VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MODULAR MULTIPLE LIQUIDITY SOURCE PRICE STREAMS AGGREGATOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ ROZSNYÓ

BRNO 2012

MODULÁRNÍ AGREGÁTOR CENOVÝCH ZDROJŮ POSKYTOVATELŮ LIKVIDITY

MODULAR MULTIPLE LIQUIDITY SOURCE PRICE STREAMS AGGREGATOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ ROZSNYÓ
AUTHOR

VEDOUCÍ PRÁCE doc. RNDr. JITKA KRESLÍKOVÁ, CSc.
SUPERVISOR

BRNO 2012

Abstrakt

Abstract

This Master Project provides a theoretical background for understanding financial market principles. It focuses on foreign exchange market, where it gives a description of fundamentals and price analysis. Further, it covers principles of high-frequency trading including strategy, development and cost. FIX protocol is the financial market communication protocol and is discussed in detail. The core part of Master Project are sorting algorithms, these are covered on theoretical and practical level. Aggregator design includes implementation environment, specification and individual parts of aggregator application represented as objects. Implementation overview can be found in last Chapter.

Klíčová slova

Keywords

financial market, foreign exchange, FX, FOREX, FIX protocol, high-frequency trading, price aggregator, sorting algorithms, QuickFIX/n

Citace

Tomáš Rozsnyó: Modular Multiple Liquidity Source Price Streams Aggregator, diplomová práce, Brno, FIT VUT v Brně, 2012

Modular Multiple Liquidity Source Price Streams Aggregator

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením paní doc. RNDr. Jitky Kreslíkové, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Tomáš Rozsnyó
July 31, 2012

Poděkování

Chtěl bych poděkovat paní doc. RNDr. Jitky Kreslíkové, CSc. za uvedení do problematiky a užitečné rady, které mi poskytla během diplomové práce.

Contents

Preface

Trade is normal daily part of our life. It is so natural activity,that we do not even realise it. The history of trading goes back somewhere to era of Egyptian rulers. These were the times when people traded food, items and other daily used instruments. The basic act of trading was pretty much the same until the creation of a great invention, telephone. With the technology of telephone started a whole new era of trading. With computerized systems this era continued to progress to the point as we know it nowadays. Whilst trading by itself maintained its principles, new types of trade commodities were introduced. The technology allow traders to trade from different parts of the world at the same time. Speed of some trading processes is reduced to millisecond, microsecond in some cases. Today, traders use sophisticated trading systems with a wide range of algorithms to support their trading.

Nowadays high-frequency trading applications are a very frequently used in financial market. Multiple liquidity price aggregator belongs to the group of these applications. The most important qualities required for creating a aggregator application are low latency or in other words processing time, application stability, reliability and security of transferred data. These properties are absolutely mandatory for meeting nowadays foreign exchange market requirements to successfully capture execution opportunities 24x7. To achieve these requirements strict latency testing needs to take place in every part of the aggregator application, followed by continuous cycle of testing and improving by the help of performance profiler.

Chapter 1 brings a global view on financial market background. Describes the present assets, structure, trading types and possible future development trends. Chapter 2 focuses on high-frequency trading from the view of trading strategies, development and cost. Foreign exchange fundamentals are necessary to understand the function of FX market. Price analysis is important for understanding the past, present and future prediction of exchange rates. Fundamental and technical analysis deals with this topic in Chapter 3. FX interbank trading lies mostly on the use of FIX protocol. FIX technical specification and engines providing API or other interface for developing FIX implementation is described in Chapter 4. The heart of the aggregator is a sorting algorithm. Chapter 5 gives detail description to this important topic. Firstly, it includes theoreticall overview, followed by practical results. The research from Chapter 5 allowed to lay the foundation to aggregator design. Aggregator design is covered in Chapter 6. This Chapter specifies ground features of quality aggregator application and according to this specification brings proposal of aggregator, data, user interface and modularity properties. Chapter 7 gives implementation overview of encountered difficulties.

The Master Thesis continues and build on the knowledge obtained in Term Project. The Term Project included first four parts of this Master Thesis. These four parts brought theoretical insight into the huge financial market world. This knowledge helped to define input data for testing and properties for sorting algorithm best suitable for the aggregator. Further, it helped to understand the communication transaction between the liquidity provider and client, trader. Thus a suitable FIX engine can be examined and chosen.

Chapter 1

Financial markets

„Investors, traders and trading systems are increasingly covering multiple markets and asset classes. Therefore, it important to have a basic grounding in all the major markets.“ [7] The following Chapter 1 contains a summary of financial markets most important parts. These informations were obtained mainly from book Algorithmic Trading & DMA [7].

1.1 Main categories

The world's financial markets are enormous both in size and diversity of products they incorporate. We can divide them into several main categories:

- Capital markets
- Foreign Exchange markets
- Money markets
- Derivate markets

The focus of capital markets is on medium and long-term financing through stock and fixed income assets. While the foreign exchange markets enables the transfer of money across currencies. Money markets provide short-term financing, which are closely linked to both foreign exchange and fixed income. The derivatives markets provide a means of trading financial contracts, which are in turn based on underlying assets. The underliers can possibly include stock, bonds, currencies, commodities or even other derivatives. In fact, so much commodity trading is handled through derivatives the commodity market has effectively been subsumed into the derivatives market.

Despite of the strong differences in market identities involving many of the same players (brokers and investors), they are still often treated separately. In part, this is because of their various market structures. Conventionally, trading for equities and listed derivatives has centred on exchanges, while for many of the other asset classes over-the-counter (OTC) trading has predominated.

1.2 Asset classes

Financial assets generally provide their issues with a means of financing themselves and investors with an opportunity to earn income. For instance, stock represents a share in the company that issued them, whereas fixed income assets correspond to loans. Foreign exchanges is a transfer of cash deposits in different currencies. Derivatives offer a way of trading on the future price of assets, providing both a means of insurance and an opportunity for speculation.

Equity
Stock allows companies to finance themselves by actually making their ownership public. Each share represents a portion of the firm's inherent value or equity. This value represents what the corporations remaining assets are worth once all the liabilities have been deducted. A finite number of shares are issued by public corporations, although they may also make periodic dividend payments to shareholders. Shares are forward looking investments, since they represent a portion of the firm's total equity. Investors expect the value of their shares to increase or dividends to be paid in order to compensate them for the risk of bankruptcy.

Fixed income
Fixed income assets, such as bonds, allow both governments and corporations to publicly issue debt for periods up of 30 years, or even perpetually. The issuer is obligated to repay the holder a specific amount (the principal) at a set date in the future (the maturity date). Many names are given to these assets, but we will use the term bond. The issuer of the bond will usually pay interest (the coupon) to the holder at a given frequency for the lifetime of the loan, e.g. every 6 months. Alternatively, zero coupon bonds make no such payments; instead, their price is discounted by an equivalent amount. In general, bond are more straightforward to price than equities.

Foreign exchange
Foreign exchange (FX) represents the trading of currencies, essentially by transferring the ownership of deposits. Generally, what we first think of as foreign exchange is spot trading. FX spot transactions lock-in the current exchange rate for cash settlement, usually within two days. So a U.S. dollar / Japanese yen trade consists of selling U.S. dollars to another counterparty in exchange for a specific amount of yen. There are also FX derivative contracts, namely forwards, options and swaps.

Money markets
The money markets are key to the provision of short-term financing for banks, institutions and corporations. This may range from overnight to around a month, although it can be for as much as a year. Typical money market assets are short-term debt, deposits, repos and stock lending. Therefore, the money markets are also closely linked to both fixed income and foreign exchange.

Derivatives
As their name suggests, derivative contracts are derived from other assets, referred to as the underlying assets. The main types of derivative contract are forwards and futures, options, swaps and credit derivatives. They may be classified as either „listed“ (or exchange-traded) or over-the-counter (OTC). Listed derivatives are standardised contracts that may be traded much like stocks; typically, these are futures and option contracts. OTC contracts are bespoke, so they may represent more complex or exotic derivatives contracts.

Futures and options
A forward contract is an agreement to buy (or sell) a fixed quantity of a given asset at a certain price at a specific date in the future, e.g. a contract to buy 5,000 bushels of grain in July for $3. When the maturity date is reached, the contract expires and the transaction must then be settled. A futures contract is just a forward with standardised terns (e.g. amounts, prices, dates) which is traded on a exchange.

Option contracts are similar to forward and futures in that they allow a price to be struck to trade a set quantity of a given asset at a specific future date. The main difference between futures (or forwards) and options is that the owner of an option is not obliged to trade.

Swaps
A swap is essentially an agreement between two counterparties to exchange cash streams. These may be linked to fixed/floating interest rates, currencies, or even dividend payments. Thus, they offer a flexible means of controlling cash flows, which goes some of the way to explaining their popularity.

Credit derivatives
Credit derivatives act as financial guarantees, which may be used to provide protection against a variety of credit events, but most typically default or bankruptcy. A credit default swap (CDS) is a contract that transfers the credit risk for the notation amount of debt issued by an entity. The purchaser gains protection from this credit risk, whilst the seller is paid regular premiums for the lifetime of the contract. If a credit event occurs then the seller compensate the purchaser and the contract expires.

Other asset classes
A few types of financial asset do not quite fit in the main categories. Most notably depository receipts and exchange traded funds. Both assets classes have seen substantial growth over the last years.

Depository receipts (DRs) represent shares in a foreign company. They provide an alternative to cross-border trading, as well as being an important means of accessing the major financial markets for foreign companies.

Exchange traded funds (ETFs) are tradable share in a investment fund. Since they represent baskets of assets, they are in extremely efficient means of gaining exposure to whole sectors or markets, much like index futures.
For this section I gathered information from [7], Chapter 3.

1.3 Market microstructure

The field of market infrastructure concentrates on actual trading process, analysing how specific mechanisms affect both observed prices and traded volumes. Most of the following information is from [7], Chapter 2.

1.3.1 Function

The fundamental purpose of a market is to bring buyers and sellers together. Broadly speaking, the capital markets may be categorised into primary and secondary markets, based on the stages of assets lifecycle. The primary markets deals with the issuance of new assets/securities. Subsequent trading of these assets takes place on the secondary markets.

New government bonds are generally issued via specialised auctions. For equities, the primary market is concerned with initial public offerings (IPOs), follow-on offerings and right issues. Similarly, new corporate debt is generally placed using underwriters (usually a sindicate of banks).

Historically the second market for bonds has often been „over-the-counter“ (OTC), although there are now also substantial inter-dealer and dealer-to-customer markets. The situation is similar for the trading of foreign exchange and many derivative assets. Whilst for equities the main marketplaces are exchanges, although increasingly these must now compete with other venues such as Electronic Communications Networks (ECNs) and Alternative Trading Systems (ATSs).

The secondary markets are vital since investors will be more willing to provide capital if they know the assets readily be traded. This flexibility allows them to withdraw capital when needed and to switch between assets.

1.3.2 Participants

Conventionally, market roles have been defined by trading needs. The „buy-side“ corresponds to the traditional customers, namely institutional and individual investors. Whilst the „sell-side“ represents the brokers, dealers and other financial intermediaries who service customer needs. Brokers act as agents to facilitate the actual trading, whilst dealers (or market makers) trade on their own behalf trying to profit from offering liquidity. Speculators act independently, trading for themselfs.

In comparison, the market microstructure models in academic literature tend to classify the participants based on the information they posses:

- „Informed traders“ are assumed to have private information, which enables them to accurately determine the assets true value.
- „Liquidity traders“ must trade in order to fulfil certain requirements. such as to release capital or to adjust the balance of a portfolio.

1.3.3 Liquidity

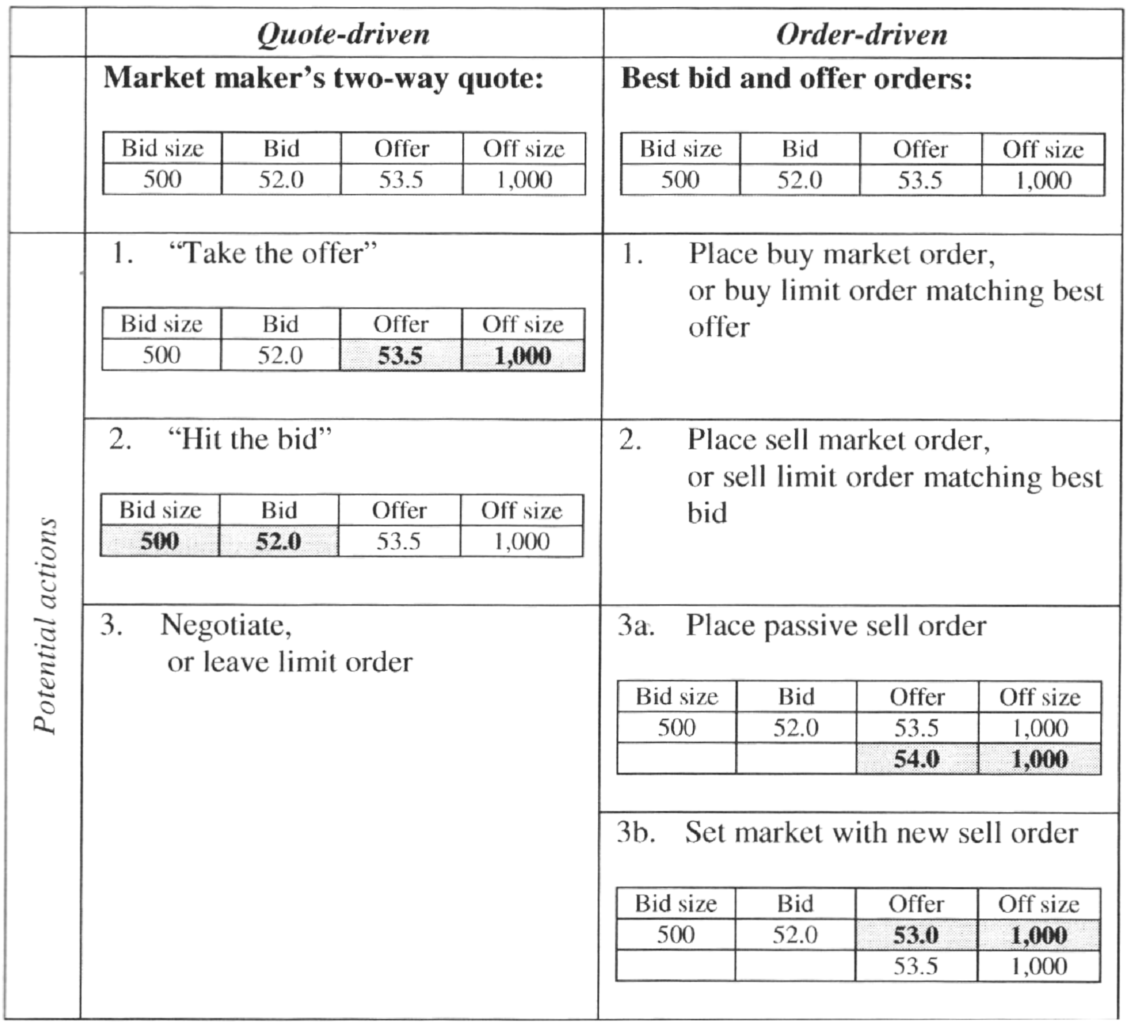
Trading generally means converting an asset into cash or vice versa. How much this conversation actually costs may be represented by the liquidity of the asset, or the market it was traded on. An asset price is closely linked to its liquidity.

1.3.4 Trading mechanism

Markets are generally thought of as being either quote-driven, order-driven or a mix (or hybrid) of the two. A purely quote-driven market means traders must transact with a dealer (or market maker) who quotes prices at which they will buy and sell a given quantity. Order-driven markets allow all traders to participate equally, placing orders on an order book that are then matched using a consistent set of rules.

For a *quote-driven market*, when faced with two-way quote we can choose to „take the offer“, „hit the bid“, negotiate or just leave it. Choosing to „take the offer“ result in a buy execution priced at their offer, whilst hitting the bid results in a sell. The market maker's two-way quote provides a guaranteed execution at that price, for a set size.

With *order-driven market*, the prices are established by actual orders. The best bid price



	Quote-driven	*Order-driven*
	Market maker's two-way quote:	**Best bid and offer orders:**
Potential actions	1. "Take the offer"	1. Place buy market order, or buy limit order matching best offer
	2. "Hit the bid"	2. Place sell market order, or sell limit order matching best bid
	3. Negotiate, or leave limit order	3a. Place passive sell order
		3b. Set market with new sell order

Market maker's two-way quote / Best bid and offer orders:

Bid size	Bid	Offer	Off size
500	52.0	53.5	1,000

1. "Take the offer"

Bid size	Bid	Offer	Off size
500	52.0	**53.5**	**1,000**

2. "Hit the bid"

Bid size	Bid	Offer	Off size
500	**52.0**	53.5	1,000

3a. Place passive sell order

Bid size	Bid	Offer	Off size
500	52.0	53.5	1,000
		54.0	**1,000**

3b. Set market with new sell order

Bid size	Bid	Offer	Off size
500	52.0	**53.0**	**1,000**
		53.5	1,000

Figure 1.1: Comparing quote-driven and order-driven trading mechanisms [copy from 7]

represents the highest priced buy order, whilst the best offer is set by the lowest priced sell order. A trade may only occur when a buy order matches (or betters) the current best offer price, whilst for a sell order the best bid price is the target. So instead of responding to the market maker's two-way quote we react to the available liquidity on the order book. Figure 1.1 shows both types of trading mechanisms.

1.3.5 Trading frequency

The frequency of trading is the other main classifier for the structure of markets, since this determines when requirement matches (whether they are from quotes or orders) are actually turned into executions. Generally, one or more of the following types:

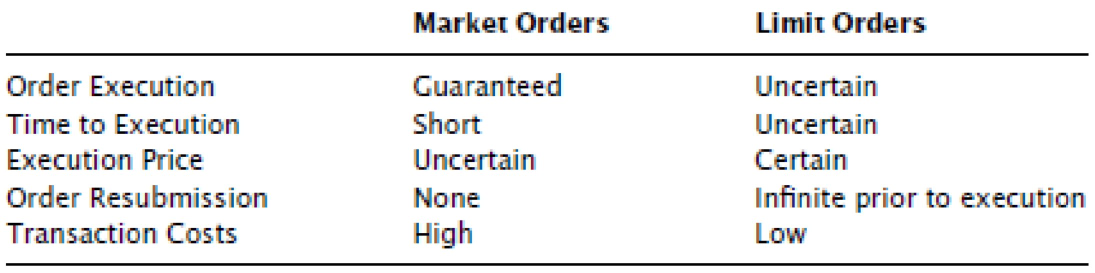
- Continuous trading
- Periodic trading
- Request-driven trading

Continuous trading provides a convenient and efficient means of execution. Although such immediacy can lead to price volatility, particularly when there is a imbalance between supply and demand. Periodic trading is generally scheduled for specific time/s in the day. The time period beforehand permits more considered price formation, it also allow liquidity to accumulate. Request-driven trading means requesting a quote from a market maker, whilst it may be convenient it is not necessarily as efficient in terms of the price achieved.

1.3.6 Order types

Orders also play an important role in market structure. An order is simply an instruction to buy or sell a specific quantity of a given asset. Market microstructure tends to differentiate orders both by their liquidity-effect and by their associated risks. The two main types are:

- Market orders - these are directions to trade immediately at the best price available. Hence, they demand liquidity and risk execution price uncertainty.
- Limit orders - these have an inbuilt price limit that must not be breached, a maximum price for buys and minimum price for sell orders. Thus, limit orders can help provide liquidity but risk failing to execute.



	Market Orders	**Limit Orders**
Order Execution	Guaranteed	Uncertain
Time to Execution	Short	Uncertain
Execution Price	Uncertain	Certain
Order Resubmission	None	Infinite prior to execution
Transaction Costs	High	Low

Table 1.1: Limit Orders versus Market Orders [copy from 7]

More market attributes can be seen in Table 1.1. Note that markets may also differ in terms of the behaviour of these order types. This Chapter is covered in much more detail in Chapter 3 of [7].

1.4 Market structure

To start off the market is so specialised that there may only be a single dealer prepared to „make a market“ for this. Any trading invariably a one-to-one process between the dealer and each client (investor), as show in Figure 1.2(a).

Gradually the number of investors (or potential clients) will increase and so more dealers will get involved. This enables clients to check prices with several dealers before deciding to trade, as shown in Figure 1.2(b). Trading is still a bilateral and quote-driven, but at least there is more choice. This type of structure is the basis for much of the „over-the-counter“ (OTC) trading which takes place for a wide range of assets, from stock to bonds and derivatives.

Eventually the marketplace may become so large that there is enough trading (and participants) to warrant dedicated execution venues, as shown in Figure 1.2(c). Trading for many of the world's assets is now based on market structures similar to Figure 1.2(c).

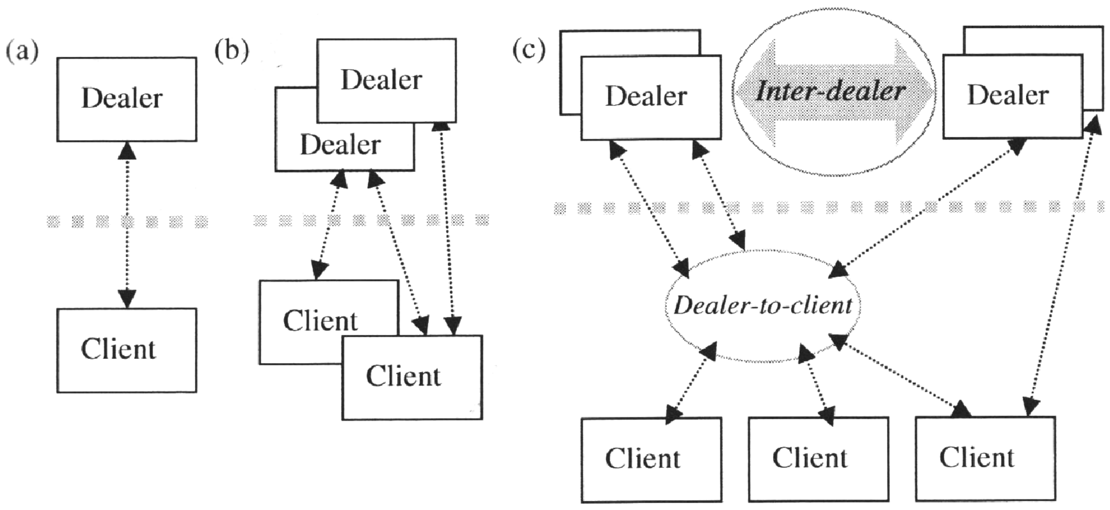
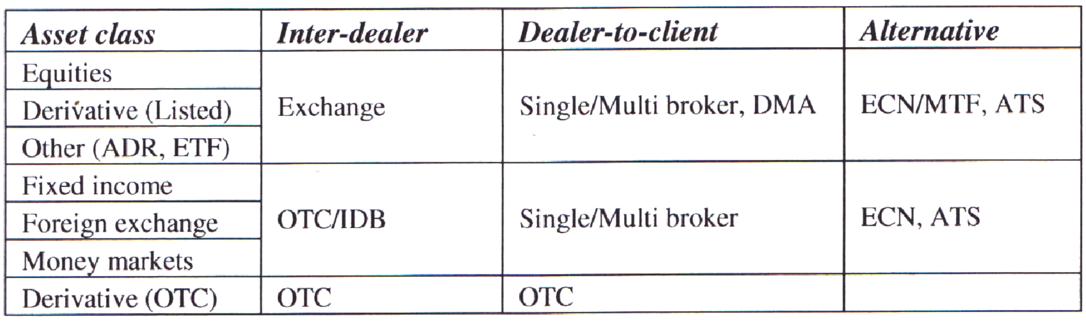


Figure 1.2: Three phases of marketplace development [copy from 7]



Asset class	*Inter-dealer*	*Dealer-to-client*	*Alternative*
Equities	Exchange	Single/Multi broker, DMA	ECN/MTF, ATS
Derivative (Listed)			
Other (ADR, ETF)			
Fixed income	OTC/IDB	Single/Multi broker	ECN, ATS
Foreign exchange			
Money markets			
Derivative (OTC)	OTC	OTC	

Table 1.2: Market structure across asset classes [copy from 7]

Table 1.2 tries to provide a broad summary of these market structures for the major asset classes, although inevitably there are some isolated exceptions.

Over the last few years, the strict segmentation between the dealer and client markets, visible in Figure 1.2, has started to blur. Alternative trading venues (accessed by both dealers and clients) have become increasingly important. They are the Electronic Communication Network (ECNs) or Multilateral Trading Facilities (MTFs) and „dark pool" Alternative Trading Systems (ATSs) which have started to spread across the world's markets. These further complicate the market structure, to the point where clients may even use certain venues to bypass dealers altogether.

Inter-dealer markets

A strong inter-dealer market is vital for liquidity. Otherwise, dealers may be less willing to trade beyond their current inventory. Having a market where they can easily trade with other dealers enables them to control their positions and so meet their clients' requirements.

Dealer-to-client markets
Traditionally, bridging the gap between dealers and clients has been the domain of brokers. Nowadays dealer-to-client (D2C) markets are represented by a mix of phone-based trading, single and multi-broker/dealer electronic trading platforms, provided by a range of brokers and third-party vendors.

Alternative markets
Alternative trading venues have become more and more important over the last few years. Many of these venues serve both the buy and sell-side, so they effectively straddle the strict segmentation we saw back in Figure 1.3. Hence, market structures are becoming increasingly complex, as we can see in Figure 1.3.

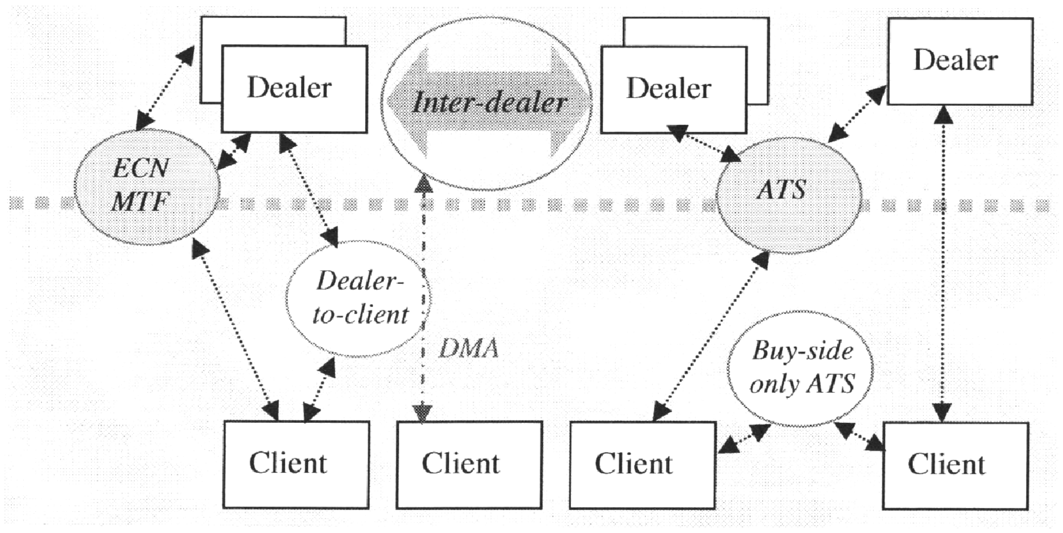


Figure 1.3: Marketplace interconnections for alternative venues [copy from 7]

Brokers/dealers have started to to provide DMA access to exchanges and other execution venues (shown as the grey dashed line). Clients can also get direct access to many of the ECNs and ATSs; so they are no longer restricted to dealer-to-client platforms. In fact, some ATSs are buy-side only, completely bypassing the broker/dealers. Effectively, these are client-to-client trading networks [7].

1.5 Institutional trading types

Core execution methods refer to methods used to actually execute orders. Main groups are:

- Manual Trading
- Algorithmic Trading
- Direct Market Access & Crossing

Manual trading or „High-touch“ trading is where orders are worked manually by a trader.

„Algorithmic trading is simply a computerised rule-based system responsible for executing orders to buy or sell a given asset.“ [7] Algorithmic trading is sometimes referred to as „Low-touch“ trading, since it requires little or no handling by actual traders and so can be offer as a lower cost agency service. So a trading algorithm is just a computerised model that incorporates the steps required to trade an order in a specific way. Admittedly, for the algorithm to react to ever changing market conditions these rules can become quite complex.

The final piece of the puzzle is DMA, which is also referred to as „Zero-touch“. „Direct Market Access (DMA) enables clients to send orders to exchanges by using their broker's membership“ [7]. With DMA the broker's own electronic access to markets is extended out out to their clients. The sell-side traders have nothing to do with the order; instead, the execution is handled manually by the client. Crossing is used by institutions who often need to trade in large sizes, but large block orders can expose them to substantial price risk. Crossing systems provide an electronic mechanism allowing investors to carry out their own block trading anonymously.

The increasing focus on transaction costs by the buy-side has meant a decline in the more traditional „High-touch“ trading. Still, all these methods are in fact complementary, since they are trying to meet the same objectives.

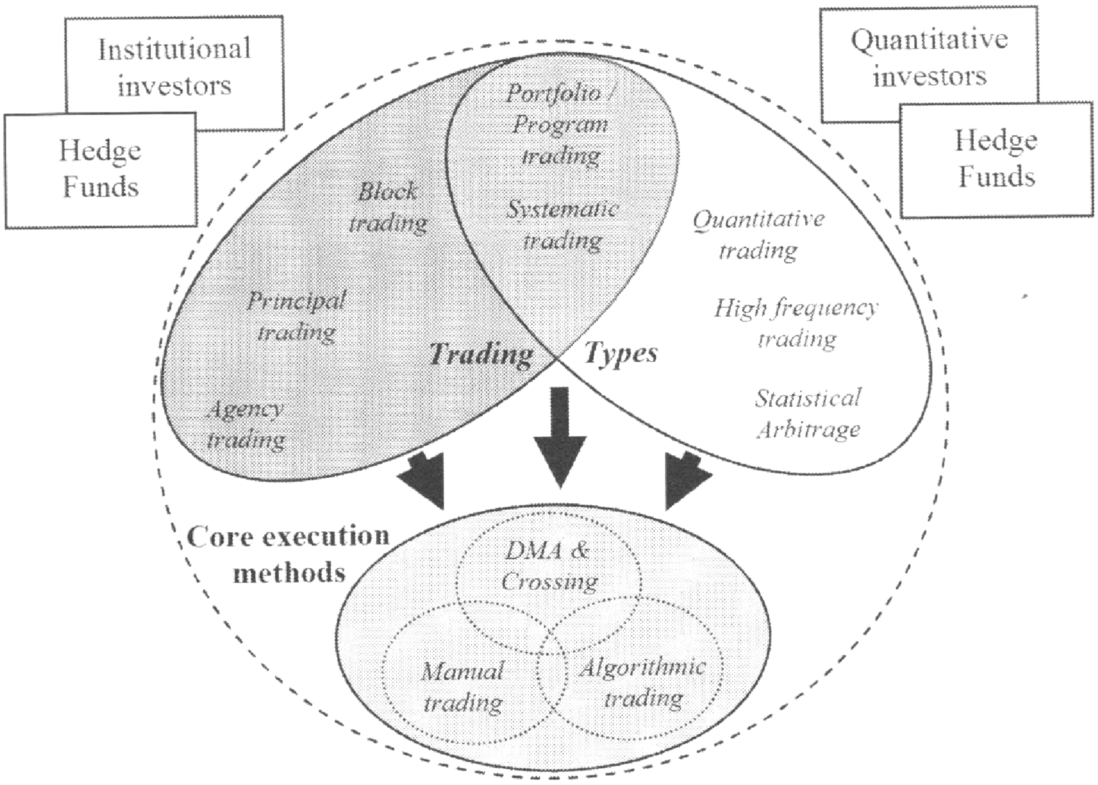


Figure 1.4: Different trading types [copy from 7]

Traditionally, institutional investors, such as investment and pension funds, maintain large portfolios with specific investment criteria. Orders are generated when they need to change the make-up of their portfolios. For single assets, they may choose to trade in a either an agency or principal fashion, whilst block trading may be used for larger orders.

Quantitative investment funds adopt more highly automated strategies, as do some hedge funds. For those targeting short-term arbitrage opportunities or generating revenue by market making, this means much higher trading frequencies. So they are even more focussed on low-cost execution methods, such as algorithmic trading and DMA.

Portfolio trading is sometimes referred to as basket or program trading. It provides investors with a cost-effective means of trading multiple assets, rather than having to trade them individually. Systematic, quantitative („Black-box“) and high-frequency trading are terms which all sound like references to algorithmic trading, and are sometimes mistakenly used as such. However, they have as much to do with the style of investment as the actual trading. In fact, they are all forms of systematic trading (or investment), and are sometimes referred to as Automated trading. Predominantly, these strategies are adopted by either quantitative investors or proprietary trading desk.

Systematic trading as its name suggests, is all about consistently adopting the same approach for trading. This may be used to dictate points for trade entry and exit.

Quantitative („Black-box“) trading is often confused with algorithmic trading. Here the trading rules are enforced by adopting proprietary quantitative models. The difference is fairly subtle, but quantitative trading systems instigate traders whereas algorithmic trading systems merely execute them.

High-frequency trading aims to take advantage of opportunities intraday. The time scale involved range from hours down to seconds or even fractions of second. Effectively, it is a specialised form of black-box/quantitative trading focussed on exploiting short-term gains.

Statistical arbitrage represents a systematic investment/trading approach, which is based on a fusion of real-time and historical data analysis. The main difference from high-frequency trading is that strategies may span over longer timeframes. Other than this, the goals are generally the same, both try to take advantage of mispricing whilst minimising the overall exposure to risk. Strategies try to find trends or indicators from previous data (intraday and/or historical) and then use these to gain an edge [7].

1.6 Global market trends

There are certainly differences amongst the world's major markets; however, there are also many similarities. A focus on costs, both the buy- and sell-side, and the onward march of technology, mean that the markets for each asset class are exhibiting many of the same trends, albeit at different rates. Notably, many of these markets are seeing increases in:

- Electronic trading
- Transparency
- Accessibility

Electronic trading
Electronic trading is already widespread; indeed, for many markets it is now the norm. Order placement and execution times are now measured in fraction of a second. In comparison, the old U.S. equity inter-market (ITS) allowed dealers up to 30 seconds. We are currently in the midst of a race between venues to provide the fastest platforms. A few years ago,

a latency, or delay of 300 milliseconds (10^{-3}s) used to be perfectly reasonable. Nowadays cutting-edge venues are targeting latencies in microseconds (10^{-6}s). In the light of this, co-location services are becoming increasingly popular. These allow market participants to host their computerised trading systems in the same machine rooms as the execution venue, thus minimising any transmission-related delays.

Average order sizes have seen considerable shifts as well. In some markets, such as FX and U.S. listed options, order sizes are actually increasing, in order to cope with the huge increases in trading these markets are experiencing. However, the long-term effect of electronic trading is probably best illustrated by the equity market. Over the last ten years, there has been a steady decline in average order size on electronic trading platforms for equities.

Trading volumes have also rocketed over the last few years, in part due to the easy access and lower afforded by electronic trading.

Markets will continue to evolve in order to adapt to the changes which electronic trading brings. Algorithmic trading is a key tool to help cope with these changes [7]. According to estimates conducted by Boston-based Aite Group, shown in Figure 1.5, adoption of electronic trading has grown from 25 percent of trading volume in 2001 to 85 percent in 2008. Close to 100 percent of equity trading is expected to be performed over the electronic networks by 2010.

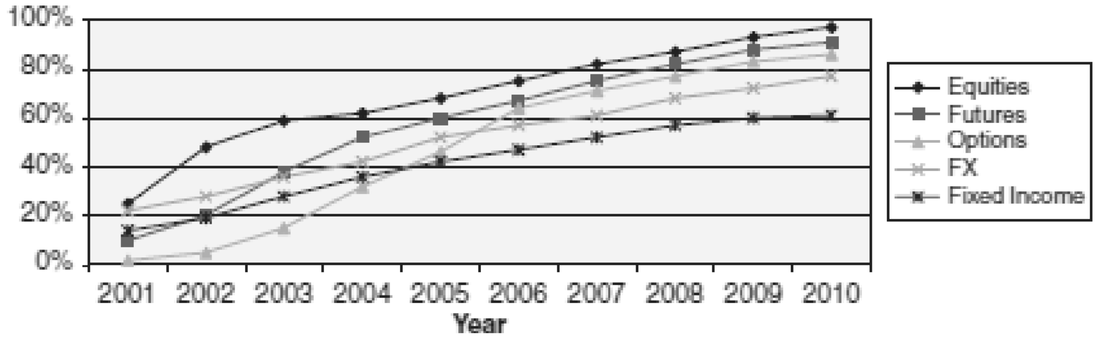


Figure 1.5: Adoption of electronic trading capabilities by asset class [copy from 7]

Transparency
Transparency is also improving, for both pre- and post-trade information.

Accessibility
Accessibility has also become a key issue as buy-side traders seek to gain access to the same venues as their sell-side counterparties. Running alongside these trends is seemingly endless cycle of centralisation and fragmentation. New venues keep appearing, attracting order flow and fragmenting the available liquidity. Then within a few years, consolidation means the market starts to centralise again [7].

Chapter 2

High-frequency trading

Chapter 2, gives an insight into high-frequency trading. What is typical for high-frequency trading, strategies according to trading frequencies and basic development information. The following Chapter information can be found in High-frequency Trading book [1].

2.1 Overview

The main innovation that separates high-frequency trading from low-frequency trading is a high turnover of capital in rapid computer-driven responses to changing market conditions. High-frequency trading strategies are characterized by a higher number of trades and a lower average gain per trade. Many traditional money managers hold their trading positions for weeks or even months, generating a few percentage points in return per trade. By comparison, high-frequency money managers execute multiple trades each day, gaining a fraction of a percent return per trade, with few, if any, positions carried overnight [7] [1]. The absence of overnight positions is important to investors and portfolio managers for three reasons:

1. The continuing globalization of capital markets extends most of the trading activity to 24-hour cycles, and with the current volatility in the markets, overnight positions can become particularly risky. High-frequency strategies do away with overnight risk.

2. High-frequency strategies allow for full transparency of account holdings and eliminate the need for capital lock-ups.

3. Overnight positions taken out on margin have to be paid for at the interest rate referred to as an overnight carry rate. The overnight carry rate is typically slightly above LIBOR (is among the most common of benchmark interest rate indexes used to make adjustments to adjustable rate mortgages). With volatility in LIBOR and hyperinflation around the corner, however, overnight positions can become increasingly expensive and therefore unprofitable for many money managers. High-frequency strategies avoid the overnight carry, creating considerable savings for investors in tight lending conditions and in high-interest environments.

We can sum up main key characteristics of high-frequency trading in these points:

- Tick-by-tick data processing
- High capital turnover

- Intra-day entry and exit of positions
- Algorithmic trading

2.2 High-frequency strategies

High-frequency trading has additional advantages. High-frequency strategies have little or no correlation with traditional long-term buy and hold strategies, making high-frequency strategies valuable diversification tools for long-term portfolios. High-frequency strategies also require shorter evaluation periods because of their statistical properties [7]. The Figure 2.1 show us that for different various trading instruments suits different optimal trading frequencies:

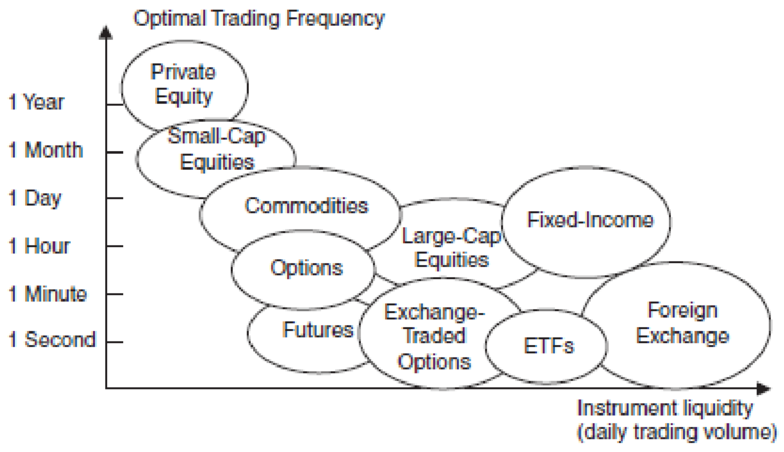
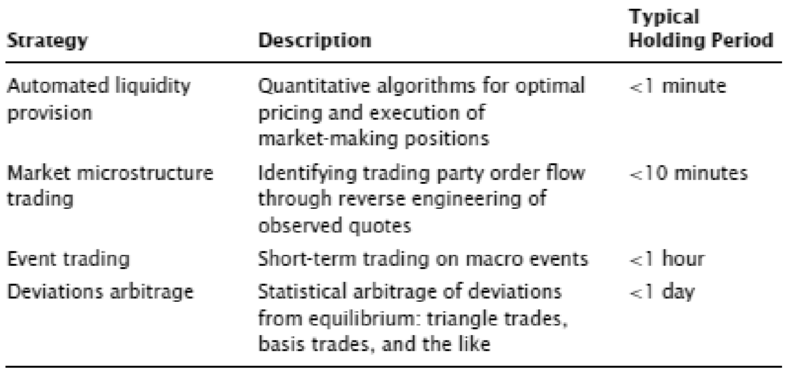


Figure 2.1: Optimal trading frequency for various trading instruments, depending on the instrument's liquidity [copy from 1]

Many successful high-frequency strategies run on foreign exchange, equities, futures, and derivatives. By its nature, high-frequency trading can be applied to any sufficiently liquid financial instrument. A "liquid instrument" can be a financial security that has enough buyers and sellers to trade at any time of the trading day.

Currently, four classes of trading strategies are most popular in the high-frequency category: automated liquidity provision, market microstructure trading, event trading, and deviations arbitrage. Table 2.1 summarizes key properties of each type [1].



Strategy	Description	Typical Holding Period
Automated liquidity provision	Quantitative algorithms for optimal pricing and execution of market-making positions	<1 minute
Market microstructure trading	Identifying trading party order flow through reverse engineering of observed quotes	<10 minutes
Event trading	Short-term trading on macro events	<1 hour
Deviations arbitrage	Statistical arbitrage of deviations from equilibrium: triangle trades, basis trades, and the like	<1 day

Table 2.1: Classification of High-frequency Strategies [copy from 1]

2.3 Development and cost

Developing high-frequency trading presents a set of challenges previously unknown to most money managers:

1. Dealing with large volumes of intra-day data. Intra-day data is much more voluminous and can be irregularly spaced, requiring new tools and methodologies.

2. Precision of signals. Since gains may quickly turn to losses if signals are misaligned, a signal must be precise enough to trigger trades in a fraction of a second.

3. Speed of execution. The only reliable way to achieve the required speed and precision is computer automation of order generation and execution.

Three main components, shown in Figure 2.2 , make up the business cycle:

- Highly quantitative, econometric models that forecast short-term price moves based on contemporary market conditions.

- Advanced computer systems built to quickly execute the complex econometric models.

- Capital applied and monitored within risk and cost-management frameworks that are cautious and precise.

The main difference between traditional investment management and high-frequency trading is that the increased frequency of opening and closing positions in various securities allows the trading systems to profitably capture small deviations in securities prices. When small gains are booked repeatedly throughout the day, the end-of-day result is a reasonable gain.

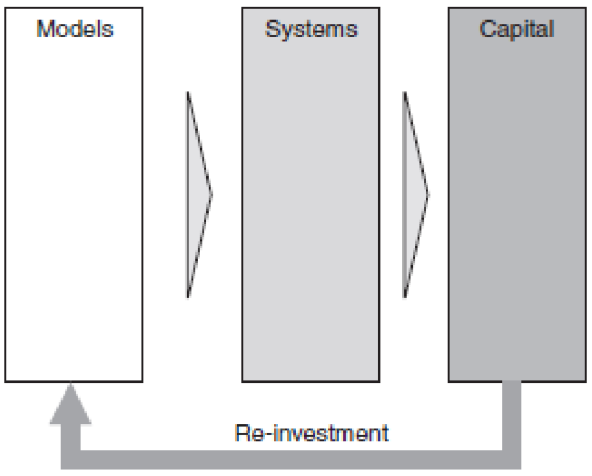


Figure 2.2: Development cycle of a high-frequency trading process [copy from 1]

Developing a high-frequency trading business follows a process unusual for most traditional financial institutions. Designing new high-frequency trading strategies is very costly. On the other hand executing and monitoring finished high-frequency products costs close to nothing. By contrast, traditional proprietary trading businesses incur fixed costs from the moment an experienced senior trader with a proven track record begins running the trading desk and training promising young apprentices, through the time when the trained apprentices replace their masters. In a long run Figure 2.3 show us general view on cost, of course this may vary from project to project [1].

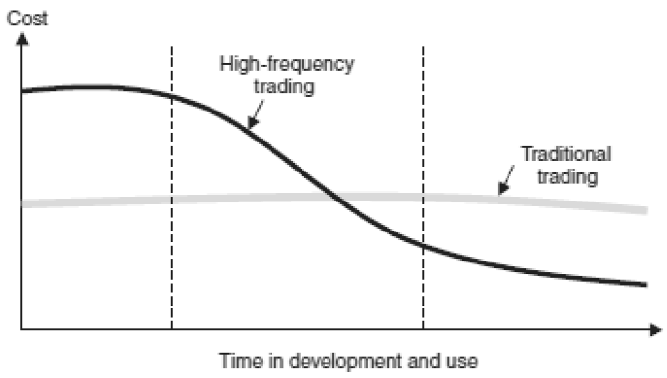


Figure 2.3: Economics of high-frequency versus traditional trading businesses [copy from 1]

Developing a reliable, low reaction or processing time response and tested high-frequency trading application solution can take minimum 2+ years. A proper profound testing on older market data is crucial, before live use of the application in market. Any unpredicted reaction to market changes can lead to huge financial losses. Non less important is maintenance in the means of developing new algorithms to widen the applications capabilities and adapt to market changes.

Chapter 3

Foreign exchange (FX or FOREX)

The following Chapter 3, gives general information about FX, further it explains elementary terms to understand the financial dialogue. Predicting upturns and downturns of currencies is the responsibility of price analysis. We will look into this topic in part three of this Chapter.

3.1 General information

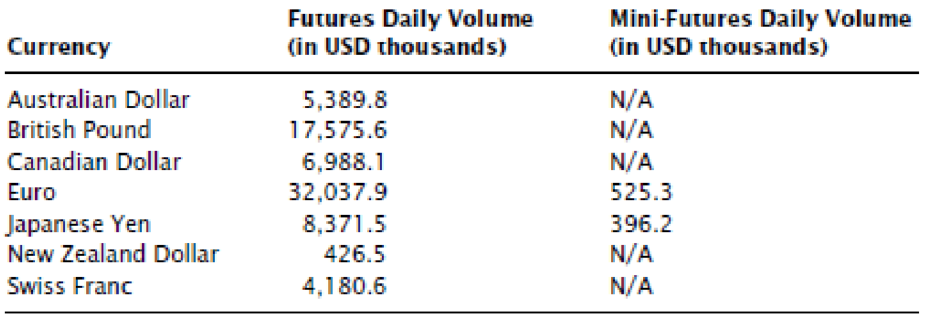
It is a global, world-wide decentralized financial market for trading currencies and most importantly it is the largest and most liquid market on the planet with daily turnover around $4 trillion. Trade goes on 24 hours a day, 7 days a week. FX is a OTC (over-the-counter) or off-exchange market trading directly between two parties. The core players of the FX market are central banks, commercial banks and investment banks. This type of market is called interbank or inter-dealer market. At present, inter-dealer trading is declining in importance as the proportion of client-driven trading increases, particularly from hedge funds.

Trading of currencies is performed essentially by the transfer of ownership of deposits. Therefore, a U.S. dollar / Japanese yen trade consists of one counterparty selling U.S. dollars to another in exchange for a specific amount of yen. Seven major currencies include:

- USD (US dollar)
- EUR (Euro)
- GBP (British Pound)
- JPY (Japanese Yen)
- CHF (Swiss Franc)
- CAD (Canadian dollar)
- AUD (Australian dollar)

Naturally the FX incorporate other currencies, but they account just for about 10% of daily trade. The major pair traded on the FX are EUR/USD, GBP/USD, USD/CHF, USD/JPY, USD/CAD, AUD/USD. FX trading is largely concentrated on a small subset of currencies. In fact, around two thirds of all trading involves just four major currencies.

Table 3.1 shows daily electronic trading volumes in most common foreign exchange futures on CME Electronic Trading (Globex) on 6/12/2009 computed as average price times total contract volume [9] [22] [24] [1].



Currency	Futures Daily Volume (in USD thousands)	Mini-Futures Daily Volume (in USD thousands)
Australian Dollar	5,389.8	N/A
British Pound	17,575.6	N/A
Canadian Dollar	6,988.1	N/A
Euro	32,037.9	525.3
Japanese Yen	8,371.5	396.2
New Zealand Dollar	426.5	N/A
Swiss Franc	4,180.6	N/A

Table 3.1: Daily Dollar Volume in Most Active Foreign Exchange Products on CME [copy from 7]

3.2 Fundamentals

The currencies are always traded in two-way quoted pairs. Incorporates two currencies and Sell/Buy price respectively more common used is Bid/Ask price. When we choose to „take the Offer" or „hit the Bid" we must set the quantity. The quantity goes always along with the Bid/Ask price as show in Figure 3.1. For understanding FX market fundamentals is crucial to understand term related to FX trading. The next section brings explanation to these terms. Provided information is processed from [1] [9].

Spread is the difference between the Ask and Bid price. In Figure 3.1 is related term Breadth. Spread changes with liquidity and volatility.

Liquidity can be expressed in terms of immediacy, which reflects the ability to trade immediately by executing at the best price. Asset price is closely linked to its liquidity.

Volatility or price fluctuations linked to a time parameter.

The bid price and the ask price are defined for liquidity quantities OA and OA' that represent market depths at bid and ask prices, respectively.

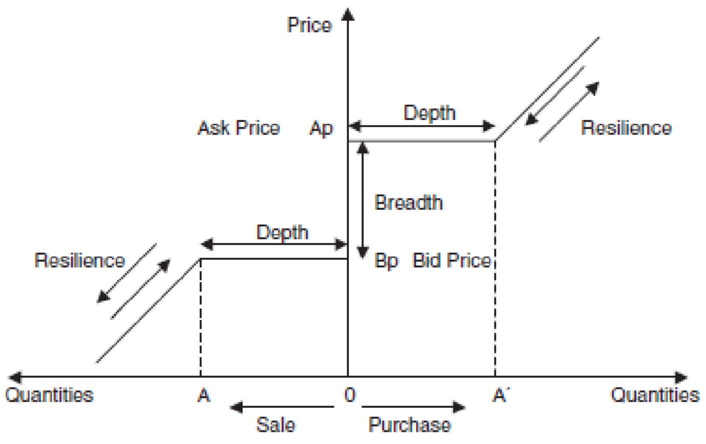


Figure 3.1: Aspects of market liquidity [copy from 7]

Slippage is the difference between the posted price and the actually filled price. The higher the volatility and the lower the liquidity, the higher the spread and the higher the slippage.

Pip (Point) is the smallest measure of price move, regardless of fractional representation: 1.3345, 23.23, 0.5168. For USD/JPY a pip=0.01 and for most other currency pairs a pip=0.0001.

1 LOT = 10000 units of base currency

1 MINILOT = 10000 units of base currency

Margin is the minimum amount of money to be deposited with your broker to maintain a desired position size.

Margin call is request to add money to your account to maintain the open position(s).

Leverage is the position size divided by margin.

3.3 Price analysis

Pricing foreign exchange rates can be quite difficult, since they are affected by a wide range of economic factors. There are two main approaches to analyse fluctuations in exchange rates, fundamental and technical analysis.

Fundamental analysis
Fundamental analysis (FA) attempts to predict impact of economic and domestic data on currency fluctuations. Macroeconomic information affects exchange rates directly, but also indirectly via order flow, or net buying pressure. Subsequent studies have shown that order flow can both can explain and forecast changes in exchange rates. Both exchange rates and order flows react to macroeconomic factors such as changes in economic growth, inflation, interest rates or budget/trade balances, quarterly GDP data, consumer sentiment indices,

central bank governor speeches. Any such information, which might reflect future order flows, can have an important, and often immediate, effect on exchange rates. Most of these factors are interconnected with specific dates. A so called macroeconomic calendar hold these dates. Unexpected events like war, nature catastrophes cause huge exchange rate fluctuations. Another factor, which can have a considerable effect on foreign exchange, is the possibility of central bank interventions. Although these may be able to smooth short-term fluctuations, it is just no longer feasible to prop up a currency for any length of time [15] [9]. Exchange rate prognosis prediction can normally incorporates three steps:

1. Global analysis - examines over econometric situation
2. Field analysis - examines trends in the field
3. Analysis of specific entity - review the value of stocks, commodities and currencies

Technical analysis
Technical analysis (TA) is a market dynamics research with the help of statistics, tables, graphs, indicators, etc.. To determine the trend in prices. Technical analysis consists of several approaches and views on the development of price movements. An objective interconnection of these movements is crucial for evaluating the past [15]. Various methods of technical analysis can be summarized into three basic groups:

1. Evaluation of directions and trends
2. Exploration and chart evaluation
3. Examination of various indicators

Chapter 4

Financial market communication protocol

Raising popularity of algorithmic trading and large number of algorithms involved poses issue for OMS (Order Management Systems)/ EMS (Execution Management Systems) vendors. Keeping up with the rapid flow of new algorithms is difficult, particularly for those providing algorithms from a wide range of brokers. In result delays can occur before clients can get access to the latest algorithms [4]. FIX protocol was designed to bring solution to this issue. Chapter 4 brings insight to market communication protocol, FIX.

4.1 FIXatdl

Financial Information Exchange (FIX) group provided a possible solution. XML schema based Algorithmic Trading Definition Language (FIXatdl). Provides standard framework for algorithm parameters defining:

- Core data, such as their type and FIX tag
- Validation rules
- User interface requirements

FIXatdl is built on top of the widely adopted FIX Protocol and allows firms receiving orders to specify exactly how their electronic orders should be expressed. Orders built using FIXatdl can then be transmitted from traders' systems via the FIX Protocol. All versions of the underlying FIX Protocol are fully supported and zero changes to underlying FIX infrastructure are required.

4.2 FIX protocol

The market participants share a vision of common communication language for the automated trading if financial instruments. As a result of this vision the FIX protocol was developed through the collaboration of banks, exchanges and other financial market members around the world. The protocol's specification ownership and maintenance is goal of FIX Protocol, Ltd.. Another goal of FIX Protocol, Ltd. is to keep it under public domain. FIX has become de-facto standard for many markets. It is intended for the exchange of any information related to securities transactions. Hence, it supports the transmissions of orders, market data, security information and settlement details. It exists in two main formats, a strictly machine-readable protocol and one based around XML schema (FIXML) [21] [4].

4.2.1 Communication system

Fix protocol is used for communication involving two entities, the Customer and the Supplier. Customer initiates connection by opening TCP socket. Usually before TCP handshake occurs, authentication process has to carry out successfully. Followed by creation of secured, encrypted connection between these entities. After these steps TCP handshake can take its place.

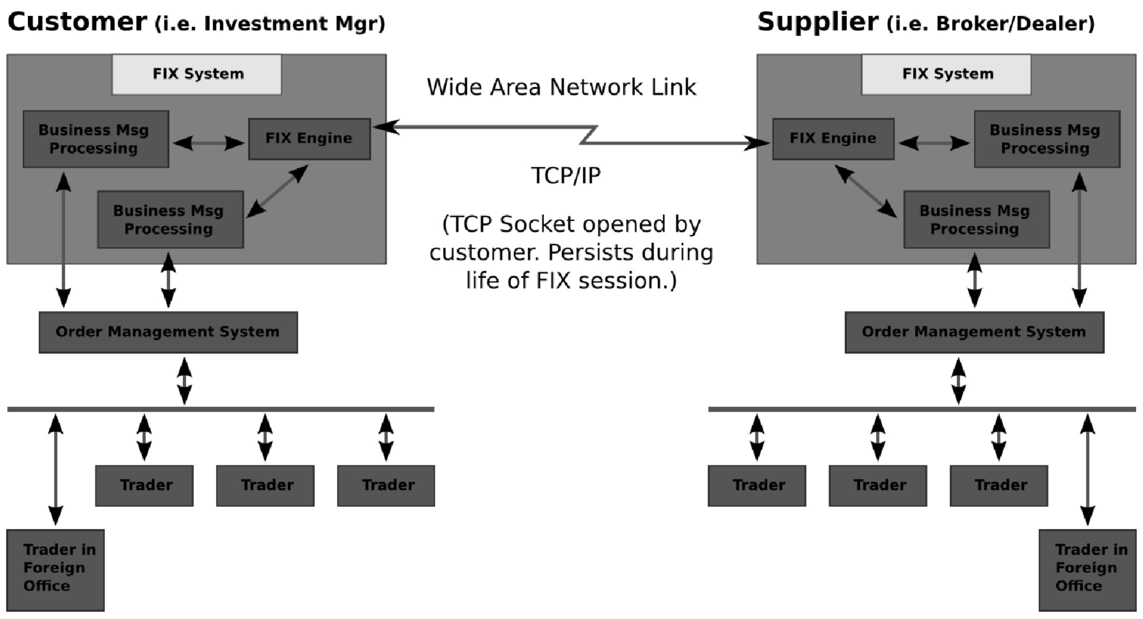


Figure 4.1: FIX communication system example [inspired 21]

4.2.2 Technical specification

Each message consists of three parts, header, body and trailer. The message fields are delimited using the ASCII 01 (start of header) character.

Header

Up to FIX.4.4, the header contained three fields tags:

- 8 (BeginString),
- 9 (BodyLength),
- 35 (MsgType).

From FIXT.1.1 / FIX.5.0, the header contains five mandatory fields and one optional field:

- 8 (BeginString),

- 9 (BodyLength),
- 35 (MsgType),
- 49 (SenderCompID),
- 56 (TargetCompID),
- 1128 (ApplVerID - if present must be in 6th position).

Body
The body of the message is entirely dependent on the message type defined in the header (TAG 35, MsgType).

Trailer
The last field of the message is TAG 10, FIX Message. It is always expressed as a three digit number (e.g. 10=002). It may be used for validation.

Example
Example of a FIX message:
Execution Report (Pipe character is used to represent SOH character)

```
 8=FIX.4.2 | 9=178 | 35=8 | 49=PHLX | 56=PERS |
52=20071123-05:30:00.000 | 11=ATOMNOCCC9990900 | 20=3 | 150=E | 39=E | 55=MSFT |
167=CS | 54=1 | 38=15 | 40=2 | 44=15 | 58=PHLX EQUITY TESTING | 59=0 | 47=C |
32=0 | 31=0 | 151=15 | 14=0 | 6=0 | 10=128 |
```

Understanding the FIX message structure is important in later phase of designing the aggregator. Particularly in designing the database table structures, where quotes values are to be stored.

4.2.3 Engines

FIX engine or in other words API. These APIs provide is an easier way of handling communication on Application layer level. Four engines are examined:

- **ValidFIX:** Is a company whose product scale involves range of applications to manage FIX data. The applications are web-based.
- **VersaFIX:** Its a full featured, open source FIX engine for .NET platform supporting FIX messages from type 4.0 through 5.0.
- **QuickFIX:** Is a full-featured open source FIX engine, currently compatible with the FIX 4.0-5.0 spec. It runs on Windows, Linux, Solaris, FreeBSD and Mac OS X. API's are available for C++, .NET, Python and Ruby. The developers offer QuickFIX Log Viewer, designed to parse FIX messages out of a given file, even if there are other messages within, making it flexible and usable on more general log files.
- **UL FIX:** Free FIX engine built on pure Java-based solution. The main features are unlimited message size, modular persistence, modular transport layer, raw FIX message listener.

Taking in consideration the fact that C# .NET was chosen as development platform for the aggregator, two possibilities are left, VersaFIX and QuickFIX. As referred in Ing. Tomáš Olejník diploma thesis [16] VersaFIX's more sophisticated implementation became a commercial tool and also instability issues during connection sessions brought me to a decision to chose QuickFIX engine. QuickFIX/n is the C# branch of QuickiFIX.

4.3 Interbank connection interfaces

The following three bank institutions were chosen to provide liquidity for the aggregator:

- UBS
- Morgan Stanley
- Deutsche Bank

Chapter 5

Price aggregation

The most important part of price aggregation process is value sorting. The following Chapter 5 provides insight into couple of sorting algorithms. Starting from classification, general algorithm description, continuing with testing and finishing with results and conclusion.

5.1 Sorting algorithm's classification

Common classification of sorting algorithms includes the following parameters:

- Time complexity
- Memory (space) complexity
- Stability
- Recursion
- Computational complexity of swaps
- General sorting method: insertion, exchange, etc.
- Type of input data for sorting
- Adaptability

Three mostly evaluated values are time and space (memory) complexity with stability or instability of algorithm. „A sorting algorithm is said to be stable if it preserves the relative order of items with duplicated keys in the file“ [19].

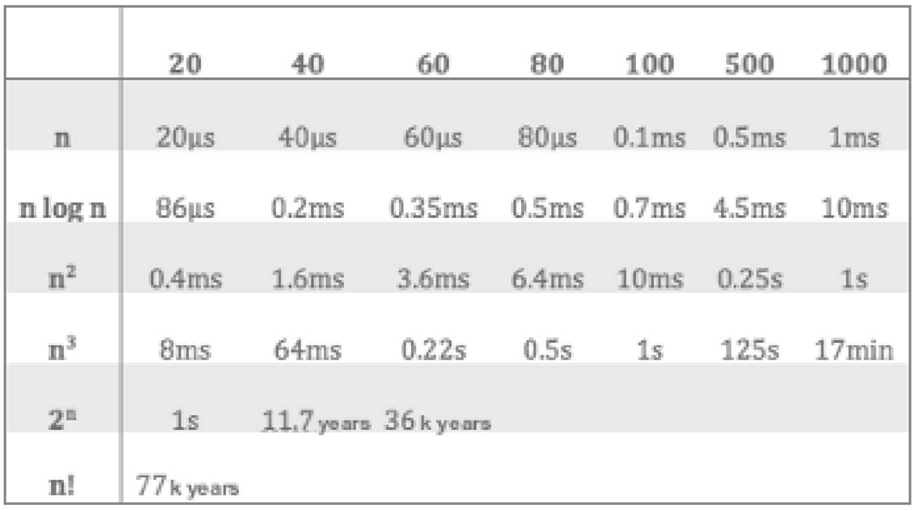
Time complexity represented by well known Omicron (Big O) notation expresses the upper limit of the time behaviour of the algorithm. Further Omega (Ω) notation represents the lower limit of time behaviour of the algorithm and Theta (Θ) denotes the time behaviour as the given function. More detailed description of complexity is out of scope of this work and can be found for example in Algorithms and Data Structures [6].

Space complexity represents the memory space required for the execution of code and data of algorithm. Most frequent complexities are:

- Θ (1) constant complexity
- Θ log(n) logarithmic complexity

- Θ (n) linear complexity
- Θ (n*log(n)) „linearithmic“ complexity
- Θ (n*n) or Θ (n^2) quadratic complexity
- Θ (n*n*n) or Θ (n^3) cubic complexity
- Θ $(k \exp n)$ (where k is real positive number, mostly integer) exponential complexity

For better imagination Table 5.1 shows time complexity examples.



	20	40	60	80	100	500	1000
n	20µs	40µs	60µs	80µs	0.1ms	0.5ms	1ms
n log n	86µs	0.2ms	0.35ms	0.5ms	0.7ms	4.5ms	10ms
n^2	0.4ms	1.6ms	3.6ms	6.4ms	10ms	0.25s	1s
n^3	8ms	64ms	0.22s	0.5s	1s	125s	17min
2^n	1s	11.7 years	36 k years				
n!	77 k years						

Table 5.1: Time comlexity examples [inspired 8]

Both time and space criteria, usually directed against themselves, because it is not possible to simultaneously optimize memory and time, so the programmer must always choose what he prefers. Nowadays usually time is preferred. Algorithms goal is to achieve smallest asymptotic time complexity for huge size of n.

Adaptability: This is the ability of the algorithm to take an already partially-sorted array and be able to continue where it left off and finish sooner than if it it wasn't pre-sorted. Reason is to save running time. Algorithms that take this into account are known to be adaptive.

5.2 Sorting algorithm's description

Bubble sort, Insertion sort, Merge sort and Quicksort description are brief, while these algorithm's are included in the university studies or are very common and it is easy to search detailed information about them. Simo sort, Gnome sort, Strand sort, Comb sort, Timsort are covered in more detail.

5.2.1 Bubble sort

It is very often the first sort people learn, because it is so simple. Bubble sort keeps passing through the array, exchanging adjacent elements that are out of order, until no swaps are required. At this point the the array is sorted. The name „bubble“ come from the fact that small elements „bubble“ to the front of the array. It means it’s sorting method is exchange. Bubble sort’s prime virtue is simple implementation and tiny code. Of course it’s arguable whether it is easier to implement than insertion sort or selection sort and yet it is slower.

In the best case we can achieve O (n). The average and worst case performance can rise to O (n^2). Worst space complexity is O (1).

5.2.2 Insertion sort

The method that people often use to sort bridge hands is to consider the elements one at a time, inserting each into it’s proper place among those already sorted. In a computer implementation, we need to make space for the element being inserted by moving larger elements one position to the right, and then inserting the element into the prepared position. Sorting is typically done in-place, thus O (1) additional space is required. Another advantages are that it is adaptive (efficient for already sorted data sets), stable and comes with simple implementation.

The insertion is very effective on already or nearly sorted arrays even for array size up to 100 000. The constrain for practical application is array with few elements. Practical research shows a optimal result less then 15 elements, for all tested array types. The time complexity can come very close to theoretical best case performance O (n). For larger data sets time complexity increases quickly to the average and worst case O (n^2).

5.2.3 Binary insertion sort

Is one of the variants of insertion sort. As the name implies it uses binary search to find the appropriate location to insert the new element to the output [20].

Therefore it performs O (log n) comparisons in the worst case, which in final is O (n log n). The running time as whole still has O (n^2), this is due the series of swaps required for each insertion.

5.2.4 Selection sort

Belongs among the simplest sorting algorithm’s. It works as follows. First, find the smallest element in the array , and exchange it with the element in the first position. Then, find the second smallest element and exchange it with the element in the second position. Continue in this way until the entire array is sorted.

From this simple method we get poor performance of O (n^2) for best, average and worst case. O (1) constant memory complexity is useless [19].

5.2.5 Double burst selection sort (DBSS)

The „burst-selection sort“ idea is an improvement to normal selection sort in a way that after each pass all the smallest elements end up at the beginning of the array. This approach is the same time applicable for the biggest elements of the array, therefore we got a enhanced selection sort variant Double burst selection sort.

We can consider this algorithm as a direct competitor to Insertion sort. If we look at the the time and space complexity we are getting the same values as selection sort. The algorithm should perform better then Insertion sort for lot of equal elements in array, unfortunately for array filled with two equal elements didn’t showed it’s advantage. The array has to have significantly more equal elements so the Double burst selection sort can outperform Insertion sort. The results of the tests showed single case of defeating Insertion sort and that was for reverse array. However, as an inside algorithm in Quicksort it performs solidly and gives better results as Simo sort, which uses Insertion sort array of smaller size.

5.2.6 Quicksort

Quicksort is very popular sorting algorithm. It's easy to implement, works well for a variety of different kinds of input data achieving this with fever resources then other sorting algorithms in many situations. Quicksort is a representative of divide and conquer method for sorting. It works by partitioning an array into two parts, then sorting the parts independently. This can be done efficiently in linear time and in-place. The main steps for the algorithm are:

1. Set a element called pivot. Correctly picking pivot is crucial for performance. Unsuitable pick of pivot value can highly degrade sort performance and vice versa. Usually there are four ways to determine pivot value. Firstly is randomly choosing from array. Secondly calculating is the middle as: (first - last)/2 + last. Thirdly, we take first, middle, last and we determine the median. First, middle and last are indexes if given array and are self-explanatory. Fourthly, calculating average from array elements.

2. Reorder the array in the way that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position.

3. Recursive call to sort the both sub-lists is performed.

Normally 2-way partitioning is used. A variation 3-way partitioning is introduced. Has a slightly has slightly higher overhead compared to the standard 2-way partition version. Both have the same best, typical, and worst case time bounds, but this version is highly adaptive in the very common case of sorting with few unique keys. When stability is not required, 3-way partition Quicksort is the general purpose sorting algorithm of choice [18]. Suggested optimizations for Quicksort is to sort smaller arrays using other sorting algorithm's for example Insertions sort.

Quicksort makes O (n log n) comparisons to sort n items. In the worst case, it makes O (n^2) comparisons, but this behaviour is unlikely. Typically it is just O (n log n) for average and best case. If in-place partitioning is used it requires O (1) space, but on the other hand it's unstable. After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most O (log n) space [19] [25].

5.2.7 Quicksort with random pivot

Standard Quicksort with 2-way partitioning where pivot value is chosen randomly. As mentioned random pivot can highly degrade in normal case very good Quicksort performance. And it does. The practical test showed very poor performance compared to other implementations or let's say variations of Quicksort.

5.2.8 Quicksort with median pivot and DBSS

Again we have a 2-way Quicksort, but this time the pivot value is determined as median from first, middle and last element of array. Further it implements DBSS for fast sorting small arrays, therefore it skips partitioning partitions with less than 30 items and runs DBSS to each of these partitions. This combination brought superb practical results. It finished in the top three fastest sorting algorithms tested.

5.2.9 Simo sort

Is another member in the Quicksort family. Brings optimization for Quicksort that applies 2-way partitioning combined with Insertion sort. This recursive sorting algorithm repeats the following tests:

1. Get the average value of numbers in the array.
2. Divide the array into two arrays (array 1 and array 2) and the average value will be chosen as the pivot.
3. Any value smaller than the pivot will be in first array and any value larger than the pivot will be in the second array.
4. Repeat the same algorithm on array 1 and array 2 till you reach the ending condition.

Optimization was done in two cases. Firstly, stop conditions. Which are not normal recursive ending conditions, but nevertheless raise performance a lot. These conditions were obtained under try and error testing. Secondly, entering Insertion sort part with condition that the number of elements in array is less then 16 [5].

Example
Normal case situation of algorithm functionality is showed in Figure 5.1.

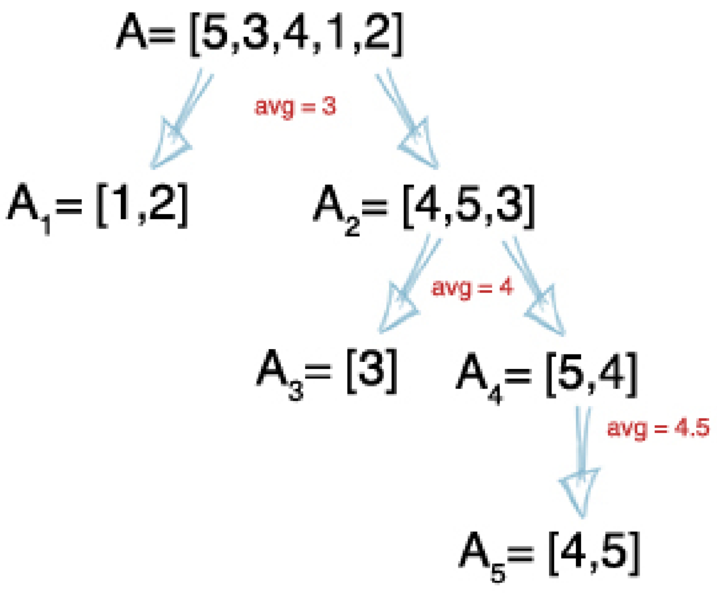


Figure 5.1: Simo sort sorting example 1 [copy from 5]

Figure 5.2 shows how the algorithm works in its best case scenario where elements of the array have little variance between them. In the case where there where only 2 numbers „1“ and „0“ the algorithm had a complexity of O (n).

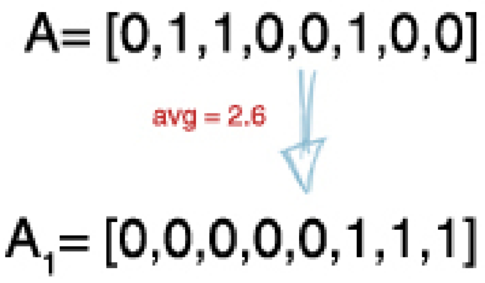


Figure 5.2: Simo sort sorting example 2 [copy from 5]

When the algorithm stops in the shown figures it means a leaf has reached an ending condition.

Complexity
The search is Stable and has an upper bound space Complexity of O (1). After doing the time complexity analysis calculations. The average and worst case scenarios are of (5/6)*(n log n) complexity which maps to O (n log n) The best Case is O (n) as shown in Figure 5.2.

Practical results
Practical tests showed excellent results independently on the array type, which we will evaluate later. In order of Quicksorts is ranked second, in global testing is representing 4th fastest sorting algorithm independently on array sizes.

5.2.10 Mergesort

Merge sort is a comparison-based sorting algorithm based on a straightforward divide-and-conquer algorithm. Merge sort works recursively by first breaking the input array into half's until the recursion process stops when array size is equal to one, in each recursive level. Finally passing the resulting sorted array to the levels above to combine the ordered parts to make the whole ordered file. Performing the merge algorithm on two arrays with one element in each is trivial. There are two well known implementations, top-down and bottom-up implementation. Further it is a stable sort, and this feature tips the balance in its favour for applications where stability is important. Another future of Mergesort that is important in certain situations where is normally implemented such that it accesses the data primarily sequentially. For example is a choice for sorting a linked list, where sequential access is the only kind of access available [19].

Mergesort comes with different variants. One is called Hybrid mergesort, use another sorting algorithm to sort relatively small sub-lists. Afterwards repeatedly merge sub-list to produce new sub-list untill only one list remains. Natural mergesort is other variant, which works very similarly to Bottom-up mergesort with addition that any naturally occurring runs in the input are exploited. From practical side this proves very useful while random input usually has many short runs that just happen to be sorted [23].

Complexity
One of Mergesort's most attractive properties is that it sorts file of 'n' elements in time proportional to O (n log n), independently of input for all cases. Worst case space complexity is O (n) auxiliary.

5.2.11 Comb sort

Is another comparison and exchange sort which builds on the idea of the Bubble sort. Was developed as an improvement to to Bubble sort and Quicksort using the idea of eliminating turtles or small values near the end of array. Big values at beginning of array called rabbits do not pose a performance problem for Bubble sort.
Always adjacent elements are compared in in each data set in case of Bubble sort. The Comb Sort improves on this by adding a gap which allows non-adjacent numbers to be compared. After each iteration, the gap is reduced by a shrink factor until it reaches the value of 1. Dividing the gap by $(1 - \exp^{-\varphi})^{-1} \approx 1.247330950103979$ works best, but 1.3 may be more practical. Hence, at the very end, the Comb Sort behaves exactly like the Bubble Sort. Some implementations use the insertion sort once the gap is less than a certain amount.

The algorithm
Each iteration of the algorithm consists of three stages:

1. Calculation of the gap value based on the number of elements to be sorted.

2. Iterating over the data set comparing each item with the element that is “gap” elements further down the list and swapping them if required.

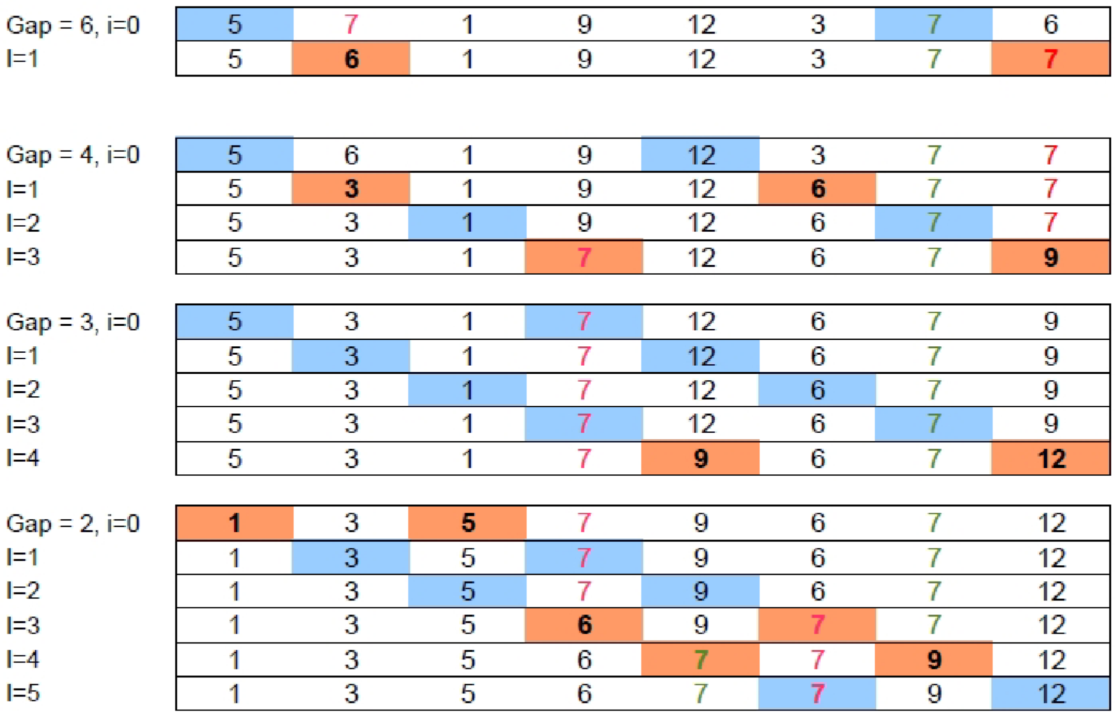
3. Checking to see if the gap value has reached one and no swaps have occurred. If so, then the set has been sorted.

1. Calculating the “gap”
The gap, which is essentially an offset used when comparing elements, is something that changes for each iteration. The value needs to change in a uniform manner in such a way as to provide optimum comparisons during the sorting phases. As mentioned the shrinking factor 1.247330950103979 works best, but 1.3 can be used.

2. Iterating and swapping
This phase is the same as the Bubble Sort. The only difference in the Comb Sort is that the elements which are compared are i and i + gap as opposed to i and i + 1. For each iteration, we start at the beginning of the data set and loop until i + gap is greater-than or equal to the size of the data . For each sub-iteration in this loop, we compare i and i + gap and swap the values if required. At this point the gap is recalculated and we go again.

3. Terminating the loop
When the gap value reaches one, we know that we’re now behaving the same as the Bubble Sort algorithm (since we’re comparing i and i + 1), so we know that if we don’t perform any swaps during the iteration then the set must be sorted.

Example



Gap = 6, i=0	5	7	1	9	12	3	7	6
I=1	5	6	1	9	12	3	7	7
Gap = 4, i=0	5	6	1	9	12	3	7	7
I=1	5	3	1	9	12	6	7	7
I=2	5	3	1	9	12	6	7	7
I=3	5	3	1	7	12	6	7	9
Gap = 3, i=0	5	3	1	7	12	6	7	9
I=1	5	3	1	7	12	6	7	9
I=2	5	3	1	7	12	6	7	9
I=3	5	3	1	7	12	6	7	9
I=4	5	3	1	7	9	6	7	12
Gap = 2, i=0	1	3	5	7	9	6	7	12
I=1	1	3	5	7	9	6	7	12
I=2	1	3	5	7	9	6	7	12
I=3	1	3	5	6	9	7	7	12
I=4	1	3	5	6	7	7	9	12
I=5	1	3	5	6	7	7	9	12

Figure 5.3: Comb sort sorting example

Last pass of sorting for gap value 1 is not showed. No swaps will be performed and the algorithm finishes. We can follow the value 7 to verify that it is not stable sorting method. The blue column represents comparison. The orange column comparison with swap. Further on the Figure 5.3, gap size reduction can be recognizably observed. Two occurrences of number 7, with two different colours (green, red) reveal that this method is not a stable method, because the order of same keys changes according to number of swaps required. The stability is not maintained.

Complexity

It is quite interesting, despite the fact that is based on the principle of Bubble sort the average time complexity is just O (n log n). The worst case situation can reach O (n^2) however the best case situation is O (n). No auxiliary array is needed during sorting, out of which we get O (1) worst case space complexity.

Practical results

Compared to Bubble sort it has major speed enhancement. Which is more and more noticeable with growing array size from 100 elements. Especially sorting random array. However the algorithm does not provide sufficient speed up to cope with the fastest algorithm's in test, like : Quicksort with median pivot and DBSS or Timsort. Therefore I wouldn't recommend it for practical application due to weak performance.

5.2.12 Gnome sort

Or sometimes referred to as Stupid sort. Is a an extremely simple sorting algorithm which is similar to Insertion sort. The difference is in moving elements, which is accomplished by a series of swaps as in Bubble sort.

The algorithm
Algorithm start at the beginning of array. Two adjacent elements are compared, if they are in the incorrect order, swap is proceeded. As in Insertion sort, it only keeps track of the current element. After moving the element backward into place, it „walks“ forward, checking the order of items as it goes until it reaches the next out-of-place element. Which is where progress has left off.

Complexity
Worst and average case complexity is O (n^2) for arbitrary data, but approaches O (n) if the input list is nearly in order or in order [14].

Practical results
Strong side of the algorithm is when used on sorted or nearly sorted arrays. In these cases it can keep up with the fastest algorithms until array size 100 000. Sorting the other types of arrays resulted in very poor performance and it is not recommend to use at all. It is not optimal for large data sets and absolutely not for reverse arrays, the same as Bubble sort. Under real-world conditions Insertion sort or Double burst selection sort are much more suitable options for small data sets.

5.2.13 Timsort

Timsort is a hybrid sorting algorithm. This means that while the two underlying sorts it uses (Mergesort and Insertion sort) are both worse than Quicksort for many kinds of data, Timsort only uses them when it is advantageous to do so. The design of Timsort was aimed to perform well on many kinds of real-world data achieving it with adaptability while maintaining stability. It was invented by Tim Peters in 2002 for use in the Python programming language. Timsort has been Python's standard sorting algorithm since version 2.3. Nowadays it is also used to sort arrays in Java SE 7 and on the Android platform.

The algorithm
The main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs „intelligently“. Everything else is complication for speed, and some hard-won measure of memory efficiency [17].
The advantage of Timsort is that it takes inconsideration partial orderings that already exist in most real-world data. Timsort operates by finding runs, subsets of at least two elements, in the data. Runs can be either non-descending (each element is equal to or greater than its predecessor) or strictly descending (each element is lower than its predecessor). If it is descending, it must be strictly descending, since descending runs are later reversed by a simple swap of elements from both ends converging in the middle [27]. Use of strict descending definition ensures that a descending run contains distinct elements.

Minrun - A natural run represents a sub-array that is already ordered. Different size of natural run leads to using a (Binary) Insertion sort for sizes smaller than certain number. The size of the run is checked against the minimum run size. The minimum run size (minrun) depends on the size of the array. We pick a minrun in range(32, 65) such that N/minrun is exactly a power of 2, or if that isn't possible, is close to, but strictly less than a power of 2 then the final algorithm for this simply takes the six most significant bits of the size of the array, adds one if any of the remaining bits are set, and uses that result as the minrun. This algorithm works for all cases, including the one in which the size of the array is smaller than 64 [17] [27].

Insertion sort - When an array is random, it is very unlikely long runs will occur. If a natural run contains less than minrun elements stable Binary insertion sort is used to increase the size of run to minrun. Thus, most runs in a random array are, or become, minrun in length. This results in balanced merges, which provides an efficient way to proceed. It also results in a reasonable number of function calls in the implementation of the sort [17] [27].

Merge pattern - After run length optimisation, runs are merged. The merged is done by a specific technique that will ensure the highest efficiency. When a run is found, the algorithm pushes its base address and length on a stack. A function is then called which determines whether the run should be merged with previous runs. Timsort does not merge non-consecutive runs because doing this would cause the element common to all three runs to become out of order with respect to the middle run [27]. Therefore, merging is always done on two consecutive runs. For this, the three top-most unsorted runs in the stack are considered. Lets say, A, B, C represent the lengths of the three uppermost runs in the stack, the algorithm merges the runs so that ultimately the following two rules are satisfied:

1. $A > B + C$
2. $B > C$

For instance, if the first of the two rules is not satisfied by the current run status, that is, if $A < B + C$, then, Y is merged with the smaller of A and C. The merging continues until both the rules are satisfied. Then the algorithm goes on to determine the next run [17].

The thrust of these rules when they trigger merging is to balance the run lengths as closely as possible, while keeping a low bound on the number of runs we have to remember. This is maximally effective for random data, where all runs are likely to be of length minrun, and then we get a sequence of perfectly balanced merges. The algorithm also tries to exploit the fresh occurrence of the runs to be merged, in cache memory. Thus a compromise is attained between delaying merging, and exploiting fresh occurrence in cache memory [17] [27].

Merging memory - Temporary memory is used for merging adjacent runs. The temporary memory has to be the minimum size of length(run A) + length(run B). The algorithm copies the smaller of two runs into the temporary memory and then uses this freed up memory of the smaller run to store the other run after sorting.
When we reach the point where we want to merge adjacent runs A and B. The fist step we take is we do a binary search, to find B[0] position in A. Elements in A preceding that point are already in their final positions, effectively shrinking the size of A. Similarly we also search A[-1] position in B and elements of B after that point can also be ignored. This cuts the amount of temp memory needed by the same amount [27].
Merge algorithms - merge_lo and merge_hi functions or methods are where the majority of the time is spent. The merge_lo deals with runs where A<= B, and merge_hi where A >

B. Merging starts comparing the first element of A to the first of B, and moving B[0] to the merge area if it is less than A[0], else moving A[0] to the merge area. This process is called öne pair at a time: During that we keep track of how many times in a row the winnercomes from the same run. If that count reaches MIN_GALLOP, we switch to galloping mode: At this point we find B, where A[0] belongs, and move over all the B's before that point in one chunk to the merge area, afterwards move A[0] to the merge area. Then we search A for where B[0] belongs, and similarly move a slice of A in one chunk. Then back to searching B for where A[0] belongs, etc. We stay in galloping mode until both searches find slices to copy less than MIN_GALLOP elements long, at which point we go back to one-pair-at-a-time mode [17]. Detail description can be found in python's list sort description [17], which is Timsort.

Example
The following Figure 5.4 demonstrates the work of Timsort algorithm in a more coherent way. It is a visualisation of sorting a shuffled array of 64 elements:

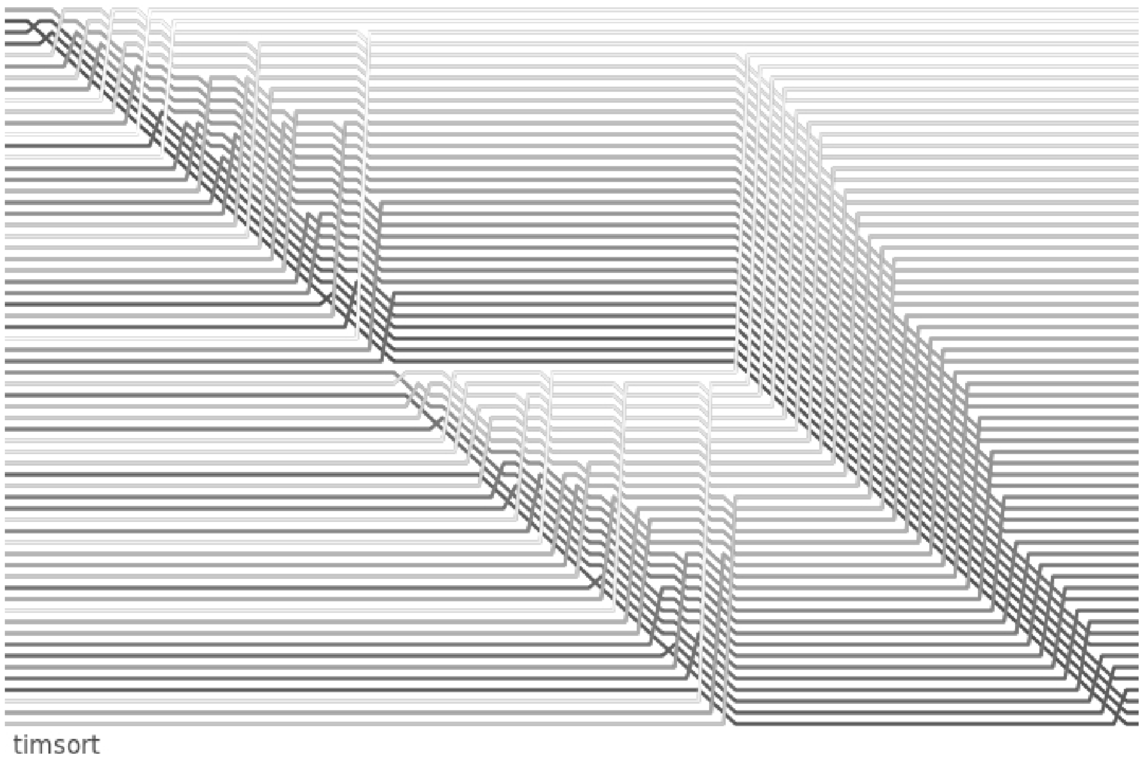


Figure 5.4: Timsort sorting visualisation 1 [copy from 2]

It is immediately distinguishable that the data is divided into two blocks of 32 elements. The blocks are pre-sorted in turn (the first two „triangles“ of activity, reading from left to right), before being merged together in the final step (the cross-hatch pattern at the right of the diagram). If we take a closer look, it's even possible to tell that the pre-sorting seems to be using Insertion sort - the triangular pattern here corresponds to the triangular pattern of Insertion sort run through the same data [2].
The next Figure 5.5 visualisation allows us more detail in sorting a partially sorted array.

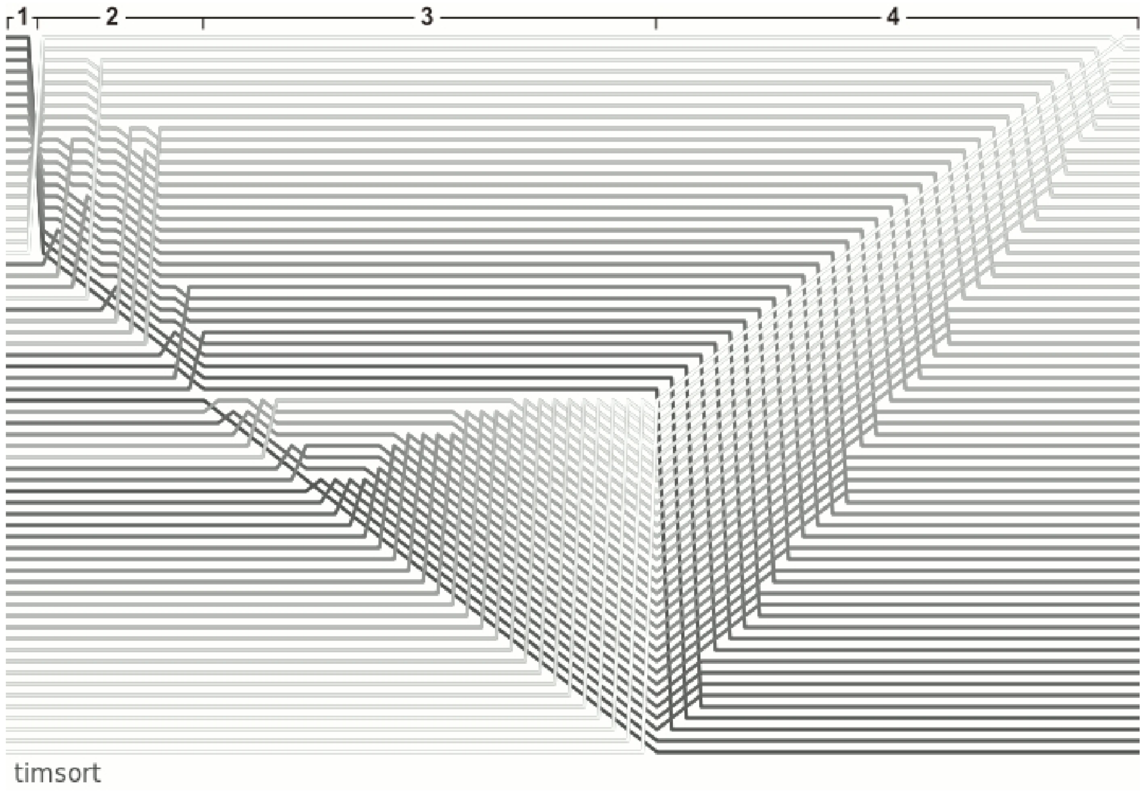


Figure 5.5: Timsort sorting visualisation 2 [copy from [2]]

Let us follow the marked progression from left to right:

1. Firstly, Timsort finds a descending run, and reverses the run in-place. This is done directly on the array of pointers, so seems „instant“ from our vantage point.
2. Secondly, the run is now boosted to length minrun using Insertion sort.
3. Thirdly,if no run is detected at the beginning of the next block, and insertion sort is used to sort the entire block. Note that the sorted elements at the bottom of this block are not treated specially - Timsort doesn't detect runs that start in the middle of blocks being boosted to minrun.
4. Finally, Mergesort is used to merge the runs [2].

Complexity
Building on the knowledge that no comparison sort can perform better than O (n log n) comparisons in average case. Timsorts enormous benefits come from taking advantage of the fact that the sub-lists of the data may already be sorted brings us to the situation where it often requires fewer than O (n log n) comparisons for real-world data sets. It comes close to O (n) for best case. Random data sets does not allow us to take advantage of partially ordered sub-arrays. In this case, the algorithm approaches the theoretical limit of log (n!), which is O (n log n) in worst case. The considerable disadvantage of Timsort is space complexity of O (n).

Practical results
Strengthened the assumption for great results for real-world data sets. For arrays until 100 elements it achieved very good average results. In arrays of more than 100 elements the sophisticated implementation of algorithm is fully reflected in the excellent results. It even outperformed the build in and fully optimized C# list Sort() sorting algorithm for all tested array types ()except for the case of sorting array of random numbers. We can consider Timsort the best sorting algorithm for sorting arrays of array size bigger than 100 elements. But we must not forget about it's space complexity O (n), while it failed to sort the array of size 10 000 000 elements, reporting out of memory exception. If it is possible to work around space complexity constrain, afterwards it becomes the best algorithm to go for. In case of too many random elements, it is advisable to sort it in first pass with optimal Quicksort or build in Sort method, and then use Timsort to sort the array regularly.

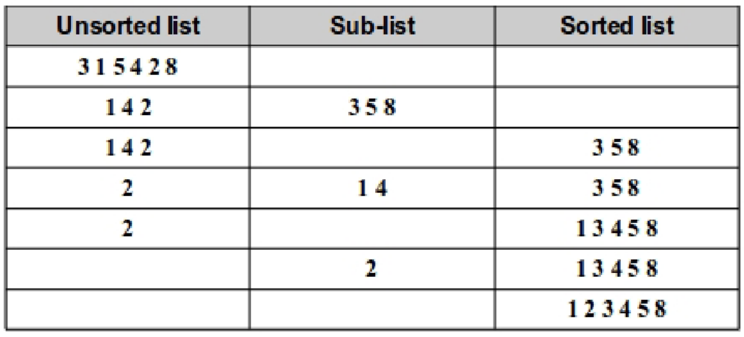
5.2.14 Strand sort

Belongs to the group of merge sorts. The strength of the algorithms is that it works well if frequent occurrence of ordered elements is located in array. Stability is maintained during sorting.

The algorithm
Firstly, we begin by putting the first element from the original, unsorted list to the sub-list. For each subsequent element in the original list. If it is greater than the last element of the sub-list, remove it from the original list and append it to the sub-list. The next step is followed by merge of the sub-list into a final, sorted list. Repeatedly extract and merge sub-lists until all elements are sorted [14].

1. Parse the unsorted list once, taking out any ascending (sorted) numbers.
2. The (sorted) sub-list is, for the first iteration, pushed onto the empty sorted list.
3. Parse the unsorted list again, again taking out relatively sorted numbers.
4. Since the sorted list is now populated, merge the sub-list into the sorted list.
5. Repeat steps 3–4 until both the unsorted list and sub-list are empty.

Example



Unsorted list	Sub-list	Sorted list
3 1 5 4 2 8		
1 4 2	3 5 8	
1 4 2		3 5 8
2	1 4	3 5 8
2		1 3 4 5 8
	2	1 3 4 5 8
		1 2 3 4 5 8

Figure 5.6: Strand sort sorting example [inspired 26]

Complexity
The average and worst case goes with O (n^2). The possibly worst situation occur when sorting reverse order array. The best case situation arises when array is already sorted. Then the algorithm is linear. The space complexity is O (1) auxiliary elements.

Practical results
It is known to be most useful for data sets which are stored in a linked list. Strand sort is most useful for data which is stored in a linked list, due to the frequent insertions and removals of data. Using another data structure, such as an array, would greatly increase the running time and complexity of the algorithm due to lengthy insertions and deletions. Strand sort is also useful for data which already has large amounts of sorted data, because such data can be removed in a single strand.

5.3 Test methodology

Gives insight on how where the tests proceeded. With what kind of data structures, input data, array value order and time measurement techniques were used.

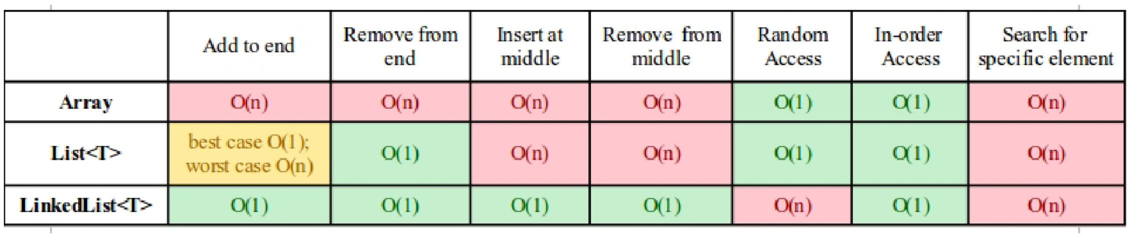
5.3.1 Data type and structure

If we consider aggregation, we know sorting will be a mandatory and most important part of aggregator. In first step we have to consider input data type, value and amount in certain time. It means a data structure includes floats and strings. The value of Bid and Ask is stored in floating point type double. Double was chosen based on the knowledge that PIP (smallest measure for price move) has two possibilities depending on currency pair, pip=0.01 and pip=0.0001. Double number representation has precision 15-16 digits. For testing I was considering three data types: float, double, decimal. They differ in precision, internal representation and speed. Now, for our purpose floating point representation float should be sufficient. But for future improvements, which can include more precise data representation, testing was performed on double data type. The precision of mentioned data types are:

- Float - 7 digits (32 bit)
- Double - 15-16 digits (64 bit)
- Decimal - 28-29 significant digits (128 bit)

Float and double represent floating binary point types unlike decimal, which is a floating decimal point type. The question can occur, why is not decimal a better choice, if it is more accurate. That is true, yet performance can make a big difference. As we expected binary number representation is much quicker processable by processors. According to [3], „decimal types ran over 20 times slower than their float counterparts“. For financial application as the aggregator performance and accuracy is a big issue.

In next step, we need to take inconsideration the data structure used for storage. For our purpose three data structures come in account: Array, List, LinkedList. We will examine their properties and pick one for testing [12] [10].



	Add to end	Remove from end	Insert at middle	Remove from middle	Random Access	In-order Access	Search for specific element
Array	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n)
List<T>	best case O(1); worst case O(n)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
LinkedList<T>	O(1)	O(1)	O(1)	O(1)	O(n)	O(1)	O(n)

Figure 5.7: Data structures properties

List was chosen as most suitable one due to random access with O (1) value. For testing on many algorithms is suitable to choose general data structure and clearly List has better properties then Array. LinkedList has best properties but is not suitable to every tested sorting algorithm. For specific algorithm implementation data structure type need to be reconsidered according to algorithms, operations and frequency of operations applied on data structure. For most cases LinkedList will be the best option to select.

5.3.2 Input data

Pseudo increasing market data
In the first case I generated constantly growing input data. The init value was 1.1111. For every following array number 0.4160 was added. The purpose of the test was to test the sorting algorithms on constantly growing input data and later compare it with pseudo generated market data and real market data. The values are generated by calling RunSortingTests.PseudoIncreasing() metod. Example values:
[1.1111, 1.5271, 1.9431, 2.3591, 2.7751, 3.1911, 3.6071, 4.0231, 4.4391, 4.8551]

Pseudo market data
To simulate real market data, two methods were created, RunSortingTests.PseudoMarket2decimalPIP() and RunSortingTests.PseudoMarket5decimalPIP(). The goal of these methods is to generate data similar to those of the market, i.e. pseudo market data. Therefore pip=0.01 2decimal and pip=0.00001 5 decimal values distribution. For 2 decimal values we achieve this by starting with initial number 1.11 and adding or subtracting randomly generated numbers from range: 0.00, 0.01, 0.02, 0.03, 0.04, 0.05, where number can not decrease under value 0. From real market data observation it is know that next value will fall into the range of 0.00,...,0.05. For 5 decimal similar procedure is applied. We start with initial number 1.1111 and again adding or subtracting randomly generated numbers from range: 0.00001, 0.00002, 0.00003, 0.00004, 0.00005, 0.00006, where initial number bottom value is restricted to 0. The example generated values fro both methods are:
[1.02, 1.02, 1.05, 1.06, 1.07, 1.09, 1.09, 1.11, 1.11, 1.14]

[1.1111, 1.11116, 1.11119, 1.1112, 1.1112, 1.1112, 1.11121, 1.11121, 1.11123, 1.11124]

Real market data
Finally, in the fourth case, real-world market EUR/USD currency pair data from one day trading was used to fill the tested list arrays. First ten values of Ask are:

[1.22833, 1.2283, 1.22828, 1.22824, 1.22824, 1.22818, 1.22817, 1.22817, 1.22815, 1.22816]

5.3.3 Array value order

Before sorting on array can start, we need to prepare array value distribution. Five types of arrays suits are needs perfectly. The following points shows array value order and their corresponding initializing method:

- Sorted - PrepareSortedList()
- Random - PrepareRandomList()
- Nearly sorted - PrepareNearlySortedList()
- Few unique - PrepareFewUniqueList()
- Reverse - PrepareReverseList()

Sorted, random and reverse are self-explanatory. For nearly sorted array, every two neighbours are switched. For example, if we have array [A, B, C, D], we get [B, A, D, C]. In the case of few unique array the first half of the array is initiated and then copied to the second part of array. The reason is to get duplicate element values. For example, the array [A, B] is then duplicated to get [A, B, A, B] etc. In one test, sorting algorithm sorts all of mentioned arrays.

5.3.4 Time measurement

Crucial part of testing was the time measurement of sorting algorithms sorting data for different array value distribution. To measure time accurately I considered C# Stopwatch Class as most suitable. To achieve very accurate results, ticks (ElapsedTicks) and millisecond (ElapsedMiliseconds) are used as units of measurement of result. Tick is the smallest unit of time that the Stopwatch timer can measure. In first step, array value distribution is proceeded. Consequently timer starts to count, followed by sorting with a specific sorting algorithm. After sorting is finished, timer is stopped and result are written to file and terminal output.

5.4 Test results

All advises and conclusion are based on the results obtained respecting the constrains listed in Chapter 5.3 Test methodology. There are four possibilities to generate input data. With list count (array size): 10, 100, 1000, 10 000, 100 000 for all types generated market data and 1000 000, 5000 000 and 10 000 000 also for pseudo increasing market data. With five array types with different value orders. For all list count the test were done five times. The fist measurement was always much slower, this was probably due it was not cached in the memory. The rest four measurements show very similar results. At the end I counted the average from four results per array type, to get more corresponding result without fluctuations.

List count 10
Generally for sorting 10 elements a lot of sorts are suitable, but if our concern is fastest sorting time then our choice is Insertion sort. For my surprise Insertion sort was often a little faster than Binary insertion sort. Quality result were also brought by Double burst selection sort and Quicksort with median pivot and DBSS, which implements the DBSS

for smaller array sizes. Simo sort hold to the top with internal implementation of Insertion sort. Results are clearly visible in Figure 9.1 in Appendix 1.

List count 20
For 20 elements in the list we still consider Insertion sort the fastest choice. We can observe that Simo sort again, as in a 10 element list is doing very good, is holding second position after Insertion sort. This is the result of the build in Insertion sort, which starts to work for list of 15 and fever elements.

List count 50
Surprisingly Insertion sort is still doing a very good job for sorted and nearly sorted list, but it is already loosing performance . Quicksort with median pivot and DBSS is the fastest for 50 real-world market data elements. Closely followed by Simo sort. The values are still measured in ticks. Results are interpretable in Figure Figure 9.2 in Appendix 1.

List count 1000
The values are measured in milliseconds. The fastest sorts can achieve time under 1 ms. These are Quicksort with median pivot and DBSS, Simo sort, Timsort and C# list Sort. 1000 elements is the point where the gap between the fastest and rest of tested sorts start to grow rapidly.

List count 10 000, 100 000, 1000 000
The sorting power if Timsort overtakes and bring best performance. Except for the case with random data order. The only sort coming close is C# list Sort. Not surprisingly, Quicksort with median pivot and DBSS and Simo sort are holding their level also. The result can be viewed in Figure 9.3 in Appendix 1.
Unfortunately, these results have just informative purpose. The aggregator will not sort list with these count sizes.

All the test were performed on laptop Lenovo L412, CORE i5 CPU, M520 2.4GHz, 2x2GB RAM 1333MHz, Windows 7 64bit with Service Pack 1.

5.5 Conclusion

Overall best result were achieved when sorts had to sort pseudo increasing market data. This vector contained increasing unique numbers. C# internal comparison could handle them faster. Further, excellent results were obtained sorting real-world market data. Where results were quite close to the result from pseudo increasing market data. This was caused by lot of elements with equal values. Their occurrence was appropriately reflected in performance due to lower number of swaps and lower comparison delay. In general it means 5 to 20% better or worse performance, depending on the sorting algorithm internal algorithm's. The ones which are optimized or their algorithms are suitable to sort same elements, have done a better job. The others were penalized by worse performance.

In general sorting pseudo market data with both 2 and 5 decimal pip values was slower compared to two other input data values. It is a result of number distribution. Where lot of same value elements and random value distribution occur. This leads to a little longer comparison time. Also the number of decimals has impact on comparison time. Although we cannot state, that the fewer the decimals the faster the sorting. On the other hand, we can state that the value distribution has an impact on sorting speed. It mostly depends if the sorting algorithm can take advantage of it's internal algorithm techniques. For instance, occurrence of ascending or descending numbers in array. Nevertheless, these facts have not changed the final order of the sorting algorithms according to their speed.

The built in C# list Sort method handled all list counts very good. In the most list counts it belong to the first three fastest independently on array value order. In general it is the easiest and most reliable option considering sorting. For special purposes, where faster sorting is required I suggest using Insertion sort for list count from to 15, maximum 20 elements. If for a reason alternative is required, Binary insertion sort and Double burst selection sort. DBSS is more appropriate for sorting currency pair values (Bid/Ask), where lot of same values occur.
After 20 elements random and reverse value orders decrease overall performance significantly. For list count above 20 elements Quicksort with median pivot and DBSS proved optimal. Simo sort is slightly behind. Both handle all five array value orders superbly. Combination Quicksort with DBSS or Insertion sort and correctly determining pivot value proved as extremely efficient.
I have included Bubble sort for two particular reasons. Firstly, because is mostly the first sorting algorithm students and programmers get in touch, to study the basic principle of sorting. The second reason was to actually see and prove on results that is suitable for learning purposes and definitely not for practical use on real-world data sets.
Timsort's sophisticated algorithms were also tested on real-world data sets. And not just on them. The bigger the list count the better the performance. Unfortunately we cannot forget about memory requirements. As I acknowledged, it was the only sorting algorithm that I got out of memory exception for list count 10 000 000. Timsort is really appropriate for sorting real-world data sets where elements cannot be in random order. If they are, the performance is poor compared to C# list Sort, which was the only sorting algorithm providing sort result coming near to Simsort's.
Mergesort performs average for all array value orders and in every case was behind the leaders like Quicksort with median pivot and DBSS, Simo sort, Timsort and C# list Sort. Smothsort an improved version of Mergesort promises excellent practical results considering theoretical time and space complexities with space complexity O (1) compared to Timsort's O (n). Unfortunately my implementation of Smoothsort, which was re-edited from C++ implementation, brought very poor results. This was caused probably due to incorrect implementation, which I didn't have time to repair.
The other sorts, Comb sort, Strand sort, Gnome sort did not bring any positive surprise results. Although I got a negative surprise result from Quicksort with random pivot, which clearly is not suitable sorting currency pair Ask or Bid values. This is because choosing a random pivot in a very restricted value range proves as totally ineffective.

In final, from the test results I suggest the following sorting algorithms for sorting currency pair values Ask or Bid:

- Until list count 20:
    1. Insertion sort
    2. Binary insertion sort
    3. Double burst selection sort
- From 20 to 10 000:
    1. Quicksort with DBSS

    2. Simo sort
    3. C# list Sort
    4. Timsort

- From 10 000 elements:
    1. Timsort
    2. C# list Sort

Based on the results of testing, Insertion sort and Quicksort with DBSS were selected for aggregator implementation.

Chapter 6

Aggregator design

High-frequency trading is highly demanding trading concerning nowadays financial market speed requirements to successfully capture execution opportunities 24x7. The most important features required for creating a multiple liquidity software aggregator are minimum application latency, stability and reliability with secure transfer of market data. Market type in which we wish to trade is crucial for designing the appropriate aggregator. The following Chapter brings insight to all necessary parts of this design process and aggregator application specification.

6.1 Implementation Environment

For development of such an applications a robust development environment with appropriate tools is necessary. Because the chosen development language was C#, Visual Studio appeared as the best option, while it has native C# support. Visual Studio provides a robust IDE and with the .NET Framework helps to build, debug and maintain applications more effectively, reducing development time.
The .NET Framework is an application development platform that provides services for building, deploying, and running desktop, web, and phone applications and web services. It consists of two major components: the common language runtime (CLR), which provides memory management and other system services, and an extensive class library, which includes tested, reusable code for all major areas of application development [11]. The .NET Framework's Base Class Library provides user interface, network communication, database connectivity, data access, cryptography, web application development, etc. Other frequently used .NET Framework components:

- **ASP.NET** - technology for creating web applications
- **Windows Communications Foundation (WCF)** - technology for creating web services and communication infrastructure of applications
- **Windows Workflow Foundation (WF)** - technology for defining heterogeneous sequence processes
- **Windows Presentation Foundation (WPF)** - technology for creating visually appealing user interfaces
- **Windows CardSpace** - implementation of Information Cards standard

- **LINQ** - Language Integrated Query, object access to data in database, XML and objects, which implement the IEnumerable interface

I chose Visual Studio 2010, the latest version of Microsoft's development environment. It comes with new features, as better code intellisense, improved multi-targeting support, improved code collapse, variable highlighting and multiple monitor support. The features directly improve production.

Further, it has very good integration with MS SQL and is highly optimized. Another reason was that the Microsoft enables to download student licensed version of his products from MSDN. I used this opportunity and downloaded MS SQL Server 2008 R2 (64bit) Express and Visual Studio 2010 Professional.

6.2 Design specification

Our focus is on FX market. It means, the aggregator has to be capable of sorting currency pair values (Bid/Ask) as fast as possible. The liquidity is provided by three bank institutions: Deutsche Bank, UBS and Morgan Stanley. The application must qualify for the secure transfer of data. The messages pending from client to server are processed with the help of QuickFIX/n API. This API is specified to use with .NET technology. The sorted currency values are easy viewable in one window. The application window is highly customizable to suit the needs of trader. Further it allows managing input/output feeds online. Visual Studio 2010 provides framework, which is a suitable working environment to build Windows Form application programs. The aggregator application diagram gives a simple image of data inputs/outputs and implementation technologies.

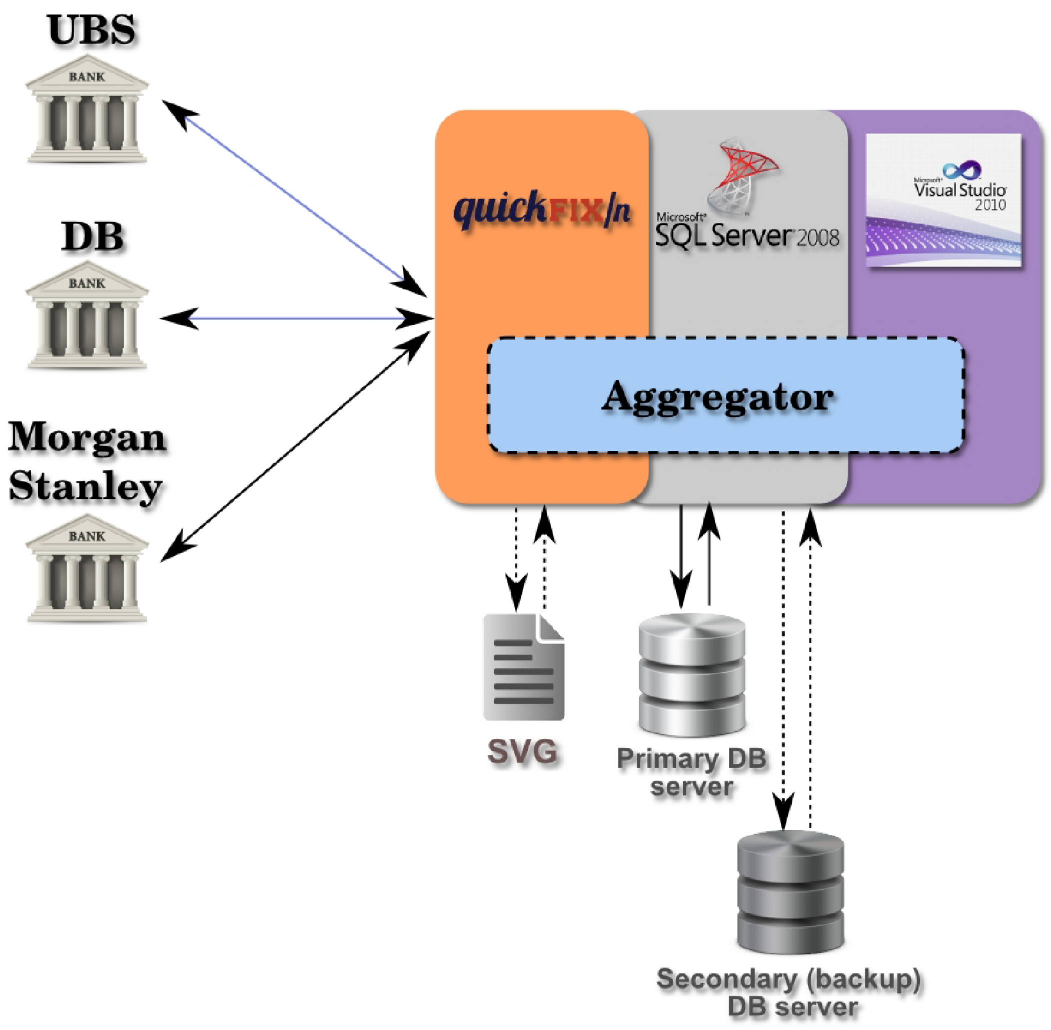

UBS
BANK
DB
BANK
Morgan Stanley
BANK
quickFIX/n
Microsoft SQL Server 2008
Microsoft Visual Studio 2010
Aggregator
SVG
Primary DB server
Secondary (backup) DB server

Figure 6.1: Aggregator application design

Let's sum up the specification requirements:

- collecting data from many liquidity providers
- easy extension of the system to implement trading strategies
- minimum latency in the aggregator processing
- the parsed market data can be stored in file optionaly and will be stored in relation database compulsory
- individual functional parts will be divided into separate libraries
- the code of the application will be optimized by profiler to reduce latency
- the user interface will allow managing input/output feeds online
- the user interface will provide custumizable window

6.2.1 Latency

Main feature concern of aggregator application is delay or in other words latency. Lets brake down the latency times between trading server and client. They are two types of latencies. The ones we can reduce and the ones we cannot. My focus was of course on the reducible latencies. Figure 6.2 provides an easy way of seeing these latencies.

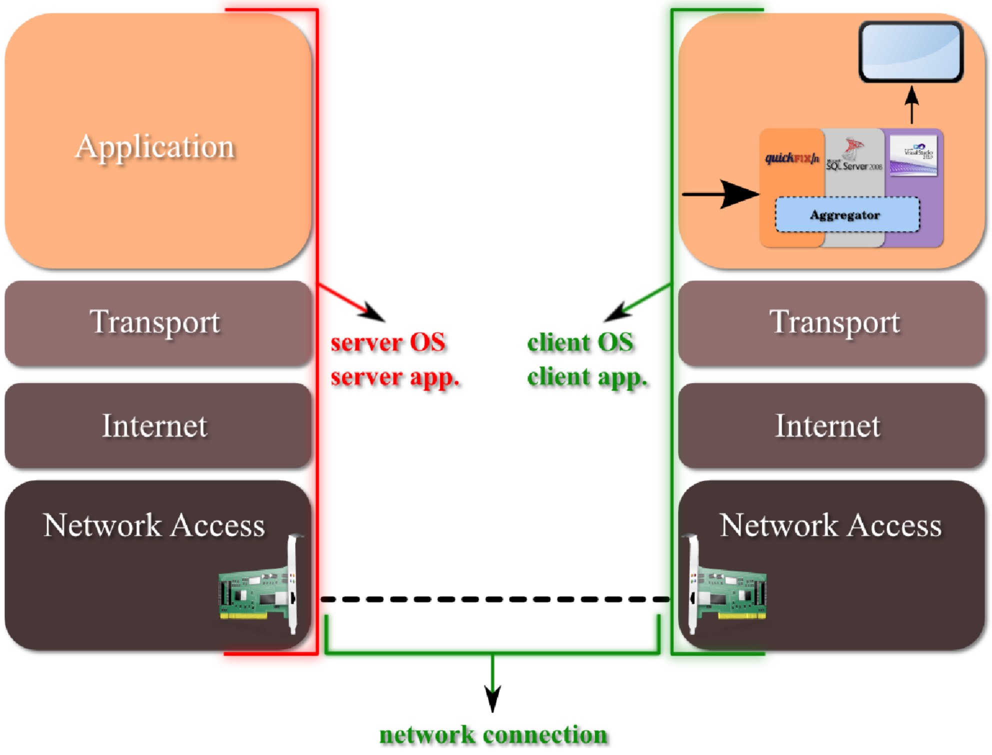


Figure 6.2: Latency break downs end-to-end

The red line surrounds us the server side. Bank, trading venue, in short liquidity provider. We cannot affect latency on this side. Here, the server is purely responsible for latency. Now, where can reduce latency is the rest part from network connection until the data displayed in the application window. Network connection depends on ISP (Internet Service Provider) which provides us with appropriate bandwidth, type of network connection, certain delay in ISP network topology. These terms should be included in SLA (Service-level agreement). Further, on the client side appropriate HW and SW configuration. For my solution I used Windows 7 64bit installed on my laptop. Quality network card, CPU, memory and data storage. These configurations depends on the number of liquidity providers and amount of data needed to be stored. In general for a simple client aggregator the HW and SW specification of my laptop is sufficient. That is, Lenovo L412, CORE i5 CPU, M520 2.4GHz, 2x2GB RAM 1333MHz, Windows 7 64bit with Service Pack 1.

In the aggregator application by itself includes three parts. Starts from QuickFIX/n API, progresses through aggregator, and finishes by writing the expected results on the application window. QuickFIX/n is a library so there is not any space for latency improvement. In the case of aggregation and writing data to display and storing them in data storage latency improvements are possible. Firstly, correct selection of sorting algorithms based on test results is necessary. This was completed in Chapter 5 of this Diploma Thesis. The Insertion sort and Quicksort with median pivot and DBSS were selected. Know these have to be implemented in aggregator along with writing the data to display and storage. Secondly, the finished working application latency is reduced by use of profiler. These conditions meet the requirements for aggregator application with minimum latency.

6.2.2 Modularity

Object oriented programing (OOP) allows us to look at desired solution as on objects. This is helps us to imagine interactions more easily, because working with objects is more intuitive. OOP allows us to design, program and maintain code more effectively. Aggregator application's modular solution is the follows:

- namespace AggregatorClient
  - class Aggregator
  - class DataStorage
  - class MarketData
  - class UserInterface

The aim was to divide the aggregator application into objects, where each represent main logical functionality. By fuctionality meaning different type of tasks. Let us look at the classes description.

- **class Aggregator** - implements methods for sorting Bid and Ask values.
- **class DataStorage** - implements methods for inserting market data from memory to database, handling primary database failure, backuping database.
- **class MarketData** - implements methods for storing marked data information in memory after processing with QuickFIX/n methods.
- **class UserInterface** - implements methods for handling UI events, UI components and user interaction.

6.3 Bank connections

In the case of Deutsche Bank and UBS security is guaranteed with encrypted transfer. Authentication with certificates can be handled by stunnel. Stunnel provides SSL tunnelling. Morgan Stanley security involves process of IP address approval. Firstly, trader or company has to send their IP address used for liquidity transfer. Secondly, this IP address goes through approval process in the bank and if approved, added as exception to their firewall.

6.3.1 Stunnel

The stunnel is an encryption wrapper between remote client and local or remote server. No change in program's code is necessary. It initiates SSL handshake using certificates. After successful SSL handshake client-server communication security is provided. Stunnel uses OpenSSL libraries for cryptography, it support any cryptographic algorithms compiled into my own library. It is distributed under GNU GPLv2 licence.

6.4 QuickFIX/n

Is a open source FIX engine that implements the FIX protocol on .NET. It provides methods for processing FIX protocol messages. the message content includes basically two types of messages. The first types are management messages. These role is to initiate Login, Logout and Session Management. The other types are trading messages.

6.4.1 Application callbacks

```
public class MyQuickFixApp : Application
    {
        public void FromApp(Message msg, SessionID sessionID) { }
        public void OnCreate(SessionID sessionID) { }
        public void OnLogout(SessionID sessionID) { }
        public void OnLogon(SessionID sessionID) { }
        public void FromAdmin(Message msg, SessionID sessionID) { }
        public void ToAdmin(Message msg, SessionID sessionID) { }
        public void ToApp(Message msg, SessionID sessionID) { }
    }
```

These methods are called on QuickFIX/n events. The next section brings description of the function of these events. These descriptions are from the official QuickFIX/n web page [13].

- **FromApp** - every inbound application level message will pass through this method, such as orders, executions, secutiry definitions, and market data.
- **FromAdmin** - every inbound admin level message will pass through this method, such as heartbeats, logons, and logouts.
- **OnCreate** - this method is called whenever a new session is created.
- **OnLogon** - notifies when a successful logon has completed.

- **OnLogout** - notifies when a session is offline - either from an exchange of logout messages or network connectivity loss.
- **ToApp** - all outbound application level messages pass through this callback before they are sent. If a tag needs to be added to every outgoing message, this is a good place to do that.
- **ToAdmin** - all outbound admin level messages pass through this callback.

6.4.2 Session configuration

Before any trading command can be initiated, Logon to the liquidity provider must proceed. Session is initiated whose configuration is read from the input session file. The session file can include the following settings:

- **Session** - BeginString, SenderCompID, TargetCompID, etc.
- **Validation** - DataDictionary, etc.
- **Initiator** - ReconnectInterval, HeartBtInt, SocketConnectPort, SocketConnectHost, etc.
- **Acceptor** - SocketAcceptPort, SocketAcceptHost
- **Storage** - PersistMessages
- **File storage** - FileStorePath
- **Logging** - FileLogPath

The Session settings contains information provided from liquidity provider to successfully create session and logon to their server. The configuration file example follows:

```
# default settings for sessions
[DEFAULT]
FileStorePath=store
FileLogPath=log
ConnectionType=initiator
ReconnectInterval=60
SenderCompID=TW

# session definition
[SESSION]
# inherits from default
BeginString=FIX.4.1
TargetCompID=ARCA
StartTime=12:30:00
EndTime=23:30:00
HeartBtInt=20
SocketConnectPort=9823
SocketConnectHost=123.123.123.123
DataDictionary=somewhere/FIX41.xml
```

If the connections to liquidity providers are running and market data stream messages are proceeded, the MarketData class methods take care of storing the market stream data into into memory.

6.5 Aggregation

The aggregator algorithm will be using two types of sorting algorithms. Insertion sort and Quicksort with median pivot and DBSS. The aggregation algorithm will run in two phases. In the first phase the algorithm determines the best Bid or Ask value. The Bid represents the selling price, we are searching for the highest value. The descending value order is our goal. The Ask represents the buying price, we are searching for the lowest values and we are representing it in ascending order. If multiple same values occur, the tick time of these values is compared. The latest tick time value has the highest priority. Phase two is responsible for this job.
In other words. The aggregation process will run on separately on Bid values and Ask values. Firstly, they are normally sorted and after equal values are sorted between them self according to tick time key. After the aggregatio process, the MarketData class contains the sorted values.

6.6 Data Storage

The market data insertion to database si handled by this class, DataStorage. This class implements methods to insert market data from memory to database.

The FIX specification specifies,that Market Data Request (MsgType = V) message has to be send to the liquidity provider to subsribe the currency pairs we want to receive. In some cases a standalone Market Data Request message needs to be send for each currency pair. In other we can subscribe more currencies at the same time. The Market Data Request includes the required symbols. The server replies with with Market Data Snapshot for each symbol. FIX name 'NoRelatedSym' defines the number of requested symbols. ISO defines the curency pair format. The format is 'CCY1/CCY2'and it is writen in FIX name 'Symbol'. 'MDReqIDt' is unique message identificator, has to be set on client side. We set the Snapshot+Update (value 1) value in 'SubscriptionRequestType', the reception of whole book and later updates. FIX name 'MarketDepth' expresses the number of subscribed levels. The next Table 6.1 specifies the message in more detail.

Tag	FIX name	Type	Description
35	MsgType	Char	'V' – Market Data Request
146	NoRelatedSym	Integer	No. symbols
55	Symbol	String	
262	MDReqIDt	String	
263	SubscriptionReqType	Char	This group repeats for each symbol
264	MarketDepth	Integer	
267	NoMDEntryTypes	Integer	
269	MDEntryType	Char	

Table 6.1: Market Data Request message - currency pair subscribe

To cancel a previously created subscription we pass the Unsubscribe (2) in the in the

SubscriptionRequestType (264) tag.

Message Market Data Stream is send to the client asynchronously in the case if change. As we know this represents the situation on market as book of orders. These message can be devided into two groups:

1. **Book order update** - includes information of current Bid/Ask offers, price, size, number of orders.

2. **Information of actual trade** - information about realized transactions.

If we summarize the data, we get the 10 following values required to be stored:

- StreamID
- Symbol
- BidPrice
- AskPrice
- BidSize
- AskSize
- BidOrders
- AskOrders
- BidTimeTick
- AskTimeTick

6.6.1 Storage design

We will store each tick from the market. One tick is a book of order snapshot in moment of time change. We divide them into two tables according to Bid/Ask items. This step simplifies the aggregation process. Now the Bid/Ask values can be sorted separately. Afterwords stored in database and file. StreamID uniquely identifies the stream. Later join opperation can be performed on them using queries.

- StreamID, Symbol, BidPrice, BidSize, BidOrders, BidTimeTick
- StreamID, Symbol, AskPrice, AskSize, AskOrders, AskTimeTick

6.6.2 Database

The Express version of the MS SQL Server R2 2008 is used. Unfortunately it comes with limitations mainly maximum 10GB storage per database, uses just one processor and limitation of 1GB RAM use. But, on the other hand, MS SQL is optimized for C# integration, provides features as database synchronization with later manual restore.

6.7 User interface

Windows Forms is a class in .NET Framework to create a window applications for Windows easily. This class suits the need for building appropriate aggregator user interface. The Use Case diagram involves two actors. Trader, with priviledges to view current curency rates, view history and export values from database to CSV open file format. The other actor is trader with higher priviledges or admin. He has the same priviledges as trader plus extended for user administration and application windows customization.

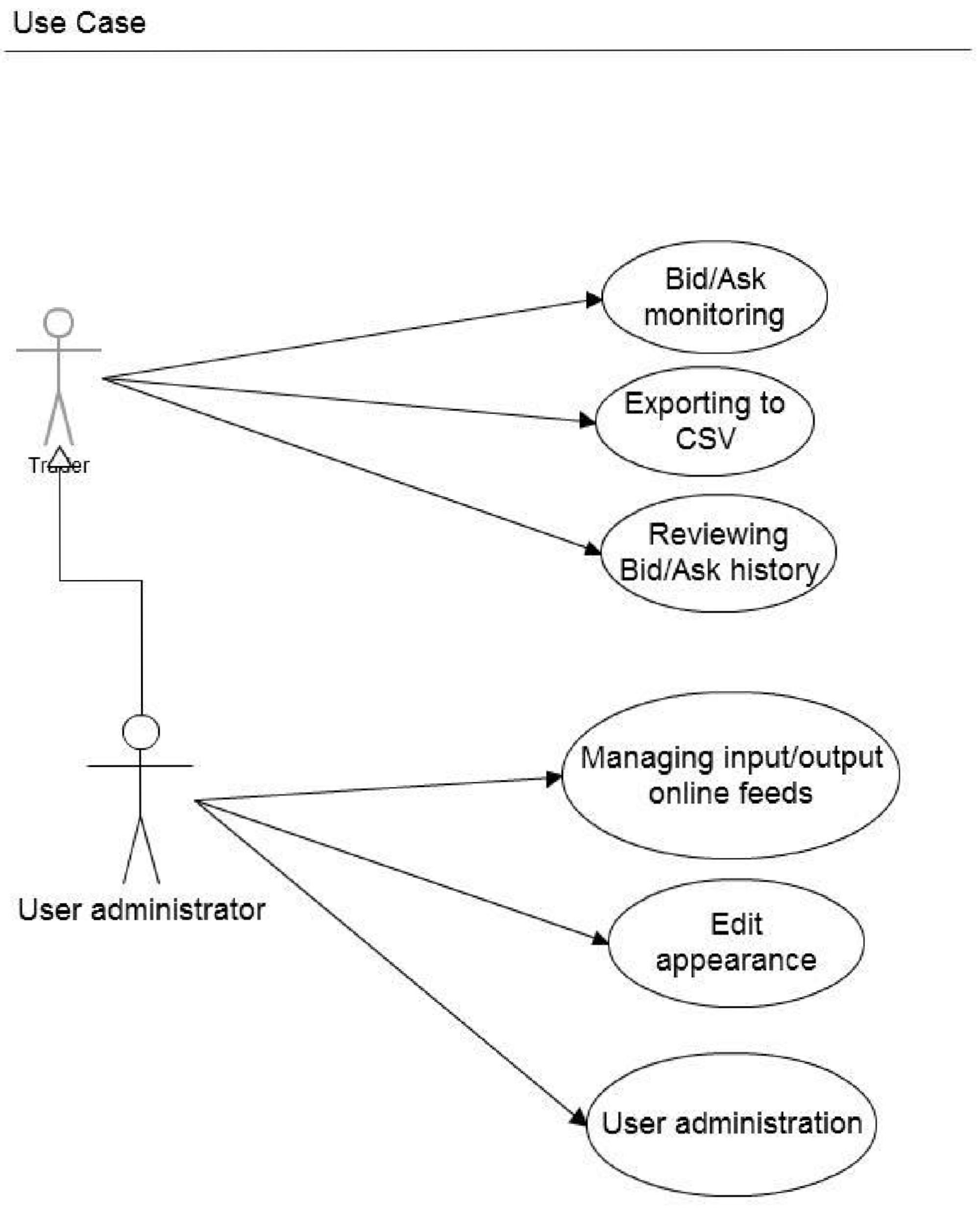


Figure 6.3: Use case diagram for user interaction with application

Chapter 7

Implementation

This Chapter shortly describes the implementation of two phasis, along with difficulities that occued and final result.

7.1 Sorting algorithm tester

I have divided the implementation into two major parts. The first part included the creation of application for testing sorting algorithms. The implementation involved the following sorting algorithms:

- Bubble sort
- Insertion sort
- Binary insertion sort
- Double burst selection sort
- Quicksort with rand pivot
- Quicksort with median pivot and DBSS
- Simo sort
- Mergesort
- Comb sort
- Gnome sort
- Timsort
- Strand sort

Further, testing on four types of input data, on list size from 10,20, 50, 100, 1000, 10 000, 100 000 for unique market data and otherwise for 1000 000 and finaly 10 000 000 list count elements. The testing result are writen to terminal output and also to text file 'result.txt'.

One difficulty was with Smothsort implementation. This implementation was a reedited version of C++ version. Unfortunately the result showed realy poor performance. This was probably due to bad algorithm reimplementation. I did not have time to look into this issue and try to solve it.

This implementation phase was completely finished.

7.2 Aggregator client

The second part was implementation of Aggregator client. At the begining of implementation phase I had a big problem. I could not connect to the liquidity providers.

First with Deutsche Bank I could not get any telnet, neither other port conenction. I also tried nmap for port scanning, without any success. Later we quiried the bank and they have responed, that they had changed their liquidity server IP. The new IP functioned two weeks and then the same problem occured. The connection to the bank started with SSL certification authentication provided by stunnel.

For Morgan Stanley a specific IP address had to be sent to go through the approval process, and if successful, they added it as an exception to their firewall. I discused this issue with the administrators in the vislittleiting university. They assigned an IP address to me and added exceptions to their firewall. Again a missunderstanding happened and my consultant forgot to send the IP address to the bank. The time I was in the visiting university the connection was still down, respectively firewall was blocking my connection.

The UBS liquidity servers were responding to my Ubuntu and eva FIT server also, throught testing ports. But somehow not from my Windows OS. Not even with turned off antivirus and firewall. Here, also stunnel should be used as an SSL wraper, but I was unable to create SSL session. The Wireshark showed no outgoing TSL/SSL packets (for both Windows 7 and Ubuntu 10.04). Not even a initiation HELLO packet. The configuration for stunnel was provided by UBS, but still. I tried a different version of stunnel, other transfer parameters, but could not create any SSL session. My consultant installed stunnel on his computer and replied, everything was working correctly and the session was created. Experimenting with stunnel slowed down the development rapidly.

I studied the material required for startign the development, but did not have enough time to solve the mentioned issues. The project was much more timeconsuming than I expected. In conclusion, the implementation of phase two was not proceeded.

Chapter 8

Conclusion

Developing a high-frequency solution can last few years, depending on the complexity of the project. The more sophisticated prediction, trading algorithms, the higher the complexity. The developing phase requires a group of skilled programmers, traders and other project staff. Liquidity source price stream aggregator however represents a part of a full featured trading solution. So the developing cycle is not so time consuming as in the case of all-in one solution.

Chapters 1, 2, and 3 gave me theoretical background for proper understanding the topic of financial markets, with focus on high-frequency trading and foreign exchange. Financial market communication protocol, Chapter 4, gives detail specification description useful for aggregator design and implementation related to communication with liquidity providers. Price aggregation Chapter covers the description of variety sorting algorithms, their properties and practical results. Sorting algorithm testing and result evaluation was followed by picking two of them. Insertion sort and Quicksort with median pivot and DBSS showed the most superb performance. Aggregator design specification specifies requirement for a stable, reliable, low latency aggregator application with secure data transfer. Further in Chapter 6, C# was chosen as programming language with the support of .NET Framework. It is comprehensive and consistent programming model is optimal for building a modular solution. Microsoft Visual Studio 2010 IDE includes enhanced features that simplify the entire software development process, long term maintenance and future improvements. MS SQL Server 2008 support the storage needs for the aggregator.

Due to complications during development the aggregator design was not implemented. Stunnel used as SSL wraper, was not able to connect to liquidity providers. I was not able to determine the cause of problem.

Firstly, the future improvements should include more liquidity providers. The wider the competition the better Bid/Ask price can be traded. Secondly, user interface interaction, friendliness and more costumization features can be added redesigning it with Windows Presentation Foundation. Thirdly, manual trading execution and especially algorithmic execution strategies moves the level of trading to a much more sofisticated level. Of course building these solution requires lot of skilled developers and traders.Longterm testing on real-world market data is necessary before real market trading.

Bibliography

[1] ALDRINE, Irene. *High Frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems.* Wiley Trading, 12 January 2010. First Edition. ISBN 978-0470563762.

[2] CORTESI, Aldo. Timsort example [online]. 2008 [cit. 2012-06-14]. URL: http://corte.si//posts/code/timsort/index.html.

[3] DOLLEY, Greg. Floating type testing [online]. 2007 [cit. 2012-06-02]. URL: http://gregs-blog.com/2007/12/10/dot-net-decimal-type-vs-float-type/.

[4] FIX Protocol Limited. FIX Protocol [online]. © 2012 [cit. 2011-12-18]. URL: http://fixprotocol.org.

[5] HESHAM EL-MAGDOUB, Mahmoud. Simo sort [online]. 2012 [cit. 2012-06-10]. URL: http://www.codeproject.com/Articles/344046/Simo-Sort-A-Sorting-Algorithm-for-Elements-with-Lo.

[6] HONZÍK, Jan Maximilián. Algorihms and Data Structures. Brno, 2007. [cit. 2011-12-27]. Technical report, Brno university of technology.

[7] JOHNSON, Barry. *Algorithmic Trading and DMA.* London: 4Myelona Press, 2010. First Edition. ISBN 978-0956399205.

[8] KLEIN, Radek. Asymptotická časová složitost [online]. 2008 [cit. 2011-12-27]. URL: http://radekklein.cz/asymptoticka-casova-slozitost-algoritmu/.

[9] KRESLÍK, Michal. FOREX. International foreign exchange trading [presentation]. 2006 [cit. 2011-08-21].

[10] Microsoft. Examination of Data Structures [online]. [cit. 2012-06-05]. URL: http://msdn.microsoft.com/en-US/library/ms379570(v=vs.80).

[11] Microsoft. .NET Framework 4 [online]. [cit. 2012-05-20]. URL: http://msdn.microsoft.com/en-us/library/w0x726c2.aspx.

[12] Microsoft. Selecting a Collection Class [online]. [cit. 2012-06-04]. URL: http://msdn.microsoft.com/en-us/library/6tc79sx1.aspx.

[13] MILLER, Oren. QuickFIX/n [online]. [cit. 2011-12-04]. URL: http://www.quickfixn.org/.

[14] National institute of standards and technology. Gnome sort [online]. [cit. 2012-06-08]. URL: http://xlinux.nist.gov/dads/.

[15] NESNÍDAL, Tomáš and Petr PODHAJSKÝ. Finančník.cz - komodity, akcie, burza, forex. In: Forex [online]. [cit. 2011-12-10]. URL: http://www.financnik.cz/clanky/virtual/forex/.

[16] OLEJNÍK, Tomáš. Zpracování obchodních dat finančního trhu. Brno, 2011. Diplomová práce. Vysoké učení tecnické v Brne, Fakulta informačních technologií, Ústav informačních systémů.

[17] PETERS, Tim. Timsort algorithm [online]. 2002 [cit. 2012-06-12]. URL: http://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[18] R. MARTIN, David. Sorting algorithms animation [online]. 2007 [cit.2012-06-08]. URL: http://www.sorting-algorithms.com/.

[19] SEDGEWICK, Robert. *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching, parts 1-4*. Addison-Wesley Publishing Company Inc., 13 July 1998. Third Edition. ISBN 978-0201350883.

[20] STEDFAST, Jeffrey. Binary insertion sort [online]. 2008 [cit. 2012-06-07]. URL: http://jeffreystedfast.blogspot.sk/2007/02/binary-insertion-sort.html.

[21] wikipedia community. Financial Information eXchange [online]. 2004-2012 [cit. 2011-11-28]. URL: http://en.wikipedia.org/wiki/Financial_Information_eXchange.

[22] wikipedia community. Foreign exchange market [online]. 2004-2012 [cit. 2011-11-28]. URL: http://en.wikipedia.org/wiki/Foreign_exchange_market.

[23] wikipedia community. Mergesort [online]. 2004-2012 [cit. 2012-06-06]. URL: http://en.wikipedia.org/wiki/Merge_sort.

[24] wikipedia community. Over-the-counter [online]. 2004-2012 [cit. 2011-12-11]. URL: http://en.wikipedia.org/wiki/Over-the-counter_(finance).

[25] wikipedia community. Quicksort [online]. [cit. 2012-06-10]. URL: http://en.wikipedia.org/wiki/Quicksort.

[26] wikipedia community. Strand sort [online]. 2004-2012 [cit. 2012-06-16]. URL: http://en.wikipedia.org/wiki/Strand_sort.

[27] wikipedia community. Timsort [online]. 2004-2012 [cit. 2012-06-12]. URL: http://en.wikipedia.org/wiki/Timsort.

Chapter 9

Appendix 1

Sorting algorithm results are displayed on three graphs. The first one 9.1 represent results for list count 10, where all sorts are included. For accurate measurement, system ticks are used. The second graph 9.2 show the same algorithms, but for list count 50. The time is still measured in ticks. Finally, the third graph 9.3 shows just the fastest sorting algorithms, for list count 100 000. In this case the time is measured in milliseconds.

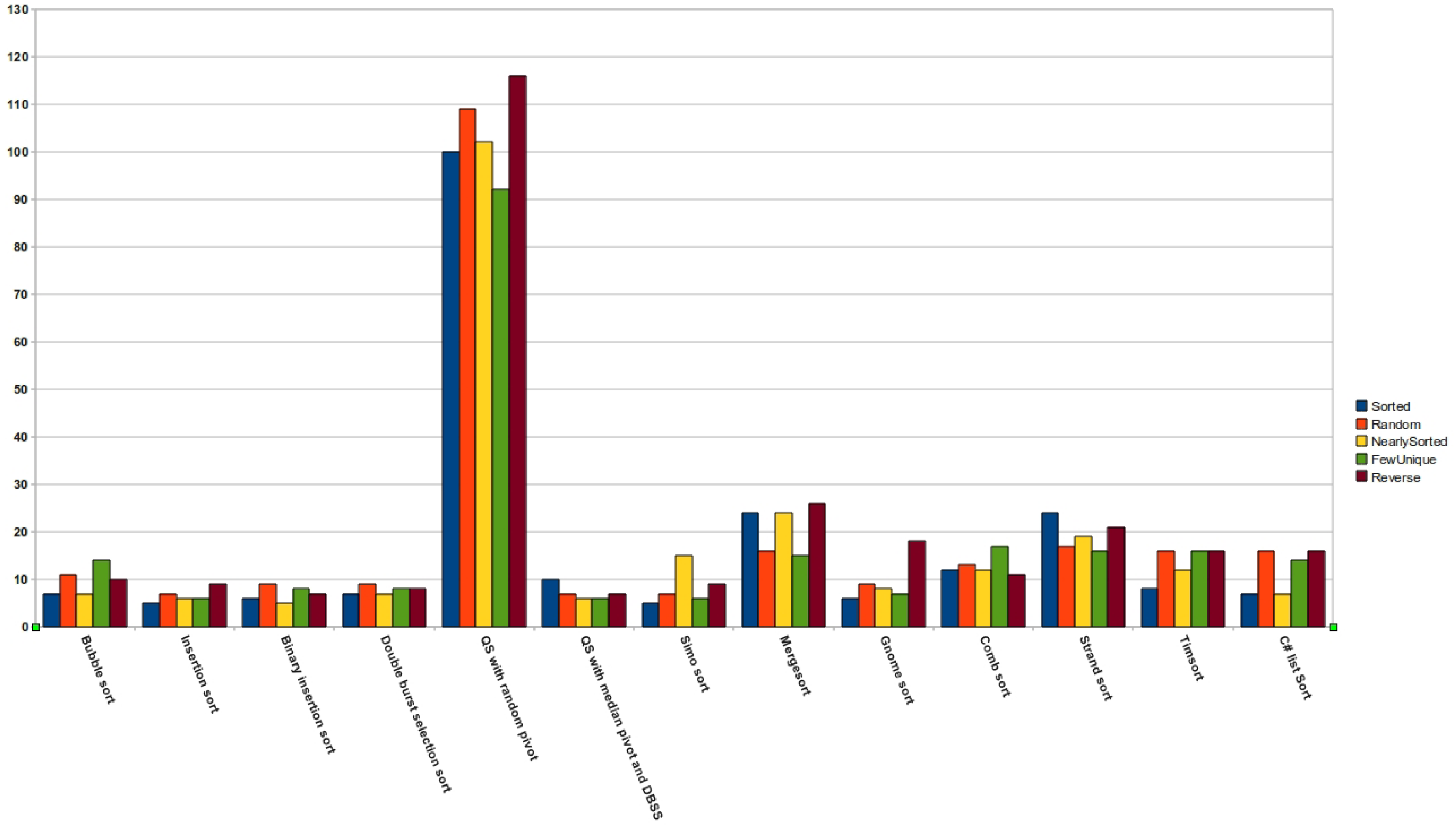


Figure 9.1: Results for list count 10, measured in ticks

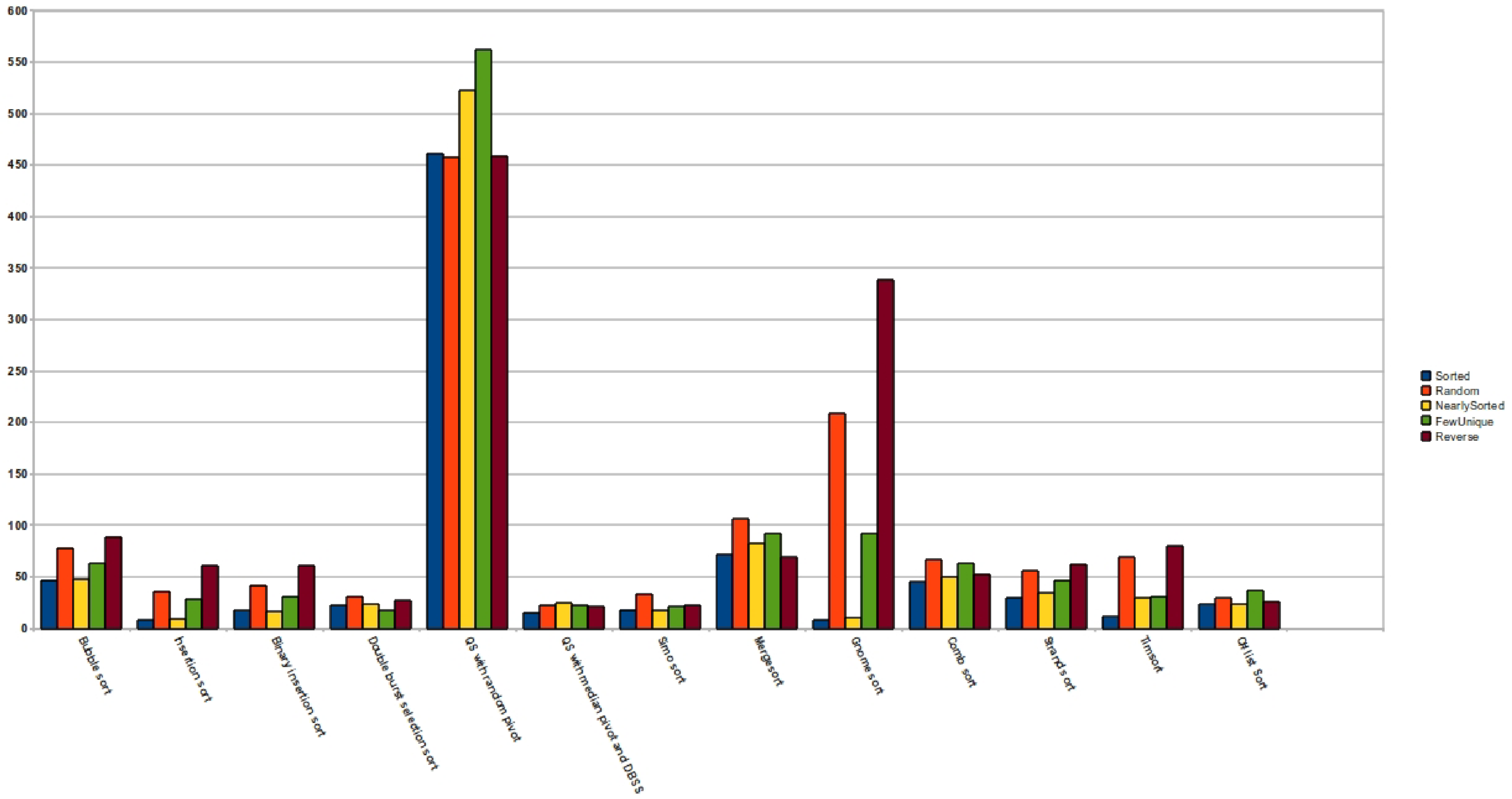


Figure 9.2: Results for list count 50, measured in ticks

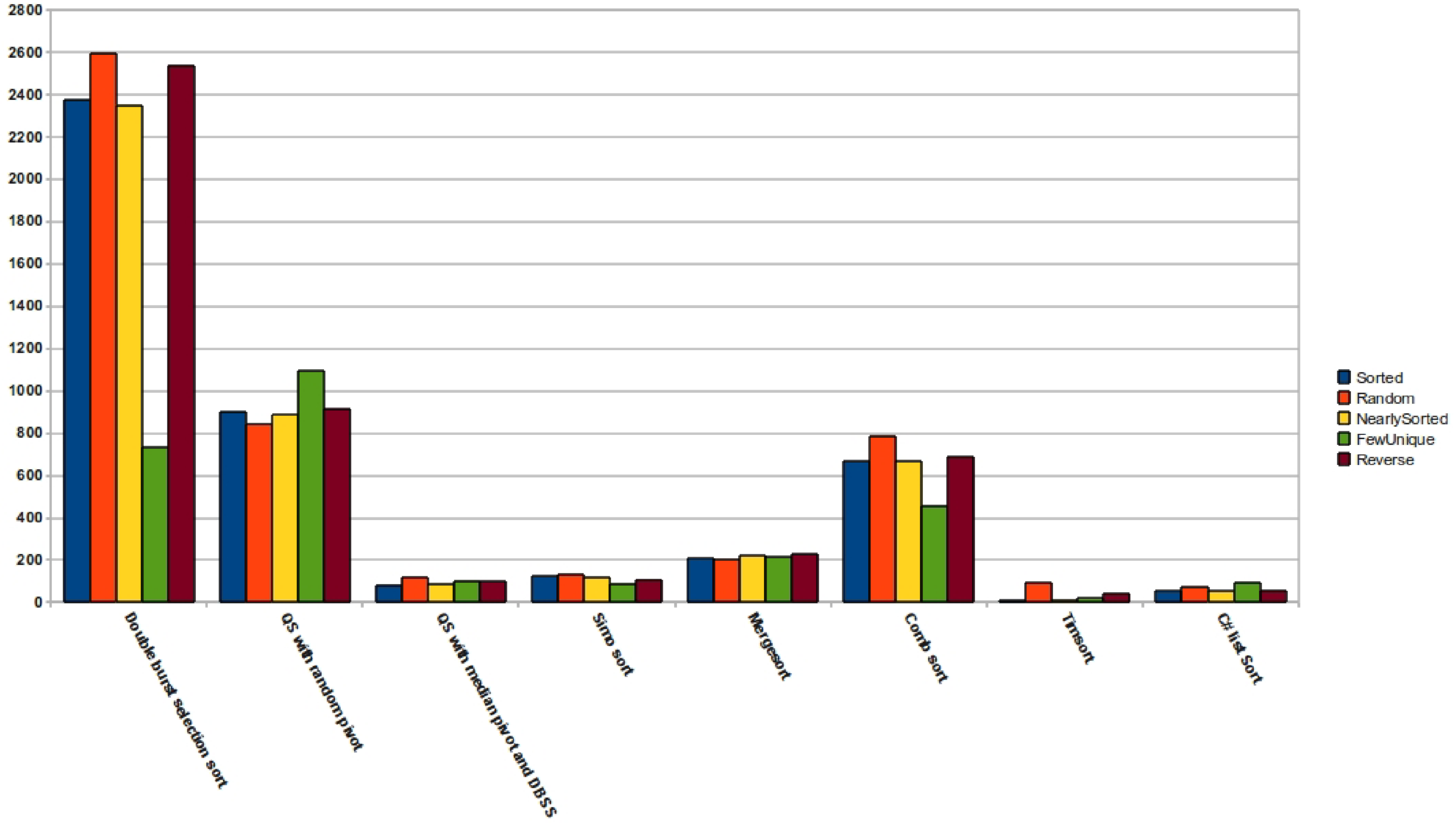


Figure 9.3: Results for list count 100 000, measured in milliseconds

Chapter 10

Appendix 2

The attached CD includes two folders. Folder Documentation includes normal and printed version (without functional links) dokumentation. The folder Sorting algorithms testing includes Project folder with the VS 2010 sorting algorithms project and Sorting results folder with the result from testing on different list count with all four input data distributions.