.4</td>
<td>Discussion on Interface Standards</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Click API Discussion</td>
<td>47</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>47</td>
</tr>
<tr>
<td>7.2</td>
<td>General Programming</td>
<td>48</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Binary Data Structures</td>
<td>48</td>
</tr>
<tr>
<td>7.2.2</td>
<td>C++ Wrappers and Object-Oriented Design</td>
<td>48</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Limit Thread Synchronicity</td>
<td>50</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Consequent Formats within API Structs</td>
<td>51</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Broadcasts</td>
<td>51</td>
</tr>
<tr>
<td>7.3</td>
<td>Developer Tools</td>
<td>53</td>
</tr>
<tr>
<td>7.3.1</td>
<td>XML Documentation</td>
<td>53</td>
</tr>
</tbody>
</table>
7.3.2 Annotated Documentation ...................................... 54
7.3.3 API Test Suite and Simulator ................................. 55
7.3.4 Better Granularity of Error Handling and Documentation . 55

8 Conclusions .................................................................. 57

References ........................................................................ 59

A Appendix ....................................................................... 67
A.1 List of Financial Terms ............................................... 67
List of Figures

1.1 Photograph of Wall Street ca. 1865-ca. 1895 ......................... 4
1.2 Calahan’s Stock Printer ............................................. 5

3.1 The CLICK System Overview .................................... 22
3.2 CLICK Architecture ................................................ 24
3.3 Generic RTR Architecture ....................................... 25

4.1 SAXESS Architecture Overview ................................. 28

5.1 CLICK Architecture .............................................. 34
5.2 Variable Information Message Format ........................ 38

6.1 Interface Standards ............................................. 42

7.1 Example of Annotation from the PHP Manual .................. 54

Figure 1.1 was taken from Andrew Dickson White Architectural Photographs Collection, Cornell University and is used within copyright restrictions for academic endeavours.

Figures 3.1, 4.1, 5.1, and 5.2 are used with permission from OM Technology AB, Stockholm, Sweden.

Figure 7.1 is taken from the PHP documentation project http://www.php.net.
# List of Tables

1.1   Option Styles .............................................................. 9  
1.2   Option Attributes .......................................................... 10  

2.1   Client/Server Delegation of Functionality ............................ 13  
2.2   Representation of the number 1025 ..................................... 19  

5.1   OMnet API Routines .......................................................... 35  
5.2   OM C Data Type Redefinitions ............................................ 37  
5.3   OMnet Broadcast Types ..................................................... 39  

6.1   Example SWIFT Message ..................................................... 43  
6.2   SWIFT Message Series ...................................................... 43  

7.1   Examples of OMnet Date Formats ........................................ 51  
7.2   Examples of OMnet Time Formats ........................................ 52  

Chapter 1

Introduction

Within any client/server system some type of interface between communicating components needs to be designed. Whether it be through adopting an existing standard or developing a proprietary one, this interface can become either a help or a hindrance in the further development of systems. As networking technology has come into its own, this area of study became a major focus with the computer sciences. Today this paradigm along with networking technology is a mainstay for the computing community.

An example of how one company developed a proprietary client/server interface lies in OM's development of the OMnet Application Programming Interface (API). The OMnet API is currently used by programmers developing third party clients for the CLICK financial exchange system.

1.1 Purpose

This report will examine and explain the current state of the OMnet API and suggest improvements from an external programmer's point-of-view. The API will be held up against good design practices including using appropriate abstraction levels, change management, access to developer tools like documentation and testing suites as well as error handling. Other more intangible areas like learning thresholds, time-to-market for software development, and ease-of-use will be discussed in API terms. Specific functionality will also be addressed. The API will also be compared to other financial interface standard conventions in order to best determine which direction future directions. Other proprietary systems will be quickly covered in order to give better picture of where electronic trading is today and how it compares to the CLICK OMnet API. Finally, suggestions concerning key developer issues will be illustrated.
1.2 Methodology

Initially a window of time was spent in order to introduce basic options details and trading within the markets in order to give a better understanding the CLICK trading platform. After a breaking-in period, a literature search was performed to find background information on the use of CLICK and electronic trading in the financial industry. This information then used as the basis for working further with the CLICK API.

In order to become more acquainted with the external functionality of the API, a test application was developed based on the two educational applications apitest and apisample. These two sample applications are normally provided along with the API headers and compiled code and were each developed on different platforms (NT and Solaris). Although it should be noted that each application has different functionality, and thus represent the basic processes that developers first encounter when coming into contact with the API on each of the supported platforms. Alongside this development, only the documentation that would normally be available to external developers was made available. In other words, only the information that is normally provided by the financial exchange and OM to a third party developer also known as an Independent Software Vendor (ISV) was made available during the writing of this report.

After having become acquainted with the API, several interviews were conducted with developers who are currently using the CLICK system. Each of the interviewees represented different aspects of the development process from an exchange technical director to a separate software vendor specializing in exchange clients. Along the way, other developers from within the OM organization also gave their input regarding system architecture and feasibility.

After having conducted interviews and experiencing the API first-hand, research concerning technical issues was done by searching through reasearch on already established technologies. Further recommendations were written based on research done by other independent workers in comparable studies.

1.3 Brief Historical Background Information

This section gives an historical summary of some of the most important events in the economic history of financial exchanges in order to give the reader some sense of context as to which roll they play as well as how technological modernization has changed how these markets work organizationally.
1.3.1 A Brief History of Financial Markets

Then, again, within the city, how will they exchange their productions? To secure such an exchange was, you will remember, one of our principal objects when we formed into a society and established a State.

Clearly they will buy and sell.

Then they will need a market-place, and a money-token for purposes of exchange.

Certainly.

Suppose now that a husbandman or an artisan brings some production to market, and he comes at a time when there is no one to exchange with him, - is he to leave his calling and sit idle in the market-place?

Not at all; he will find people there who, seeing the want, undertake the office of salesmen. In well-ordered States they are commonly those who are the weakest in bodily strength, and therefore of little use for any other purpose; their duty is to be in the market, and to give money in exchange for goods to those who desire to sell and to take money from those who desire to buy.

-From Plato’s The Republic

As far back as Plato’s time, markets were important centers of commerce. Peoples from far and wide found it easier to gather in central places rather than constantly travel from estate to estate to sell their goods as had been the tradition. Merchants would often travel with large amounts of goods to market and thus began to accrue larger and larger amounts of currency for trading transactions. Being wary of attacking robbers and bandits, the necessity for credit was born. Promissory between-merchant notes based on bills of exchange became common practice. When cash-on-hand was needed, these notes were then sold to the local money-lender. Throughout this process, charges were levied for these services. From these meager beginnings, the need for financial markets was created and self-financed [Van den Abeele 1988].

Modern financial markets soon developed out of necessity as the populations of city centers began to grow. In the 14th century, merchants began gathering in Bruges (now in modern Belgium) in front of the house of the Van Der Beurse family. In time, the family name was transformed into "bourse" now part of the modern French translation of stock exchange "bourse des valeurs" as well as many other germanic languages.

Exchanges later became more standardized and centralized as traders moved or were booted out of London coffee houses [LSE 2003] and into buildings like that of the London Stock Exchange (1773) which was specifically built for the purpose of trading stocks. In 1792, the Buttonwood Tree Agreement was signed by 24 American colonial businessmen under a tree in New York City on the now-famous Wall Street (shown in figure 1.1). This now famous landmark was named after the street
that was built upon the 12 foot (ca. 4 m) high stockade used to protect Dutch settlers from British and Indian attacks. This agreement specifically gave preferences to the signers in much the same way that the London Exchange brokers refused to allow entry for those security dealers that they did not like. Thus a sort of exclusivity or limited membership became the norm for such institutions[Somntag 1997].

Later modern exchanges began to flourish in other European countries like Germany, Belgium, Spain, Portugal, and Sweden.

As exchanges grew and modernized, technology began to be used as a means of communicating timely information to members whose staff could not always be physically standing on the market floor[Cukier 1999]. The Calahan stock ticker (shown in figure 1.2) was first introduced by the American Telegraph Company to the New York Stock Exchange in 1867, in order to meet this growing need. Even then technology firms competed to improve upon this simple manner of distributing stock prices.

None other than Thomas Edison made substantial improvements to this device and began production of more than 5,000 units of his new device[Stock Ticker Company]. By 1878, the first telephone was commercially introduced just two years after Bell conducted his successful tests in Boston. This revolutionized even trading itself [NYSE], by influencing the way traders interacted. No longer did the trader have to wait as messages were couriered back and forth from customers. Instead orders
could be relayed by wire. Even in 1881, when the electric price board was introduced, technology was not seen as a threat but more as a seamless extension of the market itself.

By the end of the 1960’s necessity called for further automation. The markets had become so large that by 1968, a paperwork crisis ensued[1]. Exchange member firms had to work around the clock as they struggled to process trades in a timely fashion. As a result, automation became an encouraged practice.

![Figure 1.2. Calahan’s Stock Printer](image)

It wasn’t until 1971 that established physical stock exchanges had to face true competition from virtual electronic ones. The United States Securities and Exchange Commission (SEC), worried about the large growth of unlisted (so called over-the-counter or OTC) stocks being traded outside of the already established markets, gave the directive for creating a new electronic market. The National Association of Securities Dealers Automated Quotation (NASDAQ) linked the computer terminals of over 500 Market Makers and created what we know today as one of the biggest wholly electronic financial exchanges.

Even more challenging for the more traditional auction stock exchanges, like that of NYSE and AMEX, has been the emergence of Electronic Communication Networks (ECNs)[2]. An ECN automatically matches buy and sell orders from institutional investors, broker-dealers, and market makers. However, the individual investor can enter orders in the market more directly by having an account with a broker-dealer[3], something traditionally only exchange seat-owning members could do. ECNs can also easily create specialized markets and vary their business logic because they are not nearly as closely regulated as more the traditional exchanges. This however may change with the continuing interest of exchange commissions in protecting individual investors. ECNs can also lessen overhead by using large computer networks in order to bring special interest groups together. SelectNet is an example of these traits and is primarily used for trading between Market Makers. This network has the ability to allow preferential orders over those of other traders[4]. Also very popular are the many after-hours exchanges that

---

1. Warr 2002
2. Clark 2000
3. US SEC 2000
4. Investopedia
trade in traditional (ie NYSE) market securities[US SEC].

1.3.2 A Brief History of the Options Market

While technology facilitated the growth of the stock markets, the commodities market (grains, soybean oil, etc) was attempting to grow in parallel. During the start of the 1970’s, commodities brokers at the Chicago Board of Trade were looking for a way to boost the waning futures exchange (which oddly enough ended by 1973 with a very unexpected record-shattering trading volume). As a result, instead of just creating futures derived or based on crops, they experimented with futures reflecting listed stock prices. On April 26, 1973, the newly formed Chicago Board Options Exchange (CBOE) opened with 16 listed stock options with brokers setting the price any way they wished. Two years later, the American Stock Exchange (AMEX) and the Philadelphia Stock Exchange also began trading equity options. These exchanges cemented the growing decentralization from the predominant American center of commerce at the time in New York City. It wasn’t until 1985, that the New York Stock Exchange (NYSE) started to offer options[Osten 2003].

By 1987, the options market had reached its peak just as a the stock market crashed on the now infamous "Black Monday". Ironically, one of the commonly believed causes for the exacerbation of the 1987 stock market crash was from automated computer trading technology proving that even technology had its downside[Doser 1992]. As a result, suspension of automated trading occurs as soon as certain criteria called collars or circuit-breakers [Osborne 2003] on the NYSE are met.

In Europe, options exchanges also came into their own. In 1982, the London International Financial Futures and Options Exchange (LIFFE) was established in order to hedge foreign interest rates.

Slowly during the 1990’s, the options market began to again gain momentum after the markets caught their proverbial breath; however, the mix of actors in the market began to change. European financial markets began to become more interested in this now established financial tool and added their own attributes and trading styles. In comparison, older actors like the NYSE divested from the options market and sold off their options business in 1996 to the Chicago Board of Options Exchange (CBOE). [Schaeffer's Research]. European markets also adopted different exchange strategies in order to attract more long term investors and moved towards more open electronic trading platforms.

Later, in May 2000, the International Securities Exchange (ISE), entered the market with the first all-electronic options exchange in the US becoming the first new exchange in 27 years[Ruffy 2002].

1.4 History of OM

As options grew in popularity in the US, European markets also began being interested in equity options. In 1985, OM with it’s founder, Olof Stenhammer, started
offering a small number of options to the Swedish market and eventually became the first commercially operated exchange. By 1990, in parallel, the Stockholm Stock Exchange became a fully computerized exchange, a process which had started as early as 1974. Even at this point, OM was actively developing proprietary software to account for the newer financial products being offered. As a result, the internal system they developed became known as the first version of the CLICK trading system. The CLICK system then itself became a product and was sold to other exchanges such as the Austrian derivatives exchange, ÖTOB, and many other countries with blossoming options exchanges. By 1997, OM and the Stockholm Stock Exchange merged in order to solidify Stockholm as a strong financial center for Scandinavia. This extended OM’s experience further by working even more closely alongside a diversified exchange and gave OM a unique position in the financial exchange software market.

At this point in time, software vendors were in a completely different and separate markets than that of exchanges. Exchanges that had already established themselves usually achieved a critical mass that required internally developing software to meet their specific needs and trading styles. These exchanges, however, had no intentions to adapt and package their systems for further sales to other exchanges. OM, however, continued to focus on the technology/software side and eventually departmentalized this part of their organization. OM would then develop software for every vertical sector in the market such as exchanges, clearing houses and brokerage firms.

The company also gained a unique experience by working closely alongside a live stock and options exchange. Initially focusing on Europe, OM and the London Stock Exchange (LSE) announced a partnership in December 2002, in order to create a new options exchange located in London. Again, OM’s CLICK trading platform is being used as a basis for this exchange[LSE 2002]. Production for EDX London started June 2003[EDX London]. OM has also gained a strong foothold in the US financial trading software markets with customers like the International Securities Exchange (ISE) which, as of May 2003, is the largest options exchange in the US [ISE 2003]. OM also has plans to invest in the Asian, as these options markets have reached a critical mass and are now looking to electronic trading systems.

At the time of this report, Q2, 2003, OM has announced a merger with Helsinki Securities and Derivatives Exchange (HEX) which also operates the Latvian and Estonian Stock Exchanges. Under the new name OM HEX, the company hopes to further the idea of a unified Nordic exchange and gain further experience and markets for their transaction technology[OMHEX 2003].

1.4.1 OM Systems

OM currently develops two main platforms for financial exchange systems. As mentioned, the CLICK system was one of the first financial exchanges that OM developed and is currently being used in several exchanges (mainly options markets) around the world. The second financial platform currently being used by several exchanges
and being actively developed is the SAXESS exchange trading system. This system was originally launched in April 1999 and is normally used for stock exchanges. As with CLICK, this system is continually being developed and customized for various customer requirements. Both are based on slightly different design concepts for external and internal communications as well as systems architectures. More detailed discussion about the architecture and a deeper focus on external communications will follow in the next chapters of this thesis.

1.5 Options

Although the focus of this thesis is not on the options market specifically but more on the software applications facilitating the exchanges themselves, it is still quite helpful to understand options and financial instrument trading. How options are used as financial instruments is outside the scope of this thesis.

1.5.1 From Futures to Options

Traditionally, when a farmer planted his crop, he didn’t know which price he was going to receive when, several months down the line, his crop was brought to market. Since regional markets only occurred a few times during the year, the immediate supply whenever almost every farmer brought his goods to market would far outweigh demand. Buyers from the central graineries could then press the prices down so far that farmers could no longer make a profit on their labors but instead would most likely incur heavy losses if not bankruptcy. To solve this problem, the Chicago Board of Trade, in 1865, created "futures" contracts. Futures were traded for a commodity like that of grain and other crops to be delivered some time in the future. The price was determined when the contract were signed. These contracts were normally traded long before crops were due to come to market. In this manner, the system kept grain from flooding the market at any one time and this stabilized prices[CBOT].

Options, in turn, evolved from these futures markets based on other underlying instruments instead of the more traditional commodities. Examples of today’s options range from simple stock options to options based on weather, electricity, and insurance. The main difference between options and futures is based more on the obligations incurred rather than the actual underlying instrument. For futures, the buyer and seller are both obligated to perform the contract either by compensation or actual delivery of the underlying instrument. In contrast, with options the buyer has the ability to demand delivery of the underlying option or may exercise the contract. However, should the buyer not wish to use this right, they can instead forfeit the price which they paid for the option and nullify the contract[Institute 1998].
Style | Description
--- | ---
American | The option holder has the right to exercise the option on or before the expiration date of the option.
European | The option holder can only be exercised during a specified period prior to its expiration.
Asian | The exercise price is equal to the average price of the underlying asset over a given period.
Exotic | Various other styles based on a whole gamut of variations and combinations. Examples: Bermuda, Asian European, Asian American and Quanto.

Table 1.1. Option Styles

1.5.2 The Gory Details

Options are traditionally used as a financial instrument to spread risk or speculate based on the price of the underlying instrument (stocks, etc). An option usually has control of units of 100 shares of a certain stock although this can differ from market to market depending on the exchange. The major difference from buying the underlying stock outright is that when buying an option only the premium/cost for the right to buy and sell is paid which is based on only a small percent of the actual cost of the underlying stocks. Thus, the person buying the contract has so-called leverage over these stocks. This option then in turn can be bought and sold on the market. In this manner, the option has an intrinsic value.

Puts and Calls

There are two types of options. The first, and most common, is called the call option which enables its holder the right to buy or call an underlying security. The other, a put option, enables the buyer to sell an underlying security. Each of these options has a strike or exercise price at which price the shares can be bought or sold. To exercise this right is to buy/call or sell/put the underlying shares and thus pay or sell at this indicated price. Most exchanges also have a certain "style" of option contracts which determine when the option can be exercised. Several of the different option styles are shown in table 1.1.

Options can be classified into certain categories based on these aforementioned styles. All option contracts having the same style and underlying security are said to be a "class of options". All options within this class that have the same unit of trade at the same strike price and expiration date are grouped into an option series[CBOE 1996], a common term encountered by the exchange developers.

1.5.3 Electronic Trading in Options

Options aside, there tend to be two models of financial instrument trading used in today's exchanges. The older of the two, the auction or open outcry system, builds
Term | Description
---|---
Type | Whether the option is a put (sell) or call (buy) 
Underlying Instrument | The financial instrument upon which an option’s (derivate) security price is based. A typical example is stocks. 
Expiration Date | The date on which the option contract ends. 
Cash / Physical | Whether the contract demands physical delivery of the underlying instrument or whether cash compensation is instead expected for the difference in value if the exercise price had been 
Exercise Price / Strike Price | The price at which the option holder can buy or sell the underlying instrument 
Premium | The price paid for the option itself. 
Style | The details usually concerning when the option can be exercised.

Table 1.2. Option Attributes

upon exclusivity of its members. This style is used in traditional exchanges like the New York Stock Exchange (NYSE). Entry into market is based on buying a membership or seat in order to own the right to trade in the market traditionally known as the *market floor*. There may be only a certain number seats available and thus the cost of membership is typically quite high. Representing these members are *Specialists* who usually only trade in a certain instrument but may trade in group of low-volume instruments. These specialists then trade with external brokers from their own inventory. Most of the trading on the floor is then short term for perhaps the maximum window of a day. Trading for long term investors usually happens outside of the market at larger financial institutions. As a result, specialist firms may even pay brokers in order to receive their orders. This payment-for-order-flow concept has been heatedly debated within the recent months and may be later banned by the United States Security and Exchange Commission [Forbes 2003].

Another variant is the more modern electronic orderbook trading platforms (not to be confused with Electronic Communication Networks). This system can be described as sort of "democratization" of the traditional out-cry system. The threshold for entry into the exchange market is much lower than the out-cry system. Membership is open to anyone who can meet certain financial criteria as well as other exchange specific regulations. Here specialists are replaced by the slightly different roll of *Market Makers* who in turn for receiving lower transaction fees, have certain privileges but still retain some responsibilities. In electronic trading platforms, Market Makers may have priority for the best deals in the market, but in turn they must maintain a certain maximum spread. A spread being the difference between the bid and ask price for a given financial instrument. Unlike the out-cry system, Market Makers are not the only actors that have direct access to the market. Brokers, institutional investors, and even traders may become members assuming they also meet
certain criteria. Broker/dealers in this system tend to have higher transaction costs and as a result have no responsibilities in or to the market. Variations on this model tends to be used by most of the newer more modern computer based exchanges like those found in Europe.

The typical trading cycle in a traditional out-cry exchange looks as follows:

1. A broker receives an order from a customer either directly or through even another branch within the brokerage.

2. The broker sends the order to the trading floor.

3. The trader goes to a specialist to find a counterpart on the floor or on the specialist’s electronic trading book.

4. Either the transaction is completed directly with the specialist of the trader completes the transaction by buying and selling from his own firm's inventory.

In contrast, the trading cycle (without a Market Maker) looks as follows when the seller initiates the transaction:

1. A seller contacts his broker and requests to sell from his portfolio that this broker holds for the client

2. The broker at that point advertises on the electronic market place that this client wishes to sell a certain number of shares, options, etc.

3. The information is then distributed to all the other brokers electronically who are actively watching the exchange.

4. A buyer through his broker then decides to buy at the advertised price.

5. The order and the buy are then coupled together by a matching mechanism within the exchange system and this information is returned to both brokers.

6. At the end of the trading day, the orders are "cleared" by the exchange meaning that the ownership of the financial instrument changes and the process starts over the next day.

The real benefit of electronic trading comes when customers, being the fickle investors they are, want to change orders or for some reason an unforeseen error was introduced. Within the open outcry system all orders are out of the hands of the customer as soon as it has left the broker’s desk. The difference with the electronic orderbook system is that the customer is much "closer" to the market and can thus change the order more quickly compared to the open outcry model. In the outcry model the customer would have to tell their broker, who then in turn tells the trader, that then may or may not have the chance to take the order out of the market in time. Open orderbook exchanges also bring a sort of new democracy to the market.
considering that almost everyone has a relative equal footing to get a chance at the best deal.

It should also be noted that electronic marketplaces do have their critics. On some exchanges analysts have noted that all orders do not necessarily enter the electronic marketplace, but can instead just be reported to the exchange. The exchange merely takes care of clearing and settlement for that derivative instrument. In other words, the exchange administrates the invoicing and transfer of assets (clearance) but does not actually advertise the order on the open marketplace. This so-called side trading has allowed for misgivings of the electronic order system. The anonymity of these types of orders that tacitly affect the market, but never are truly announced to all the market actors cause suspicions among its critics. Other more computer-human-interaction issues like brokers fearing mistakes made by "pushing the wrong button" have also surfaced. Brokers, traders, and Market Makers have also had to rethink their market instincts considering that human interaction is at a minimum and instead information comes directly from what seems like computer terminals just spouting numbers[Leib 2000].

**Exchange Customization**

Exchanges can follow different models when trading. In reality, every market bases their trading rules on variations and combinations of the two above mentioned exchange styles. Furthermore, there can even be more complicated rules concerning other areas of trading like for example, who may get a chance to bid first, how market makers are allowed to carry out their business, how much trading traffic may come from an individual broker, etc. This along with practices outside the market like payment-for-flow and preferential treatment for larger customers greatly affect how an exchange works in reality. This information is usually spelled out very precisely by the exchange commission’s trading rules but may even be more subtle and include in other country regulatory laws or even simply tradition. As a result, when exchange systems are developed they normally have to keep some acceptable range of flexibility in order to be able to meet each country’s or exchange’s regulatory framework. This creates certain problems when selling an exchange system to such a group of diversified customers. This usually results in customization for each and every customer who purchases the system. This cost is normally incurred by the buyer of these systems but is normally developed by the software providers for the exchange itself.
Chapter 2

Client/server programming

This chapter covers background information concerning client/server computing in order to give the reader a basis for later discussion.

Client/server architecture is essentially about using processing power of a server to achieve an optimal system. In more simplified terms, it is some sort of server software or process that accepts request for data from client software and returns some sort of information. In today’s terms, the client may manipulate this data either through internal functionality or through user input. There is little doubt that client/server systems meet the needs of most applications today when accommodating proprietary services to be used by external clients. Also client/server programming makes it possible to use computer processing power more efficiently to allow for larger and more complex systems.

2.1 N-Tier Architecture

During the early days of modern computers, dumb terminals accessed programs and databases on one central mainframe computer via dedicated cables. This came to be known as a single-tier architecture. As computing and computers matured, two-tier systems became more feasible. Smarter clients connected to more powerful central serving computers where the database or other more advanced software still

<table>
<thead>
<tr>
<th>Client Functions</th>
<th>Server Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>File, Database Server</td>
</tr>
<tr>
<td>Distributed Application Processing</td>
<td>Distributed Application Processing</td>
</tr>
<tr>
<td>Local application</td>
<td>Communications</td>
</tr>
<tr>
<td></td>
<td>Network Management</td>
</tr>
<tr>
<td></td>
<td>Resource Management</td>
</tr>
<tr>
<td></td>
<td>Configuration Management</td>
</tr>
</tbody>
</table>

Table 2.1. Client/Server Delegation of Functionality
ran. These clients had the ability to support a user interface as well as other local software. This two-tier approach had some inherent limitations, though. Client software had to be distributed and managed for a potential large number of users which potential were not located in the same physical location. If the back-end software was updated, or changed in a vital manner, the procedure of rolling out a new client then became a costly and challenging event. Not to mention, that if the back-end system ever went down, this architecture could lead to system downtime and data loss.

As a result of these limitations, the two-tier system evolved into the \textit{three-tier} system. A middle tier was implemented that added functionality between the user interface environment and the server environment. After this three-tier approach which traditionally referred to the back-end service being almost entirely a database, more recent architectures could be viewed as consisting of a multi-tiered architecture or simply a three-tier architecture with add-ons.

\textbf{Database Server} This is the engine that stores the data needed for the system to be able to deliver the service needed.

\textbf{Application Server} This is the actual execution environment used to typically contact the database. It is on this level that most of the login within the systems application is carried out.

\textbf{Transaction Processor (or Middleware) Monitor} When a transaction is accepted, this middleware queues it and take responsibility for its completion. It also may handle security issues and this shield the back-end services from giving up critical information. Using this type of solution usually gives more scalability than a two-tiered system.

\textbf{Client} The service provided to the end user to allow them to access information and perform transactions on a back-end service.

- \textbf{Traditional Client} A client where data processing is also located to some extent within the client.

- \textbf{Thin Client} A client where the bulk of the computing is located on the server. Typically only the user interface is located on the client computer.

It has been shown that using this technology improves performance for groups with large numbers of users. It also allows for cross-platform flexibility considering that the interfaces between the server and the client can be host independent. In this manner, computer resources can be used more efficiently within limited computing resources. Limitations may include development complexity due to meeting organizational needs.
2.2 Event Handling

As client/server computing has evolved, focus has concentrated on a more fine-grained approach of optimizing communications between processes rather than overall client/server interaction. Experienced software developers are now examining means for efficiently designing distributed systems (i.e. applications which separate functionality into different components) on a larger scale in order to allow standardized communication between the various components. One approach that is often used and is being actively researched is event-based or implicit invocation communications. In event models, there are traditionally two roles into which applications or processes fall. The event supplier creates event information and a event observer or consumer which receives the event. So, rather than using interprocess procedures, a component announces or broadcasts event data either directly to a client or in to a general space. Consumers of events register or subscribe to a service in order to receive any events of interest that match a channel or pattern depending on the filter type. When the consumer is no long interested in the event data, then the process unsubscribes to that event service. There may or may not be an intermediary, an event channel, which facilitates broadcast distribution and allows anonymous communication between components.

2.2.1 Typed Events

Events themselves fall into two main types or categories generic or typed. Generic events consist of a data blob without any explicit structure. This allows an easy to implement communications, but in terms of efficiency is extremely poor considering that the application must then determine the format of the event. The other event category includes typed events where well-defined data structures are predetermined. Usually, this is done by a commonly agreed upon interface with set in-parameters. Typed events can range from simple to more complex formats and can include separate user-definable parameters. By using typed events, the implementation can also utilize filters. Here the opposite can be true, where efficiency is optimal but implementation is more effort.

2.2.2 Push and Pull Models

The actual communications between components can be classified with two models, a push model where the supplier pushes out events to subscribed consumers and the pull model where consumers request events from event suppliers. The push model has the advantage of avoiding excess networking messages. Unlike the pull model, it does not need to constantly poll events. As a result the pull model tends to be used less in high performance/volume communications.
2.2.3 Event Channels

Within some implementations of event services, a middle role of an event channel is used. An event channel decouples the direct connection between event supplier and consumer and instead acts as an intermediary object cloud. This then allows a sort of anonymous communication to occur since the clients need not know any specific addresses or ports in order to acquire the information to which they wish to subscribe.

2.2.4 Event Filters

An event-based system may contain several suppliers which produce information of different means which can result in a rather large amount of data within the system. As a result, event filters are used to control which events of interest a client wishes to receive. Filters enable consumers that ability to specify those constraints that might be of use only to that specific client. Events are then matched before they are propagated into the system. Filtering in larger systems is extremely important to achieve information granularity desired by specific consumers because of the potential large amount of event data being communicated.

2.2.5 Non Functional Constraints

Within an event system even non-functional services like real-time constraints, event delivery guarantee, event synchronization, and quality of service (QOS) become desired services. These issues usually tend to be implementation specific, but are often covered in several codified event models.

2.2.6 Formal Event Models

Currently, there are several formal codified event models, but in general, they all in some way address the aforementioned issues. They include:

**OMG Corba Event Models** These event models usually vary based upon specific implementation, but do follow the general guidelines set out by the Object Management Group (OMG).

**Java Event Models** Event models to handle GUI components like the Abstract Window Toolkit (AWT) and the Distributed Event Model Architecture that employs event notifications by way of event listeners.

**Scalable Internet Event Notification Architecture (SIENA)** Designed to support event notification in wide-area networks (WANs).

**COBEA - COrba Based Event Architecture** An event service designed to extend the Corba Event Model to support fault-tolerance, access control and composite events.
2.3 Synchronous Versus Asynchronous Programming

Synchronous programming communication between processes is unbuffered, and processes wait until the data between them has been transferred.

Asynchronous programming communication between processes is buffered using buffers of unlimited size. The sender never waits after sending data. The receiver only waits when its buffer is empty [Schooten 2001].

As web services have become more popular, asynchronous programming has become of primary focus for providing services within several programming frameworks like that of Microsoft’s .NET. This has also resulted in network programmer’s looking to formalize communication styles in terms of programming paradigms. Other issues like use of threads has also been drawn into this debate because of their overhead and programming challenges.

Both ideologies stem from the design problems concerning how to best manage concurrent input/output (I/O) data. The most common example used in literature tends to be the updating of a database problem. Suppose a system is based on several clients that query and update a single global database. If several of the clients then have simultaneous requests coming into the database, issues such as who gets priority and should that request block other requests becomes an implementation problem. Past that, other issues such as updating the same data within the database becomes a design consideration.

With asynchronous or sometimes called event-driven programming there are two main techniques used in order to solve some of these issues. First, one can fork off a process by starting a new process which handles each request. This simple implementation has the drawback of not scaling well because each request then starts a new kernel process which in turn has a harder time leaving an exit status. Secondly, threads are often used as an implementation. A new thread is then created, processed and rejoined to the main process for every request received rather than forking. While this implementation solution creates less overhead and is quite easy in object oriented languages, it also brings about complications like shared resource handling that easily lead to crashes. Shares resource collisions are normally solved by letting the forked process or thread wait a set amount of time and then checking to see if the resource is available. Most asynchronous systems require some sort of state mechanism in order to connect requests with replies.

On the other end of the scale, synchronous programming entails handling each request, one at a time, and in turn. Although this might be simpler implementation, the major drawback encountered is that a process is blocking communications during that time that the request is being processed. In time critical systems, this can lead to problems in gaining information fast enough to make informed decisions. Synchronous programming also allows a process to return data which it then can use to access other data. No process can start if the system is blocked and thus resource sharing is not a problem.
2.4 Endian Programming

It is allowed on all hands, that the primitive way of breaking eggs, before we eat them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us, there have been six rebellions raised on that account; wherein one emperor lost his life, and another his crown. These civil commotions were constantly fomented by the monarchs of Blefuscu; and when they were quelled, the exiles always fled for refuge to that empire. It is computed that eleven thousand persons have at several times suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have been published upon this controversy: but the books of the Big-endians have been long forbidden, and the whole party rendered incapable by law of holding employments.

- From Gulliver's Travels, by Jonathan Swift

As hardware devices were being developed, two ways of storing data came to pass. Just like Jonathan Swift's egg predicament explained above from Gulliver's Travels, developers had to decide from which end of a binary word to start reading. This idea of which end of the egg should be cracked was pick up upon by Danny Cohen in 1980 who wrote the now infamous On Holy Wars and a Plea for Peace[36] concerning the heated dicussion on proper byte order.

Binary data refers to the practice that all computing is based upon of storing data as series 0 and 1s. Every bit which is either a 0 or a 1 is stored alongside 7 other 0 and 1s making an 8-bit byte. The difficulty is whether the most significant value is stored in the lowest (first) storage address (big-endian) or whether the least significant value is stored first. See table 2.2 for an example using the number 1025.

For example, the number 1025 (2 to the 10th power) would be represented in binary as follows:

<table>
<thead>
<tr>
<th>Representation of the Number 1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 00000000 0000100 00000001</td>
</tr>
</tbody>
</table>

As networking became more of an issue, the whole idea of endianness needed to be addressed so to speak. Since some platforms had decided upon big-endianess like that of Sparc and MIPS, others had chosen little-endianess like Intel. Direct communication between the two endian platforms became less than transparent. (PowerPC's on the other hand have chosen to implement both version and thus are bi-endian.) On top of this problem, bit ordering within each 8-bit byte can also be big- or little-endian. As a result, this causes massive confusion when two platforms of differing endianness wish to transfer binary data in integer form between one another.
<table>
<thead>
<tr>
<th>Address</th>
<th>Big-Endian 1025</th>
<th>Little-Endian 1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000000</td>
<td>00000001</td>
</tr>
<tr>
<td>01</td>
<td>00000000</td>
<td>00000100</td>
</tr>
<tr>
<td>02</td>
<td>00000100</td>
<td>00000000</td>
</tr>
<tr>
<td>03</td>
<td>00000001</td>
<td>00000000</td>
</tr>
</tbody>
</table>

*Table 2.2. Representation of the number 1025*

In order to solve this problem, platforms that do not follow the standard to which they are delivered data must swap bytes in order to correctly read the integer data. This changing of bytes is done using a macro or by using native functions for the given platform.

### 2.4.1 In Combination with ANSI C

However if this were the only problem then byte swapping would be an easy solution. Unfortunately this is not the case. For example, ANSI C does not define the size of all types of data. Integers can either be 16 or 32 bits depending on what is optimal for the given platform. When it comes to the different types of integers, ANSI C is even less specific.

The sizes defined by the ANSI C specification, which is:

- short == 16-bits
- int == at least 16 bits, can be more (i.e. 24 bits, 32 bits, etc..)
- long == at least 32 bits, can be more (i.e. 36 bits, 64 bits, etc..)

This issue combined with endianess tends for developers to write error-prone code. Developers have three options open to them which can handle data storage in endian neutral formats [HP 2003].

1. Store data in an exactly defined endian format
2. Add additional data to indicate to which format the data adheres
3. Store data in ASCII format.

The first of these options obviously gives the best performance; however, even the third has been used successfully within financial software. ASCII also gives the developer the ability to easily read input/output from a given system. This should not be underestimated when working with complex systems.
Chapter 3

Overview of the CLICK Trading Platform

As mentioned earlier, the CLICK system is the first of the software systems that OM packaged into a full-fledged product. In short, the CLICK trading platform is a set of modules each designed to fulfill a certain function within the system as a whole. Each of these modules then communicate with each other using an API named OMnet. It is a subset of this API that is also used by external developers to create clients to interface with Central Marketplace and the Deal Capture functionality. OMnet is available on several platforms thus allows independent software vendors (ISVs) to develop for their client specific operating systems.

Based on a client/server architecture, client applications located at an exchange’s member site (typically a bank or broker) can interface with the back-end systems (see figure 3.1). In an effort to maintain portability, the code is written in ANSI C and C++.

The system contains several central core subsystems, a back-end database, and the communications system that connects them together. The subsystems can be categorized as follows:

- Marketplace software
- Base software
- Clearing & Settlement Software

The last area will only be covered in passing because these modules belong more to the SECUR clearing system and not the trading platform in and of itself. Although, it should be mentioned that even this area can be interfaced with the OMnet API.
3.1 Marketplace Software

The marketplace, in general terms, is where information is stored and updated about the current state of the market. It is here that options are electronically bought and sold in the form of orders. This concept of ordering is translated into computer terms as requests or transactions. Other internal modules such as Local Modules (LM) which takes care of market maker supervision and creating reports and the Supervision (SU) module are included here. However, the two most important modules are the Marketplace (MP) itself and Information Dissemination (IN) module.

3.1.1 Marketplace (MP)

It is within the marketplace that bids (buy orders) and asks (sell orders) are made for a particular financial instrument. These orders are then ranked and held in an order book. When two orders agree that is one trader is willing to buy and another willing to sell at an agreed upon price, the two are matched and the deal is closed. This module is the central and most business critical part of the CLICK system.
As a result, any other type of functionality that does not fit this profile is located outside this module in order to maintain a rapid electronic marketplace. However, even other functionality like information distribution based on these trades is a close second in being a critical business component.

3.1.2 Information Dissemination (IN)

This module includes software to cover the functionality of getting information out to clients in one of two ways. This is done via messages called broadcasts. There are in general two types of broadcasts:

- **Dedicated** where information is meant only for a specific client/user or even a group of users like that of a member with several brokers or market makers

- **General** information is sent to all currently connected clients

An important part of this module is the OMnet Distribution Utility (OMdu) process. This system takes care of information objects that are exchanged between an OMdu server and the connected applications. The communications now work over low-level local socket connections although earlier other methods like NT pipes, DECnet and RTR had been used. The implementation of receiving information is included in the external OMnet API available for clients. The OMdu server also filters traffic so that internal information is never present on a member site.

In the exchange market, information distribution is a critical component to any system. Regulatory pressures as well as demands from competition create the need for high-performance distribution of exchange data. As a result this module needs to be able to send out information at a high rate. A module of this type is usually a component of almost every financial exchange system.
3.2 Base Software

Perhaps the most important module of the base software is the Common Database (CDB) which stores all the information about a trading day also called clearing. Settings information about environment variables is also a part of this base software. General Utilities (GEN) also are included here which cover the API as well as time implementations as well as list administration.

3.3 Clearing Software

After the end of the trading day, orders are processed and financial instruments change hands. This administration comes directly from a software system called SECUR. SECUR controls and administrates all of these type of events.

3.4 Architecture

CLICK uses a three-tiered architecture utilizing a transactional processor as middleware. The three tiers include a client utilizing the OMnet API that in turn connects to a gateway. The gateway in turn contacts via a back-end the O Mex central system. Figure 3.2 shows the basic interfaces between the three tiers.

![CLICK Architecture Diagram]

For the terms of the paper, the gateway will be referred often as the front-end and the central system the back-end of the system since this is how an external developer understands the system from their point of view. The client and gateway are generally located at a member’s (broker/bank) physical location while the internal exchange system is located central at the exchange.

3.4.1 RTR

Reliable Transaction Reliable Transaction Router (RTR) (http://h18004.www1.hp.com/products/software/ntenterprise/rtr/) is a fault-tolerant middleware used to insure data transactions. Developed by DEC originally, the software was then taken over by Compaq when the company was sold and then onto Hewlett Packard when Compaq themselves merged with HP. RTR is used to ensure
that when a client disconnects for any reason, that the transactions the client has sent are transported to the back-end of the system. In the same way, if the server should go down for any reason all transactions are then held by the RTR until the server then recovers. This also applies to network links. Figure 3.3 shows the different processes that the clients, servers and RTR each manage. In this manner, RTR tries to meet requirements for continuous computing services that most financial exchanges require. RTR also attempts to simplify design issues by already having implemented functionality that can be combined with a company’s own proprietary API.

![Figure 3.3. Generic RTR Architecture](image)

In the CLICK system, the RTR system is also used to achieve enough inputs to meet the demands of large exchanges. For example, CLICK has been run with 120 gateway nodes connecting to over 1000 nodes at member sites[HP-RTR 2003]. This is done by easily adding nodes within the RTR architecture. By doing so, services can be spread out to insure better stability and dependability. RTR also provides for message content routing in the form of partitioning.

The RTR API has a C and C++ interface which are used to map data elements and structures for transporting transactions. The RTR API is used rather frequently within the OMnet API in order to provide the client with routing transparency. CLICK uses the RTR server in order to re-route client requests to different primary services in order to achieve load distribution.

RTR, while being a very helpful technology, does have it downsides. Server applications are required to build logic to handle uncertain transactions which makes these applications more complex to write. The time it takes to start a server increases also because of the recovery time it takes to repeat roll-backed transaction transactions[McCormick Tu 1999]. Although this would be a downfall that most financial exchange systems would find acceptable.

RTR is currently available on Windows®, Tru64 UNIX®, Sun Solaris, and OpenVMS platforms and at the moment of this report is available in a beta re-
lease for Linux. The router has an API in Java, C, and C++ for development of distributed applications. Hewlett Packard is actively developing the RTR and updates have continued regularly. Along with this new Java API, Hewlett Packard is even marketing this product as being capable of web services. However, the design considerations are based on the existing Java API and not on any implementation already developed. In other words, it is still up to the software developers wishing to create a service to do the implementation on top of the RTR API.

3.4.2 OMnet

As mentioned earlier, OMnet API is the interface used by the CLICK system to guarantee interprocess communication. It is used both internally and externally and is written predominantly in C. Essentially, the API is very similar to a protocol that merely transfers the data given that it is in the correct format. Since C is a strict typed language this is done in the language structure itself. The API does have certain functionality within it that an external programmer works with constantly. It covers user authentication which is set in the Central Database (CDB) and the authorization when users have correctly given login identification and passwords. It also interfaces between the aforementioned RTR system and client applications. This functionality is implemented in the gateway of the CLICK system. The API also handles communications with OMnet Distribution Utility (OMdu) which handles the distribution of broadcasts which tend to mainly consist of market data.

A more in-depth discussion of the OMnet API is found in chapter 5.
Chapter 4

Quick Overview of SAXESS

This chapter quickly introduces OM’s other electronic trading platform SAXESS in order to give the reader a better background when comparing other client/server solutions to that of the CLICK trading platform.

4.1 History

The Stockholm Stock Exchange (SSE), in an attempt to meet the every-growing trading volume, in June of 1989 brought into production the Stockholm Automated Exchange (SAX). This system allowed brokers, for the first time, to trade from any location instead of having to have personnel physically located on the exchange floor. Originally designed to handle 6,000 trades per day, the system had an over-capacity to handle the 1,000 daily trades that passed through the market at that time[Mårtensson 2000]. However, by 1992, the market started seeing considerable growth, and the demands on the existing system began to change. In numbers, the SSE had grown to new financial heights from 172 billion Swedish Kronor in 1992, to 918 billion Swedish Kronor by the year 1996. By 1997, the market had reached a capacity of 50,000 trades per day, well outside of the original system demand criteria. Still, SAX could handle this capacity by scaling up it’s resources, but performance was slowly deteriorating with the ever-increasing trading volumes.

Relaxation of financial legislation, new financial instruments and new recognized practices, as well as members introducing automated trading, changed the demands of the exchange on the software system. As a result, the current version of the system needed to be redesigned in order to handle the massive growth. The new version was to be called SAXESS. Performance and flexibility became the main goals of the project, and by April of 1999 the new version was launched.

Today, this system is predominately used by stock exchanges, but can even be used within options markets or other financial markets. The current production platform is Sun’s Solaris, but the system is also portable to any other POSIX-compliant UNIX platform and other platforms like AIX, Linux, and Windows NT.
For the purpose of this thesis, only the architectural and communications considerations that can be directly compared to CLICK’s external OMnet API will be discussed.

4.2 System Architecture

As with any well-thought-out critical business system, SAXESS is based on redundancy as well as fault tolerant technology. However, unlike the CLICK system, this technology is not based on external proprietary software like that of the previously mentioned RTR. Instead, clients are allowed to connect directly to a front-end process that is actually contained within the system itself or clients can reach this process via a separate client/server application (see SAXESS Trade Client/Server section below) which provides extra functionality. By allowing the client to directly connecting to two such front-ends within the trading platform that are geographically separated, active replication is performed internally by the system. (See figure 4.1) The client after having logged into the system sends orders or requests to the primary front end, which then in turn also routes this information to the secondary front end by way of a back-end conduit called the Request Logger (RL). In other words, it is this back-end application that maintains the replication between the primary and secondary systems. If for any reason the primary system should fail, the secondary system with the identical system data takes over; thus achieving a hardware-based failover.

![SAXESS Architecture Overview](image-url)

**Figure 4.1.** SAXESS Architecture Overview

28
When orders have been matched to other orders the information then passes through the back-end logger module (BL). The BL then logs it to a file and sends the information via the internal binary format SX to the two Dissemination Servers (DS) applications. The first of these two Central Dissemination Server (CDS) remaps this internal protocol into the external XTP protocol (see below) before sending it to Distributed Dissemination Server (DDS). The CDS has the ability to map this internal protocol into several versions of XTP if necessary. The DDS at that point feeds the correct version to connected clients and throws away any versions not needed.

In order to achieve scalability, the system allows any number of front-ends to be connected to the system. This enables more level loading on each of the primary front-ends themselves. The only drawback would be the potential traffic incurred from several front-ends having to communicate with each of the other parallel front-ends. However, this has yet to be a problem since the number of front ends has yet to meet this critical mass.

### 4.3 Exchange Transaction Protocol (XTP)

As with CLICK, SAXESS allows external communications from third party applications. However, SAXESS has separate external client protocols (XTP/XMP) and internal protocols (SX) used in the communications between different modules. The Exchange Transaction Protocol (XTP) is a proprietary, open-source protocol developed by OM in order to provide simple, standardized communications with client applications. As well as external exchange member clients, XTP is also used as a communications protocol between systems administration applications and the trading engine. The protocol is primarily ASCII, but does include some binary data to account for transaction information.

#### 4.3.1 Transaction Flows

XTP has three transactions flows which are analogous to CLICK’s own transaction terminology:

- Interactive (analogous to placing orders and other synchronous functions)
- Unicast (analogous to directed broadcasts)
- Multicast (not to be confused with IP Multicast, but analogous to general broadcasts)

The interactive transaction flow is a synchronous interface that usually returns either an error or an acknowledgement of a request. A request can either be non-updating like queries or updating like placing or changing orders. From this view, the SAXESS front end can only guarantee that the request is going to be handled, but not that that change will be carried out. Only when the information comes
back from the DDS can this update be confirmed. There should also be noted that the interactive mode is not always time sensitive in that a client could potentially get a confirmation of a updating request before the front-end has actually returned an acknowledgement. This situation needs to be covered by the client if such be the case.

The unicast and multicast transactions flows are for the most part asynchronous. (Please note that the multicast transactions have nothing to do with IP Multicast, but instead denote a multiple broadcast, hence the name.) Like CLICK's broadcast mechanism, these flows are used to disseminate information to either specific clients or to all connected clients. The unicast flow is most importantly used in confirming that an updating request either failed or succeeded.

### 4.4 Exchange Message Protocol (XMP)

While XTP is used predominately on the applications level of the protocol stack, XMP is used in the presentation and session layers. In essence, XTP messages are encapsulated with XMP frames. These XMP frames are most commonly sent over a TCP/IP session. This protocol also provides other functionality like connection supervision and session state indication. Compression is only supported from the server to the client and not vice versa. This is due to relatively little network traffic a single client could generate as opposed to the large amounts of data that the server must feed each client.

### 4.5 Exchange Message API (XMAPI)

In order to help developers use XTP and XMP, OM has developed an open source API. This API called the Exchange Message API (XMAPI) was developed in ANSI C and is currently tested on both the Solaris and Windows NT platforms. Sample code and applications are included along with this API as well as documentation for each of these. Most importantly this code gives examples of how to implement compression and encrypted communications as well as the TCP/IP abstraction layer for the two platforms.

### 4.6 SAXESS Trade Client/Server

Although not mandatory for accessibility to the trading platform, OM also markets a client/server solution called SAXESS Trade. This extra system works as a buffer towards the SAXESS trading system and provides extra functionality like storage of historical trades and creating specialized trade reports. Written in C and C++, the server software runs as a service under Microsoft Windows NT. SAXESS Trade also has it's own API which uses the COM framework and allows developers to more quickly develop applications by using the simplified COM API instead of implementing the XTP/XMP framework in its entirety.
4.7 Development

Unlike CLICK which requires a separate development system, a smaller SAXESS system can be test run on a single Sun Solaris user account. System settings can be loaded from a file by way of a State Administrator(SA) module. This allows the system to be run for a testing purposes without the need for a separate database. This can also be utilized to work with specific situations where a bug has been encountered. An individual OM developer can download parts of the source code, easily compile it for running tests, and create external applications or work with new functionality. This trait greatly facilitates developing larger and more complicated client applications. This scalability also allows shorter development times as well as easier modularity.

4.7.1 Change Management

SAXESS also allows for a window of upgrade time between external protocol versions. This is done by allowing the client to determine which version of communications protocol that it would like to use when logging into the Front-end Requester (FR). This allows for several versions of the protocol to be used at any one time, and thus gives exchange members more time to upgrade their clients. Although it should be noted that some changes to the protocol deem it necessary to upgrade to the newer protocol based on newer functionality or some other data that does not fit into the existing XTP framework. This requirement is usually offset by business agreements where the parties have agreed to limit the these upgrades types to typically once a year.

4.7.2 Documentation

Documentation of the XTP protocol is rather extensive. The detailed description of the protocol alone is 824 pages, and even the Programmer’s Guide is 120 pages. Interestingly enough, the documentation is generated from a master XML document that also generates the code for the XTP headers themselves, thus limiting the risk that the documentation differs from the protocol version that has been rolled out. This approach seems to be quite successful and also allows for cross-media publication when necessary.
Chapter 5

The Click OMnet API

The OMnet application programming interface (OMnet API) was developed in order to hide implementation issues encountered in the communications between the different components within the CLICK exchange system as shown in figure 5.1. Basically, the API acts as a protocol, or in layman’s terms, a middle instance for transferring data between two points in a communications system. Besides the external functions normally seen by the client programmer, the API also allows for internal communications between system components and for components used for the software configuration of the exchange[OM 2002a].

In general the API implements the following functionality.

- Communications with the Reliable Transaction Router (RTR) that acts as a middleware,
- Communications with OMdu broadcast mechanism,
- Session issues like login in/out and multi-threading.

The OMnet API is written completely in ANSI C and is used as an underlying component for implementing other languages like C++ and Java. In general the API is rather cross-platform friendly although certain issues have repeatedly caused problems which will be covered in a later discussion. The interface is completely synchronous and must at times wait for the gateway (and thus the server beyond that) to reply to queries or requests for information.

In general, the API has three main areas of use:

**Sending data to the central system in order to carry out transactions** These transactions usually take the form of orders. Orders can then be changed, deleted and so forth.

**Requesting centrally stored information in the form of queries** Queries are generally used when the client first connects to the central system and typically concern opening data like pricing information, which financial instruments are available, and user identification information.
Collecting distributed information in the form of broadcasts Broadcasts are information that is distributed to connected clients. The broadcasts can either be directed to a single user ID (directed events) or to the market in general (general or network events). Broadcasts are also used to maintain a reliable connection with the client by using a so-called heartbeat broadcast. If the client for any reason misses a number of heartbeats then the client is automatically logged out from the central system. A third type of broadcast is also available for calling attention to certain issues (attention broadcasts). These tend to go out to certain user types and not to individual users. The broadcast mechanism is a pull type event service. Broadcasts are stored for each client on the gateway in a buffer and each client must then connect and check for broadcasts.

![CLICK Architecture](image)

**Figure 5.1. CLICK Architecture**

All transaction information used within the system is represented as C data structures or *structs* in several header files. These header files then become the mainstay of a developer's information base about which data needs to be sent into the system and what form it should take when replies are returned. Other constants representing different error values and RTR "facility types" (communication channels) for the CLICK system are also included in these header files. Exchange specific data that may be unique to the system is also contained among these header files.
**OMnet API routines**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>omniapi_create_session</td>
<td>Creates a thread through which you login to the API and is able to use all routines below. The return value (omniapi_session_handle) is used as first parameter to those routines below that are handled by the gateway.</td>
</tr>
<tr>
<td>omniapi_close_session</td>
<td>Closes a previously created session in the OMnet API.</td>
</tr>
<tr>
<td>omniapi_tx_ex</td>
<td>Transfers messages, such as orders, to a central (or back-end) application.</td>
</tr>
<tr>
<td>omniapi_query_ex</td>
<td>Used for requesting information from a central application.</td>
</tr>
<tr>
<td>omniapi_set_event_ex</td>
<td>Enables the reception of broadcast (asynchronous) messages.</td>
</tr>
<tr>
<td>omniapi_clear_event_ex</td>
<td>Disables the reception of broadcast messages.</td>
</tr>
<tr>
<td>omniapi_read_event_ext_ex</td>
<td>Reads broadcast messages. The messages are queued within the OMnet Gateway until they get read. It is also used to get information on available event types.</td>
</tr>
<tr>
<td>omniapi_get_message_ex</td>
<td>Translates a C Status Code to its related text string.</td>
</tr>
<tr>
<td>omniapi_get_info_ex</td>
<td>Retrieves information based on information request type.</td>
</tr>
<tr>
<td>omniapi_cvt_string</td>
<td>Converts an 8-bit character string from or to central format. The central OMnet applications, such as the Marketplace, use the ISO Latin-1 character set.</td>
</tr>
<tr>
<td>omniapi_cvt_int</td>
<td>Swaps the byte order of an integer, if the local processor has a different byte enumeration, big endian, from the little endian. (Intel processors use the same byte order as Alpha AXP, whereas SUN and HP processors use big endian byte order.) This routine is not available in OMnet API on little endian computers.</td>
</tr>
</tbody>
</table>

**Table 5.1. OMnet API Routines**

### 5.1 Interface Design

The API in itself is a library of about 18-20 single-session and multi-session functions as listed in table 5.1. With the single-session functions, there is no need to pass a session handle while with the multi-session functions, a session handle must be initiated and passed onto the function. Multi-session functions have about the same names as their single session counterparts but are followed by _ex. Although multiple session are supported, they are strict synchronous threads and require separate login identifications. Multi-session functions are often used to monitor marketplace broadcasts in order to leave separate threads available to work with order entry and more time critical user input. These functions implement the lower-level programming needed to communicate with central system via the gateway and other APIs like the Reliable Transaction Router API and the OM Directed Information API (OMdu).

Function parameters usually consist of a session handle (for multi-session functions), a reference to a query buffer that includes a transaction header, a facility
type, and references to buffers along with the buffer size for receiving data from the central system. Functions return a single integer value that represents an completion status i.e. success or failure. These error codes can then be looked up and converted into string form by the central exchange system by using omniapi_get_message_ex.

### 5.1.1 Transaction Headers

One of the central structures used for determining which type of function is intended to be carried out by the central system is the transaction header. The transaction header struct called *transaction_type* consists of two characters and one integer. The last number also indicates whether the transaction is for external (0-255) and internal (255 +) use. External datastructures can be loosely translated into internal ones by adding 255 to transaction number.

**Central Module & Broadcast**  
M = Marketplace, C = Clearing, I = Information, S = Settlement, D = Common Database, R = Risk, U = Supervision, B = Broadcast

**Server Type**  
O = Order, Q = Query, D = Deal, C = Command, I = Information

**Transaction Type Number**  
the transaction/broadcast number that specifies action to be carried out

**Example**  
When an order is sent into the marketplace the transaction type will be MO1 (for Marketplace Order) and the transaction header is created with the following code:

```c
order_trans_t trans;

trans.transaction_type.central_module_c = 'M';
trans.transaction_type.server_type_c = 'O';
trans.transaction_type.transaction_number_n = 1;
```

### 5.1.2 Reliable Transaction Router

In order to interface with the RTR middleware, the client needs to specify which "facility" or RTR channel is responsible for that type of transaction. Facility types are the same numeric constant from system to system; however, in the *omnifact.h* header file string literals for these values. Some of the common facility literals are OMNI_FACTYP_LOGOUT, OMNI_FACTYP_LOGIN and OMNI_FACTYP_SET_PASSW which handle user administration and EPx (External Production facility with index x) which is used for many of the other transactions used.

### 5.1.3 Tunneling and Compression

Since data can potentially be sent over insecure network connections, the CLICK trading platform uses a tunneling client/server application for communications between
<table>
<thead>
<tr>
<th>Data type</th>
<th>Bit size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8</td>
</tr>
<tr>
<td>Word</td>
<td>16</td>
</tr>
<tr>
<td>Longword and int</td>
<td>32</td>
</tr>
<tr>
<td>Quadword and long</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 5.2. OM C Data Type Redefinitions

the central system and the gateway usually physically situated at the member’s site. Broadcasts are compressed and then distributed to other parallel network gateways via the OM Distribution Utility (OMdu) using User Datagram Protocol (UDP), a well-known connectionless protocol. UDP packets are then converted into TCP/IP packets for transport over a wide area lan (WAN) and recovered into UDP for local distribution once they reach a local site. Encryption of all traffic is also implemented primarily in the OMnet API. As a result, developers need not worry about unencrypting information or reencrypting their outgoing traffic. However, the developer should be aware of which data structures are compressed and which are not in order to best work with the available network capacity.

5.1.4 Structs

The central idea for the API is implementing data communications between the main system and connecting clients. This makes data structures quite central to the API. There are just under 700 unique data structures in the form of structs which contain just over 1000 unique data variables. These structs are contained in the omex.h file which can be exchange specific.

5.1.5 C Data Types

As mentioned earlier, the ANSI C standard does not specifically define the size of integers to such an extent as necessary (see earlier discussion). As a result, when reading directly from buffered data, one compiler can define an integer as another size compared to another. In order to remedy this problem, the OMnet API has redefined all generic data types as listen in table 5.2.

5.1.6 Batch Queries

Batch queries are also a common occurrence while using the API. When more data is to be sent to the client than can be easily accommodated, the system segmentates the data into several responses and indicates this using a segment number. Zero indicates that there is no more data available while 1 indicates more information. Upon detecting this, the called function needs to be re-called until the segment number is again a zero value. In this manner, the entire data transaction is marshalled over to the client application without it dominating network traffic.
5.1.7 Broadcasts

Broadcasts are used primarily when information within the marketplace has changed. This information needs to be distributed to individual clients in a time critical manner. To do this, data is written to a broadcast buffer on the gateway where it waits for the client to retrieve it. The client after having indicated that it wishes to subscribe a certain broadcast calls the omniapi_read_event_ext function where the receiving buffer is filled with the data from the broadcast buffer. The size of the buffer received is then compared to the size of known broadcast structs available within the header files to determine which broadcast format is being received. Then the received buffer is matched up with the targeted broadcast and the information is utilized.

<table>
<thead>
<tr>
<th>Broadcast type XYNN</th>
<th>Items (4)</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Named struct (A)</td>
<td>Size</td>
<td>Data</td>
</tr>
<tr>
<td>Named struct (B)</td>
<td>Size</td>
<td>Data</td>
</tr>
<tr>
<td>Named struct (C)</td>
<td>Size</td>
<td>Data</td>
</tr>
<tr>
<td>Named struct (D)</td>
<td>Size</td>
<td>Data</td>
</tr>
</tbody>
</table>

**Figure 5.2.** Variable Information Message Format

Since broadcasts represent a considerable amount of traffic coming out from the system, a newer type of broadcast has been implemented. Variable Information
BO1  Orderbook Changes with Identity
BD1  Deals in Market
BD2  Edited Price
BI1  Resumption / Suspension of Trading
BI2  Market Opened / Closed
BI5  Index Information
BI8  General Message
BI9  Price Information Heartbeat
MI1  Quote Request Info
BI22  Dedicated Market Maker Alarm

Table 5.3. OMnet Broadcast Types

Messages (VIM) are message constructed with variable size and content (shown in figure 5.2). When the program receives a broadcast, it is compared to all known broadcast sizes. If nothing matches up with the size, the only available broadcast is that of the VIM also known as broadcast BO14. Within the header of the VIM, a field called items displays the number of broadcasts included as well as the size of the entire construct. The code, after reading this information then encounters the first broadcast which has a header that includes its name and size. If the code recognizes this name and size, it can then process the broadcast, but if it is unaware or unable to process this information, then it is expected to skip it. In this manner, new broadcasts can be quickly introduced into the system without the developer having to constantly change or upgrade existing code.

It should be noted that broadcast information is compressed when it is sent through the system in order to limit the use of network bandwidth. In contrast, query responses normally engage once a client has logged in are not compressed.

5.2 Development

Development is done predominately by using a test system made available for developers. The test system is usually filled with data that would represent a normal exchanges financial products. It is then up to the developer to fill it with different orders and event situations in order to experiment with output from the system. During the writing of this report, a small application allowing the developer to examine completed transactions from the gateway was developed. At this time, there has been no feedback concerning its usability.
5.2.1 Documentation

Documentation for the external OMnet interface consists of three static documents in the Adobe’s portable document format (PDF) format presumably converted directly from Microsoft Word.

- *The OMnet Application Programmer's Interface Manual* describes how to use the API as well as some information about functionality and system design.

- *The OM System Programmer's Manual* describes transactions, queries, and broadcasts used in the OMnet API, the C structures sent as OM messages, their fields and descriptions.

- *The OM System's Error Messages* lists the error codes returned from the system.

These documents along with some sample applications are the basic information that a developer would receive from an exchange for which a specific client would be developed.

5.2.2 OM Courses

The developer also has a chance to take a course focusing on the API provided by OM to developers. The course covers developing with the API on the Windows NT platform. It gives the developer hands-on first-hand contact with the API on the NT platform and is taught by someone from within the education department or a programmer who is actively working with further development. Other courses of interest might be several other courses covering OM's own client CLICK Trade in order to work with the actual functionality provided by the system as well as one covering *Transactions and Broadcasts*. This course covers how events travel through the CLICK exchange from component to component. A combination of these courses should give the developer a basic view of how to start developing with the API and allow them to sufficiently use the CLICK Trade client on a developmental or educational system.
Chapter 6

Interface Standards

6.1 Straight-Through Processing (STP)

As exchanges and financial institutions have started computerizing almost all of the services they provide in order to cut costs and administration difficulties, the new concept of straight-through processing (STP) has taken form. STP, in short, is a goal of providing automated end-to-end trading from a customer order all the way through clearance and billing. The ultimate goal is to limit the costs involved in manual processing of transactions and meet more stringent clearing rules. One way that has been laid out has been the introduction of standards in financial applications. The idea being that if all systems can communicate using a pre-defined standard, then interoperability will be achieved independent of which software vendor or technology is being used. As a result, many organizations have been formed by industry actors with the goal of creating new standards in financial protocols using various technologies.

6.2 Messaging Protocols

Since most if not all exchange engines are proprietary in nature, most sharing between systems is done on the data level instead of further back in the system towards middleware or otherwise. With this in mind, messaging between systems has become extremely important in order to meet any sort of STP criteria. Standards within the financial arena have been trying to come to terms with how to best represent this data. XML, text based proprietary formats, and binary API are some of the solutions that have been used successfully in recent years.

6.2.1 Non-XML Protocols

One of the types of standards used in the industry is based on text formats. Some are proprietary in nature, but are usually well documented and rather straightforward. The text nature of these formats also make them easy for programmers to adapt existing systems. Usually other middleware is involved behind an external gateway
that then converts from some internal format to an external format. These formats are popular and used quite frequently within the financial sector.

**Society for Worldwide Inter-Financial Telecommunications (SWIFT)**

Society for Worldwide Inter-Financial Telecommunications (SWIFT) defines a proprietary standard format for sending and receiving financial messages from one financial institution to another over a private network. This messaging standards consists of a collection of field names and value pairs (see example in table 6.1). Messages fall into *series* categories which are named after which hundred they fall into for example the 100 series, 200 series, etc (see table 6.2). Besides the actual messaging standard itself, SWIFT also offers value added services like a central network as well as software. SWIFT also requires that each customer have a dedicated terminal with pre-accredited software.

**Financial Information Exchange Protocol (FIX)**

Much in the same way as SWIFT, the Financial Information Exchange protocol is also a simple text based protocol; however, the FIX standard is open and non-proprietary. As a result, it is being widely used by the financial sector as a means of data transfer. In a 2001 survey, 31% of the institutions already used FIX within their existing systems and 82% of the brokers surveyed [Atwell 2001]. One of the major differences between FIX and other financial protocols is that FIX is a connected, session-based protocol. FIX realized that the tight coupling between the Session and
Application Layers was resulting in a less flexible protocol and thus split the two levels. FIX consists of two layers: the Session Layer and the Application Layer. The Session layer deals with all session information like establishing connections and at message delivery including data integrity, sequencing and the like while the Application Layer deals with business related content like that of orders and execution. FIX is flexible and allows even user defined tags within the standard. SWIFT, in an effort to bring together financial standards has plans for incorporating FIX into their own services. Like SWIFT, FIX is also tag based with a key-value pair syntax (<TAG>=<VALUE><DELIMITER> CSV). FIX can be carried in any messaging protocol in the message body, and thus extends its flexibility and interoperability potential.
6.2.2 Extensible Markup Language (XML) Protocols

One of the more recent trends in standardization has been the use of Extensible Markup Language (XML). Instead of creating an internal proprietary standard which is difficult to maintain as well as parsers to pull out the data, XML offers a flexible manner of storing data which also conveys content through self-content. XML offers a simple lightweight meta-language for data exchange unlike other electronic data interchange formats. Furthermore, XML is neither product or platform specific and has become a mature technology having stemmed from Standard Generalized Markup Language (SGML) which had its first published draft in the early 1980’s[SGML 1990]. Today there are freely available parsers and APIs for almost every programming language available. Even off-the-shelf software is beginning to fully support XML documents. As a result, several organizations have been formed in order to create financial protocols that use the XML format.

At the writing of this report, several XML based financial protocols are starting to be used within the industry. Financial Information Exchange Markup Language (FIXML), Financial products Markup Language (FpML), Market Data Definition Language (MDDL), and eXtensible Business Reporting Language (XBRL) just to name a few. All of the specifications for these protocols have a controlled vocabulary determined by the standards organization themselves. The major difference between the various flavors is usually encountered when examining their origins. Some of the XML protocols come from established non-XML protocols like FIXML while others have begun their inception at the XML level.

```
<ResearchReport type="equity">
  <Title>KRT: 3Q well below est’s; FF0 should… </Title>
  <Analyst>John Smith</Analyst>
  <Date>06/05/2000</Date>
  <Company>Kranzco Realty Trust </Company>
  <Industry>Real Estate Investment Trust </Industry>
  <CurrentRank>3H</CurrentRank>
  <Yield>15.6</Yield>
</ResearchReport>
```

- Financial Data XML Example

6.3 Discussion on XML Protocols

As XML starts to emerge as a standard of choice for intercommunication between systems, it does have its advantages and disadvantages. On the plus side, XML allows programmers to focus on data content instead of format. Instead of having to develop tools to create these format and parse them, other projects like Xerces (http://xml.apache.org/) have already created efficient validators, parsers, and XML writer libraries that are well documented and used by a large internet community as well as a large number of companies. Instead developers can focus on other issues
like data processing and data-cleaning. As a plus, XML is also a text format which allows for the developer to easily display the information being sent back and forth.

As XML is maturing, it's becoming a very popular format for complex web communications and even database queries. This has furthered the argument that XML could be *lingua franca* for connecting communications formats within systems and to other external systems. Instead of vendor APIs or specialized connectors, data integration would be nothing more than pulling the information out an XML format and reforming it to meet the needs of the specific system or standard.

XML provides easy validation of data through the use of Document Type Definitions (DTDs). Other protocols can provide this service only by either buying software vendors validation tools or by internally developing one. Using DTDs as a simple data validation does have it limits. DTDs only validates types of data and whether they are obligatory or not and not the quality of the data involved. This limits XML's use as a data validation tool.

On the down side, XML is a "slower" technology when it comes to data transfer which ultimately could be its downfall. Creating XML files from raw data and pulling or parsing that data from raw XML files, takes quite a bit of processing power, and since the data is in a text format, there are rarely any short cuts. Sending this data then padded by XML content such as opening and closing tags tends to bloat communications by several times the size of the original data even if compared to that data being transferred in a raw non-XML text format. As a result, high-performance applications will avoid this technology completely and tend to lean towards more raw binary formats. Since exchanges are looking to expand at "lightening speed" this solution might scale well for users with limited communications demand on a system, but will definitely not meet the needs of more avid software clients. The timeframe between a binary communication compared to an XML one, would give a clear advantage to the binary one.

### 6.4 Discussion on Interface Standards

Standards within interfaces have several advantages to the developer and to the development process. Since standards are usually created by developers within the specific industry, their combined experience finds its way into the standards. Newer developers are then not reinventing the wheel but instead are "standing on the shoulders of giants". This leads to shorter development times as well as better documentation, as geographically distant developers then contact each other to discuss implementation issues. Standards also tend to implement a degree of abstraction in order to give it a degree of flexibility in order to meet the developing communities needs. Developers are also given a clear development direction from the standards body and most questions can be answered by standards specifications.

On the down side of using an standard interface, is that because of the generalization these implementation may perform at less than optimal speed if instead a proprietary interface were used. As different version of the standard become available,
change management becomes as development issue. Questions regarding backwards compatibility and version handling then become a problem if not handled by the standards development community in a ingenuitive manner. Not to mention what if the standard implemented happens to be proprietary in nature, what ensures that the developing organization is stable enough to continue further development.

These issues tend not to be an issue with internally developed standards. Instead, interfaces are then custom tailored to fit system’s needs although at a higher development cost. Internal competence is also bolstered by creating proprietary interface to the system and subsequent software can be developed and sold to external actors within the industry. If the interface is then successful enough then it in and of itself becomes a de-facto standard for the system. Thus standardization ensues through system and developer dominance.
Chapter 7

Click API Discussion

Imagine, hypothetically, just for a moment, that programmers are humans. No, don’t laugh, I’m serious. I know that all evidence is to the contrary: they work at night and sleep during the day, and only need 2 hours of sleep, they have a diet of pure sugar and caffeine, they can use, and actually enjoy using the vi editor, but despite all this, just for the sake of the argument, imagine that they are human.

Now suppose for a moment, also for the sake of the argument, that their chief method of communicating and interacting with computers was with programming languages.

What would we, as HCI (Human Computer Interaction) people then do?

Run screaming in the other direction, I hear you think.

- Steven Pemberton "Programmers are Humans Too, 2"

This chapter discusses the external CLICK API from a developer's point-of-view. API design is also discussed in this context.

7.1 Introduction

API programming focuses mainly on simplifying and breaking down problems regarding common implementation concerns for application developers. By giving them a basic building block structure, developers can then work on creating high-level functionality that specifically meets the needs of their group of end-users. Within any industry, these groups of end-users may have divergent needs, and as a result, any API should meet some sort of middle ground. Choosing the appropriate abstraction level becomes very important for developers using the API in order to make it more understandable and easier to learn. If this API is well designed, developers will be able to quickly understand its workings and develop applications within a short time-to-market. As the API moves from development into more stable use, a larger amount of developers can then begin to use this interface. If it
then becomes used predominately within the industry over a long period of time, then the API itself can become a standard. Developers would then have a better chance of coming in contact with the API. These design issues should also include other aspects like documentation, education, and testing tools. It should deal with mundane tasks and with potential problems commonly encountered by the system developers. With this in mind, the API must also allow for change management as newer versions are developed.

At the moment, the CLICK API is a rather stable, consciously designed API that has been used by numerous external developers. Although there is room for improvement, the API in comparison to other industry APIs fares rather well. The CLICK trading platform is used in several securities exchanges and has 30 approved independant software vendors that currently support the API. Since the system has also achieved longevity, it is well-known within the industry.

7.2 General Programming

7.2.1 Binary Data Structures

One of the areas where coding errors easily can occur, lies in the predominant use of C binary structures by the API. Since the total number of structures ranges just over one thousand, a developer can easily make a mistake as to which type of data and its correct length is used within the C structure. This in turn creates an error from the gateway and the transaction is aborted or unrecognized. As a result, the developer works almost exclusively with the raw binary data which is passed back and forth to the gateway. Since this binary data is unreadable by the naked eye, this poses a problem for developers trying to debug code and locate potential errors. Debugging is also compounded by the need to artificially impose the length of integer data in order to correctly transfer the structures in a uniform size as well as endian considerations.

This type of problem commonly occurs when low-level implementations of networking programming has been chosen. While the CLICK API does not require the application developer to work with low-level sockets implementations, the abstraction level is not far above this level.

Several approaches can be considered to work with this problem. The following section will attempt to suggest different levels of change.

XTP/XMP Interface

Perhaps one of the most radical and time consuming suggestions would be to integrate the CLICK API with the existing XTP framework. XTP has proven to be an effective means of communication on the SAXESS exchange platform, and thus should not hinder this high performance system. At the moment, OM has separate external interfaces for the CLICK and SAXESS platforms. Neither speak each other’s language interface nor can communicate in any reasonable manner other
than the implementations of FIX that have been recently developed to comply with
customer needs. If instead, CLICK were to try to implement an XTP/XMP inter-
face, developers internally and externally could focus efforts on the one front instead
of two.

XTP's real strength is a more human friendly protocol considering that the
bulk of it is in simple text. This gives programmers an easier time of reading
that data is being retrieved and sent between the systems. SAXESS also has an
implementation of sending out multiple versions of XTP in order to meet change
management needs. If this module idea could be adapted to CLICK then upgrading
between versions would be a much easier process for end-users. In this manner
the external and internal API is also separated giving greater freedom for internal
changes. Admittedly this conversion between the two formats would take some
processing time, but would most probably not be a significant load upon the system.
Development time for this kind of change would also take a longer time than other
potential measures, but should focus programming development.

Tracing Functionality

A less radical solution, but all the more difficult to effectively develop would be to
create an application that traces client data allowing the developer to read in clear
text the data sent to the gateway from a specific client by listening to each of these
ports.

Since the API hides implementation issues, external developers have a potentially
difficult time determining where errors occur within the exchange system when send-
ing transactions. Since there is, at this time, no manner to trace from the client
to the gateway and on for an external developer, most developers are programming
"blind". Developers usually have to guess how far and for what reason their trans-
action failed. At the time of the writing of this report, a new developer's tool was
designed to allow developers to see transactions that successfully navigated through
the gateway. However, even failed transactions, and internal messaging regarding
them would be of greater use to the developer.

In an attempt to provide developer tools to work with client communications,
OM has recently developed a tool to allow developers to view data in human readable
text format of the transactions that have successfully reached the gateway. This idea
should be a large step forward for detecting problems with data sent to the gateway.
However, the application should go a step further and try to match data that comes
from non-successful transactions.

Corba Middleware

Again, a radical suggestion would be to look at replacing the RTR middleware,
could be in taking a look at Corba technology. Corba has proven itself a mature
technology and is currently being used within several high performance areas like
that of medical imaging and data transfer over Gigabit networks. Even other issues
like that of failover and replication are built into many of the Corba implementions
available today. This subject in and of itself could easily be an entire Master’s Thesis, but the main programming issues that a Corba implementation would solve would be of great use to an external programmer.

The real plus side of Corba is its uniform Interface Definition Language (IDL). This definition language has precise definition for how methods are used and types are defined among other things. These definitions allow other programming languages to interface with a common Corba middleware while skirting problems like endianess and bit lengths for integer types. Furthermore, the vast number of languages that now have IDL mappings makes this technology virtually platform and language independant. So even an implementation of a C++ objectification of the interface would not suffer the overhead of a java virtual machine that is often associated with Corba. Furthermore, since Corba uses mainly an object-oriented approach, the developer could easily work with objects instead of structures and have implemented memory considerations.

### 7.2.2 C++ Wrappers and Object-Oriented Design

An approach to lowering the learning threshold encountered when working with the API, is to objectify the C code by implementing C++ wrappers around it. In this manner a more appropriate level of abstraction would be achieved in order to give the developer a better model with which to work.

Since the API is written in C, providing a more object-oriented approach in C++, shouldn't impede on performance issues. One developer in particular is already using this C++ wrapper approach and finds that development times are reduced drastically by using C++ wrappers around the OMnet API. As a result, version changes and implementation details rarely cause problems, and when they do error handling is much easier as a result[Derki2003]. Another advantage of using C++ wrappers is the ability to objectify transactions and perhaps make the exchange system more easily understandable for the developers.

This approach also allows for common application implementation to be covered without the developer having to implement them themselves. Added features like iterators would also allow developers to easily work with large sequences of data. This solution might be more appropriate as an add-on, open source, addition to the samples provided by OM. The same developer using the C++ wrappers also mentioned that most development done within their firm was not C, but other more object-oriented languages like C++ and to a limited extent Java.

At the writing of this report, C++ wrappers were being developed in order to create code for which the Java language could then access the OMnet API. These wrappers did not objectify the entire transaction system, but merely allow for stubs to be generated so that Java could then manipulate the objects returned. If the entire transactions system could be remodeled, objects could be created to resemble broadcast and transactions much like Java OMnet components.
7.2.3 Limit Thread Synchronicity

As mentioned earlier, each thread within the OMNet API is completely synchronous, and a common practice is to shuffle operations that are gateway response sensitive to separate threads. As a result, the developer tries to develope a pseudo asynchronous behavior from the gateway. This behavior could be changed to work more like that of the SASS front end. SASS’s front end module, FR, solves the synchronicity problem by returning a timestamp to the client (assuming basic correct syntax) to confirm that the module has transacted something from the client. Only through a later message from the Distributed Dissemination Server (DDS) that identifies itself with that timestamp that the client is informed if the transaction has been successfully executed and more information concerning order identifications and the like. This behavior limits the time needed to synchronize thread activity and cuts out potential lags in waiting for orders to be carried to market. Implantation could be done by using the existing directed broadcast mechanism and merely adding a timestamp functionality to the gateway that could be combined with the logged in user identification in order to achieve uniqueness.

7.2.4 Consequent Formats within API Structs

As the OMnet API has grown from version to version, standardization of formats within the omex.h header has become a difficulty. More specifically, dates and time formats have become inconsistent. No standard way of handling these formats has been addressed. This issued coupled with the big-endian problem mentioned earlier, becomes an issue for cross-platform developers. As a result, a review of the structures within the API should be considered. Issues like minimalizing the number of structures and standardizing similar formats (i.e. dates and times) should be addressed.

Admittedly, date and time formats can be of differing precision; however, by giving the developer several different formats for the exact same type of data is unnecessary and violates the central idea that an API should take care of common tasks and formats.

The solution to this kind of problem is very easily fixed. By simply defining how date and date/time formats should be expressed to the API and using it as a standard, these kinds of implementation problems could easily be bypassed. For example, as data representation plays such a central role in this type of system, an

Table 7.1. Examples of OMnet Date Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>general date format</td>
</tr>
<tr>
<td>int64_t</td>
<td>date_and_time</td>
</tr>
<tr>
<td>uint16_t</td>
<td>expiration_date_n</td>
</tr>
<tr>
<td>uint8_t</td>
<td>query_on_date_c</td>
</tr>
</tbody>
</table>
Time representation formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>several time formats like sent_time_s,</td>
</tr>
<tr>
<td></td>
<td>timestamp_time_s and modified_time_s</td>
</tr>
<tr>
<td></td>
<td>to name a few</td>
</tr>
<tr>
<td>int64_t</td>
<td>timestamp_comp_s and timestamp_dist_s</td>
</tr>
<tr>
<td></td>
<td>date_and_time and timestamp_in_q</td>
</tr>
</tbody>
</table>

Table 7.2. Examples of OMnet Time Formats

internal standardization of data would be of great benefit for the developer. Simply representing the date/time stamp in the same way as the Network Time Protocol (NTP) would help developers internal and external work with this format in a more unified manner. The NTP time format would even allow further development into date time synchronization between clients and servers if this ever became necessary.

7.2.5 Broadcasts

Push Model

Broadcasts in the CLICK system are pushed onto the gateway from the central system and stored in a memory buffer for the clients to periodically check, read, and download. This type of pull event handling from the gateway, as mentioned earlier, creates more network traffic when clients have to constantly check with the gateway for broadcasts. Instead, if the gateway could push this information out to the clients, this extra traffic would be eliminated. This would also allow for greater data flow to be pushed out to clients since clients would receive the information as soon as it becomes available instead of a fast moving stream of data filling the gateway's memory buffer and the central system writing over it. Clients instead would have to create listening threads and reply success or failure to the gateway. An IP multicast implementation for popular broadcast subscriptions could also be used here in order to limit IP connections with the gateway, however; confirmation that the client has received the data would have to be sacrificed.

Grouped User Identifications

As mentioned previously, multi-threading is a common practice used for separating time critical functionality from broadcast retrieval. As a result, each thread requires its own user identification and thus incurs licensing costs from the exchange for each thread. The developmental reasoning behind allowed exchanges to easily track the behavior in an easy and secure manner. For the developer, however, this also meant working within these boundaries and trying to limit synchronicity. By using separate threads when processing time critical transactions that usually wait longer times for responses (i.e. order placement), the client uses other threads to monitor other critical data (i.e. options price change broadcasts). These separate threads are used
to monitor broadcast because developers generally want to leave the order entry thread open for immediate use. As a result, due to the synchronicous nature of the functions, the developer must internally implement an asynchronous behavior. This design aspect seems to be common among end-user clients.

Because the preferred design mechanism is a separate thread for broadcast monitoring, directed broadcasts create challenges for developers. Only the thread/user identification that entered an order transaction can get back directed information about whether it has been matched within the central system later when a broadcast is returned to it. However, it is usually this thread that the developer wishes to avoid unnecessary blocked connections to the gateway, and as a result, using this thread to constantly check for broadcasts is not desirable. A possible solution might be to modify user rights within the Central Database (CDB) and allow user multiples to be created in the event of directed broadcasts. The gateway could maintain information about which user threads wish to redirect their broadcasts to another friendly thread. Admittedly the idea behind the user login is to enable thread control and guarantee confidentiality, but since clients software is already using workarounds in this area, it might be an idea to look further into this area.

7.3 Developer Tools

One of the main areas where the CLICK API has been more challenged is in the number and quality of tools given to the developer. While the OM courses provided are quite good for a developer beginning with the API, more indepth information and development tools are lacking. Although this might not be considered strictly as a part of the programming API, a well-managed API will provide tools to the developer in order to lower learning thresholds and facilitate further development.

7.3.1 XML Documentation

While XML may not lend itself to high performance communication applications, it does work wonderfully for documenting applications. XML is in itself flexible enough to allow manipulation, searching, and publishing of data in several formats. The SAXESS project uses XML predominantly in order to maintain it documentation. SAXESS even goes so far as to generate its programming code from the XML itself and not the other way around. This has proved quite successful in maintaining the large amount of documentation used for XTP/XMP development. This too might be a potential idea for CLICK documentation. By converting reference literature into XML as well as header information, cross-linking and searching would instantly become available by using other XML standards like XLINK and XPATH. From this XML data, documentation much like javadoc can also be provided to developers. Even cross linking with other documentation like the OMnet Application Programmer's Interface Manual's examples with pertinent transactions listen in the OM System Programmer's Manual can be of great use to developers and facilitate
future documentation. By going one step further and cross linking this information with *OM System’s Error Messages* would even help developers with debugging information for specific transactions.

### 7.3.2 Annotated Documentation

Another aspect of creating a successful API is creating the sense of a virtual community for developers. Although most developers using the CLICK API will be from competing firms, most of the developers will have experienced the same learning curve and each hold information concerning quirks that might be helpful for further development. One way to build this community and at the same time receive documentation help is by using the concept of an annotated documentation system. An annotated manual invites developers to "contribute examples, caveats, and further clarifications" in order to give feedback and help to other application developers. These notes (see figure 7.1) are then used to update the next version of documentation and address development issues otherwise commonly answered by a support organization. Error corrections can also be made available in this format and bug information may also be posted by OM developers. As a result, the loose concept of community can be artificially created in a manner that suits many developers. Besides extra documentation, internal OM developers can also infer from this immediate feedback regarding the direction of client development and cover potential implementation issues.

<table>
<thead>
<tr>
<th>rainerd at elwood dot com</th>
<th>24-Dec-2002 01:01</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Any two-place operator can be used in this operator-assignment mode.&quot;</td>
<td></td>
</tr>
</tbody>
</table>

It should be mentioned that this does not apply to the comparison operators. $a <= $b is obviously not the same as $a = $a < $b.

<table>
<thead>
<tr>
<th>oliver at handeln-online dot de</th>
<th>07-Aug-2002 06:06</th>
</tr>
</thead>
<tbody>
<tr>
<td>The short-circuiting is a feature. It is also available in C, so I suppose the developers won’t remove it in future PHP versions.</td>
<td></td>
</tr>
</tbody>
</table>

It is rather nice to write:

```
$file=fopen("foo","r") or die("Error!");
```

Greets,
Oliver

<table>
<thead>
<tr>
<th>php at cotest dot com</th>
<th>17-Jul-2002 11:08</th>
</tr>
</thead>
<tbody>
<tr>
<td>It should probably be mentioned that the short-circuiting of expressions (mentioned in some of the comments above) is often called &quot;lazy evaluation&quot; (in case someone else searches for the term &quot;lazy&quot; on this page and comes up empty!).</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.1.** Example of Annotation from the PHP Manual

54
7.3.3 API Test Suite and Simulator

One of the real strengths with the SAXESS platform is the ability of the developer to run a standalone version of the back-end in order to create and test new functionality. This also allows for easier development of external clients through the XTP/XMP interface. In contrast, CLICK lacks this free-standing test option and instead the developer must test communications on a live test system that may be simultaneously used by several other developers. However, by providing an easily installable simulator of the CLICK system, developers would have the ability to test clients under development and work with new functionality without disturbing other developers work on educational or test systems. This would also give developers a better understanding of expected thread response times.

7.3.4 Better Granularity of Error Handling and Documentation

As mentioned previously, most of the CLICK API functions return, what is referred to as, a completion status. This status denotes whether or not the transaction has been completed successfully or not. These completion status codes are described in The OM System’s Error Messages documentation. Generally, the first thing a developer’s code does after having received an error is to request the text message for the specific error incurred in order to give the user/developer a better understanding of the error situation. This message is usually where the developer starts looking for clues about what could have gone wrong. Unfortunately, this information is rather short in nature and quite vague. In fact, two different errors might have the exact same message text. Upon further inspection the developer might then turn to the error documentation. Here, only a very basic description of the problem encountered is described. Instead, it is usually up to asking other developers concerning their own experiences when encountering that specific error and hoping to start from there. This would be an appropriate area to allow for annotated text that could point out several possible causes of the problem and recommend work a-rounds and debugging protocols. But in general, more content is needed to explain possible problem areas and how they differ from other errors.
Chapter 8

Conclusions

As viewed by the sum of the entire API, the design criteria for OMnet meets the basic needs for developing client applications for the CLICK exchange system. However, there is room for improvement with its documentation. By re-evaluating how the existing documentation is provided to developers and minimizing errors, there would be great steps in minimizing implementation problems. Further work into creating cross-linked XML documentation from the existing documentation as well as API code would be of great advantage for developers. This could then lay the ground work for an annotated documentation system that allows developers to comment and make suggestions for documentation and hence create a virtual development community.

Systemwise, implementing a Corba middleware solution should be further investigated as a possible avenue for further development. Broadcast mechanisms should be reconsidered regarding the basic pull model and directed broadcasts. Allowing redirected direct broadcasts should be made available in the gateway or Central Database (CDB). Synchronicity issues concerning threads should be limited in order to cut down the time a thread must wait for an answer.

In addition, more development tools should be implemented like that of open source object-orientated C++ wrappers as well as tracing functionality for gateway transactions. A simulation or simple version of the Click system would also give developers the ability to work with an individual version of the system without having to work on a central test system.
References


59
http://www.theifm.org/tutorial/contracts3.htm

[Investopedia] *Electronic Trading: Electronic Communications Networks (ECNs).*
http://www.investopedia.com/university/electronictrading/trading5.asp;

[Investopedia.com] *Electronic Trading.*
http://www.investopedia.com/university/electronictrading/default.asp


http://www.londonstockexchange.com/about/about_05.asp


http://www.iseoptions.com

61


http://www.optionetics.com/articles/article_full.asp?idNo=8102

http://www.sfomag.com/3decades/index.htm


http://classics.mit.edu/Aristotle/politics.mb.txt

http://www.optionetics.com/articles/search/article\_full.asp?idNo=7411

http://www.wallstreetandtech.com/printableArticle?doc_id=WST20010816S0008


http://www4.ncsu.edu/~rsware/bus422/history.pdf
Appendix A

Appendix

A.1 List of Financial Terms

**American-style Option** An option that can be exercised at any time between the purchase date and the expiration. This style of option usually has a higher premium than the European-style option. See also European-style option.

**Auction or "Out-cry" Market** An exchange where "specialists" are the mechanism which "moves the market". See also Specialists.

**Call Option** A contract that gives the holder the right to buy a certain amount of an underlying security from the writer of the option at a specified price up to (American-style) or at (European-style) a specific date.

**European-style Option** An option which can only be exercised at the date of expiration.

**Exchange** A trading area for financial instruments to be bought and sold

**Market Maker** An actor in the mostly associated with an electronic or European style of Exchange that is standing by ready to buy and sell at publicly quoted prices. MM’s traditionally are offered lower trading costs from the exchange in order to be compensated for the spread they must maintain in the market.

**Member** A member firm is a company, individual or group of firms that either owns a "seat" like that for more traditional auction exchanges or is a company that has met certain regulatory criteria. Members traditionally have an exclusive right to trade on the exchange.

**Moving the market** Buying and selling within the exchange

**Specialist** An exchange member who maintains an inventory of certain securities of which that member has expert knowledge. The specialist is then responsible to see other buyers and sellers have the opportunity to buy and sell within certain bounds.