Quantitative Trading System

Denis Andrey Ignatovich
denis.ignatovich@mccombs.utexas.edu

May 5, 2006

Abstract

My interests are in the study of market microstructures, that is, how trading takes place in the markets and how those markets are organized. Models are designed to describe aspects of these organizations and one needs flexible toolsets for model description and performance analysis. The current step in my research is an implementation of a quantitative trading system. Not only is this a challenging systems engineering project, but also a powerful mechanism for data analysis and trade algorithm description.

Trading Model is defined as an investment tool comprised of buy and sell recommendations. These recommendations are much more complex than simple price change forecasts. They must be taylored to the investor in terms of his/her risk exposure, past trading history, and the market microstructure with its own constraints on the trade execution. A trading model has three main components:

- Generation of the trading recommendations.
- Accounting of the simulated transactions and their price impacts.
- Generation of the model statistics by the performance calculator.

A Trading System is, in turn, an environment where users define and, through execution feedback, adjust their trading models. An essential part of the system is the user-interface: it must be eloquent enough to allow an ease-of-use and, at the same time, powerful to describe most sophisticated trading algorithms. As in forecasting or other applications, trading models rely heavily on the quality of financial data. This constraint on the Trading System, a supply of the tick-by-tick data, is just an example of the multitude of requirements of a functional real-time trading environment.

Thie following paper describes an implementation of a quantiative trading system designed to incorporate features representative of a commercial grade trading environment. The system that I propose includes programmatic access to the underlying execution framework, powerful Python-based algorithm description environment, real-time data support, and mathetmatical interfaces, including support for ARCH-based equity volatility models.

This paper, in part, fulfills the degree requirement for Bachelor of Science in Computer Sciences (Turing Scholars Option).

Contents

1 Introduction

Quant is an algorithmic trading system. Using Python, users define *data analysis* and *trading* algorithms. The destinction is that former operate on inhomogeneous time series (tick data) and publish results to trading algorithms, that make trade decisions for portfolios they are responsible for. Users create portfolios by specifying their descriptions and initial capital. A portfolio may be managed by the *PortfolioManager* (part of *TradingModel*, please see below) that simply updates portfolio positions with the user manually modifying their composition weights. Alternatively, the user will assign a custom trading algorithm that would make the decisions automatically upon receiving signals of new data arrival. R-statistical environment provides the many necessary computational tools for making informed decisions.

2 System Model

***QuantWorld*'s synthetic trades** Designing solution to an engineering problem must start with analyzing the requirements in terms of input data, precision/algorithm running complexity, and results' presentation. In the case of Quant, there is an important issue of quality data: real-world trading systems rely on fast and precise trade information (and recently with the introduction of *NYSE's OpenBook*, even the specialists' books) for decision making. Lack of access to this level of data forces numerous assumptions on the part of the trader designing the algorithms. *Yahoo! Finance* provides 20-minute delayed price information distributed once per minute. It is the most commonly used source of financial data outside of the commercial realm, and due to the economic constraints of this project, it will service us with the foundation of our trade data. The necessary environment for the trade algorithm execution must provide the most up-to-date price information. The transition from *Yahoo! Finance* data to our algorithm environment (*TradingModel*) is supplied by *QuantWorld*, a component outside the trading system that simulates high-frequency data generation. The simulation will result in synthetic ticks containing bid/ask, price, and volume information distributed according to the parameters set by the user. The necessity of this price generation is specific to the author's use of the system, and therefore *QuantWorld*'s residence outside of the system allows for an substitution of a pricing information feed accomodating purposes of another user.

Data vs. Trading Algorithms Many operations performed by the users' algorithms would be quiet common data handling tasks. Thus, the decision to provide two algorithmic interfaces was raised by the need to factor out redundant tasks. The *Matrix* runs data-analysis algorithms and publishes the results for the trading algorithms.

UserInterface The *UserInterface* handles the task of informing the trader of not only the status and performance of his/her trading algorithms, but also provides a flexible means of their manipulation and adjustment. From my personal experiences in the industry, I have learned the ingenuity of *Microsoft Excel*'s interface. Many plug-ins to *Excel* form the foundation for decision making on Wall Street, and in my opinion, a successful system would incorporate these paradigms. At the same time, there are many tasks requiring programmatic accessibility. For this purpose, I have adopted *PyCute*'s model of a QT built-in command line parser for the interface. The control mechanism provided with the command line (or shell) entails priority execution mechanism such that any of the user commands vital to the system are executed before any data manipulation.

Summary The following are the main abstract goals of the system that served me as guides for the implementation:

interchangable data feed

real-time execution

flexible algorithm environment

3 System Architecture

3.1 Overview

Considering the constraints of a real-time system, components working on independant tasks must operate as individual threads. With this decision in mind, the next question was the method of facilitating communication between them. The most simple solution was to provide each component with a messaging queue and provide all other components with a reference to that queue. Whenit produced stock data, for example, the *Matrix* would create a thread that would wait for a notification of release of *Trading Model*'s queue and then insert the data. During the next iteration in the execution loop, *Trading Model* would wait on its queue and process the data. With the numerous references and deadlocks, I began to look for a different solution. After discussing the issue with Dr. Lavender, he recommended a publisher/subscriber model. I researched several implementations for Python and stopped on Netsvc for several reasons: for the most part, it is written in Python (no overhead code or installations needed with future deployment) and it has a very clean interface, yet enough to cover my requirements.

Netsvc package is used in two dimensions: (1) HTTP Daemon serves Remote Procedure Calls between the User Interface and the ControlCenter, and (2) Message Dispatcher runs the Publisher/Subscriber model internal to the system. The components communicate via several channels separated on the type of communication: data, messages, and commands. All messages sent within the system are of the following format:

(source, commType, args)

where:

source - string identifying component sending the message

commType - string indicating communication type ('message', 'data', or 'command')

args - tuple to be further parsed by the receiver

Messages are published through instantiating an object of the class *Netsvc.Service* and calling its method, *publish* with the message content and the intended channel. Components receive messages from a particular channel by, similarly, instantiating an object of the class *Netsvc.Service* and specifying the channels for subscription and callback methods for message handling. The callback methods insert messages onto a thread-safe queue that the component polls in its thread loop.

Component Descriptions:

Broker - receives trade requests and simulates the 'price impact' function. That is, user does not necessarily receive stocks at the price they are listed. Trading algorithms must account for this. Parameters of the impact simulator are set by the admin, similarly to the *QuantWorld* settings.

ControlCenter - the main component. It is responsible for starting/maintaining the other modules and servicing user requests submitted via RPC. It also aggregates data for the GUI output: stock data, system messages, requests results, and debug information.

TradingModel - creates/loads/runs user trading algorithms. When an algorithm makes a trade request, the model checks the parameters against current positions (do not sell more than currently have, unless short positions are allowed).

Matrix - 'listens' for the trade data submitted by QuantWorld, processes it, and performs additional user-specified operations, and then publishes results onto the network. The goal was to factor out common data operations used by several algorithms.

DataServer - stores trade data and any modifications to the current system state (new users, new algorithms, etc.)

Both the *UI* and *QuantWorld* were developed using *QT's Designer*. It is a great tool for constructing user interfaces that uses a middle language to describe forms in an implementation-independent way. After the *Designer* produces an XML description of the application, it is translated into Python class using *pyuic* that is used to derive GUI applications. An issue came up with signal interrupts for network communication while using QT's main event loop and signal handling methods. QT's messaging mechanism was incompatible with other signal handling utilities, uncluding the Netsvc package. The bypass was to simply rely on the Http RPC methodology. Trolltech is scheduled to release a flexible interface for custom signal handlers in the upcoming QT version. However, due to the open source nature of the Python implementation of Qt, it is unlikely this feature would be available for Python users any time soon.

3.2 QuantWorld

QuantWorld provides the necessity of a trading system: access to real-time data (synthetic). The user inserts companies he/she wishes to simulate, QuantWorld then queries *Yahoo! Finance* for their information. Every two minutes, QuantWorld polls the latest price for every company in the list. Between the update-periods, a Monte Carlo simulation 'produces' trades based on the latest price. The user can adjust both trade generation frequency and volatility. The trades are published onto the network for the Matrix to process. The frequency ranges from 1 to 120, or at maximum twice a second. A schedule is created every two minutes for each individual security in the list: based on the frequency, the times when a company will trade are randomly picked and marked in the schedule. For the next two minutes, the timer signal will call on a routine every half second and would compare the current time against stocks' schedules, and generate trades in cases of a match.

3.3 Interface

The User Interface connects to the system via HTTP, thus user can login from any place in the world with an Internet connection. There are three main

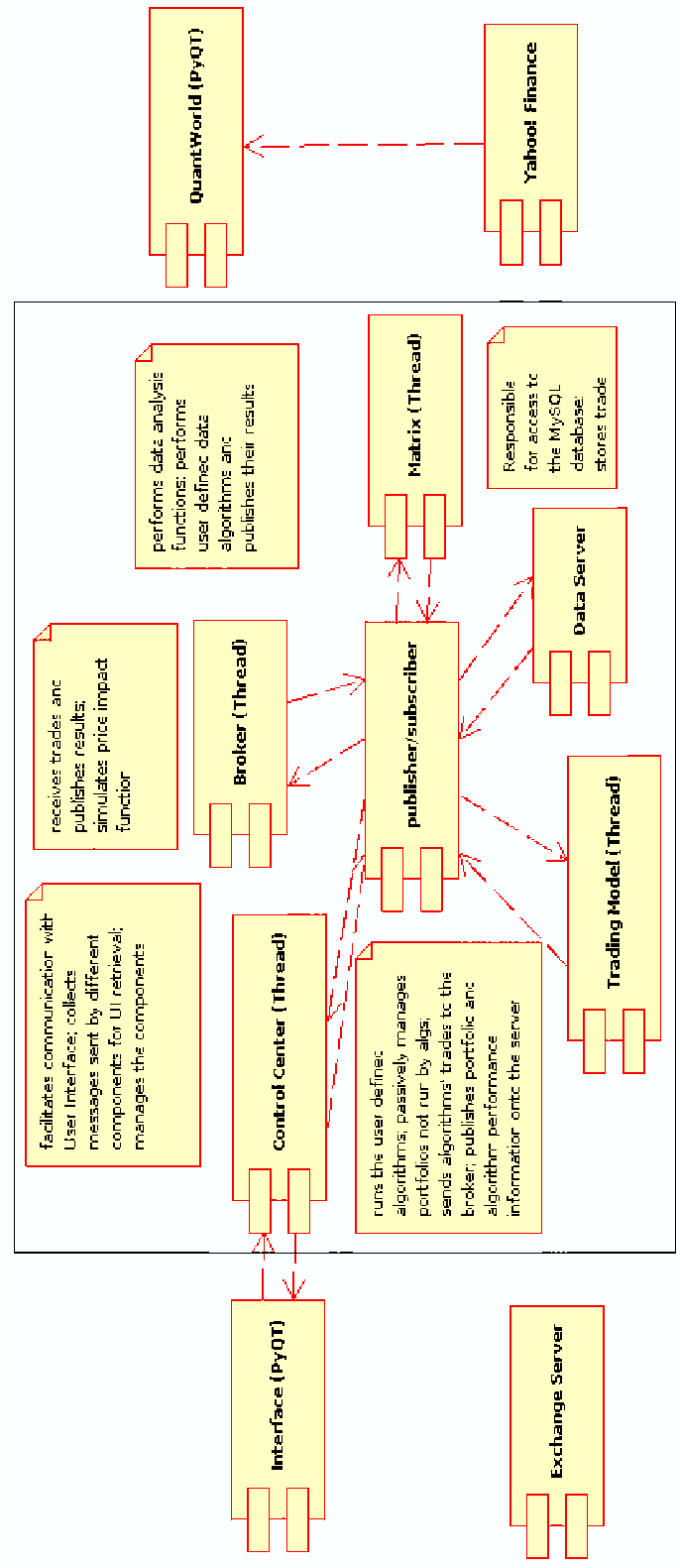

Figure 1: Quant System Architecture

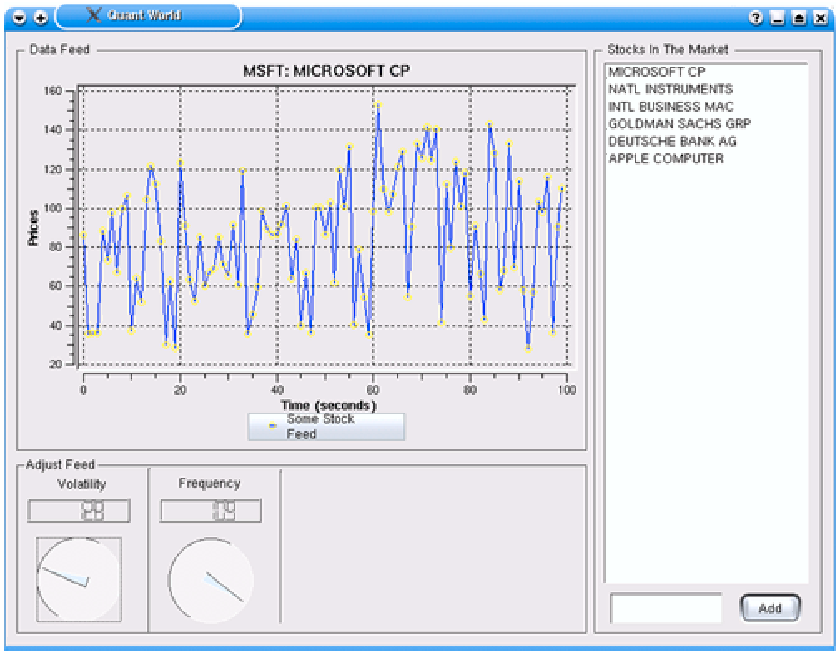

Figure 2: QuantWorld v1.0

components: command shell, messaging window, and the display. Derived from Python's *Cmd Class*, command shell provides programmatic access to the system. Initially, only the login command is visible to the user. He/she may type 'help' to view the full list of commands available. By typing 'help' followed by name of the comman, usage examples and detailed information are printed out. When the user enters a command, it is first parsed and checked for correct formatting, and then sent to the system (through a central point in the interface) along with additional user information. After successfully logging in (request is verified with the database), user's *Workspace* is initialized containing the stocks he/she is following on the display and the algorithms with respective portfolios loaded. Any changes to the *Workspace* are stored within the system, thus should the connection fail, data will be recovered.

Messaging screen echoes messages within the system (debugging information, if the option is selected, and system warning messages) and prints results of commands that user submitted. *UI* has an internal timer that executes every half of a second. On the timer event, *UI* asks the *ControlCenter* for the latest messages and resets the messaging queue in the system. For example, these include results of all portfolios' listing (with their descriptions, algorithms running it, current positions, etc.), the output will appear in the messaging window. System warnings and debug stream (if this option is set) are routed here as well.

The display is split into three pages: the matrix, stock visualization, and strategies' screen. The *Matrix* is an Excel spread-sheet like form with continuously updated company information (latest price, moving average, transaction volume, etc.), as well as results of any custom data analysis algorithms. The

user's workspace maintains the list of companies and other data tracked in the *matrix*, and any modifications are performed through the shell. *Stock visualization* is a real-time graph (PyQWT) of a given time-series. Strategies page follows the user's portfolios. Information related to status of the algorithm, its perfomance, and exceptions that might have occured are shown here as well.

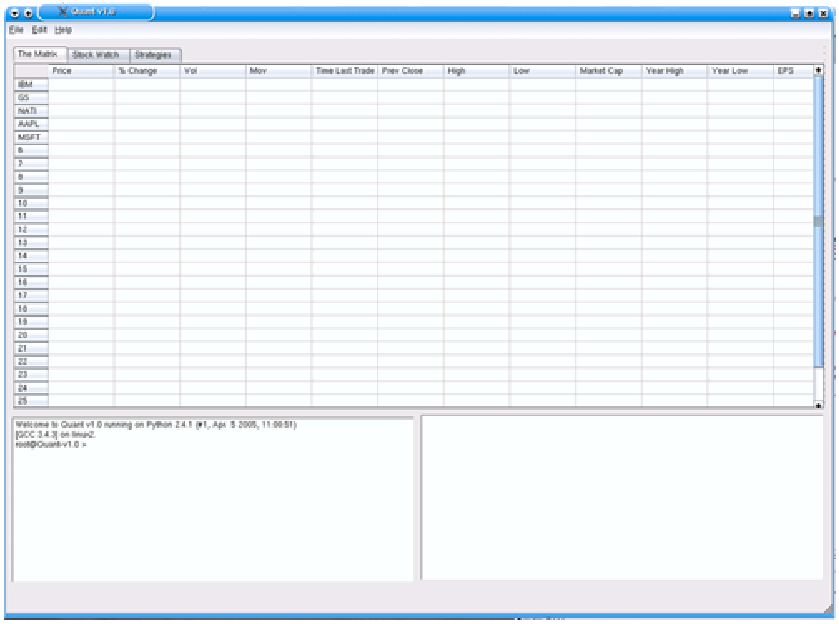

Figure 3: Quant User Interface

3.4 Trading Model

TradingModel is the center of algorithm execution. It maintains a pool of threads available for user algorithms, a list of currently loaded portfolios and the *PortfolioManager*, a semi-algorithm that merely updates positions of portfolios not managed by any user algorithms. When the algorithms make a trade recommendation, they publish it for *TM* to verify parameters of the trade. If the recommendation is valid, *TM* publishes trade request to the *Broker* without waiting for the reply; otherwise, request is ignored. Furthermore, it subscribes to all data coming from the *Broker* and when trade result is known, sends a message to the algorithm. Record of all resulting trades is maintained for each algorithm and performance analytics sent back to the *User Interface*.

There are several potentially dangerous moments when transferring control from an algorithm (when the user requested it to stop) to the *PortfolioManager* or when just creating a new portfolio. I implemented a library system to limit access to any portfolio to only one thread at a time. *TM* manages all transitions and maintains a status variable (secured under a conditional variable) for each portfolio. For example, when the user sends a command to stop execution of an algorithm, the managed portfolio must be released and handed off to the *PortfolioManager* for position updates. *TM* will first check the status of the algorithm, and it will send a 'stop' command if it is currently running. When the

algorithm parses that command, it will cease all operations and send a request of transfer back to the *TM*. After processing the transfer request, the portfolio will be added into the *PortfolioManager*'s influence.

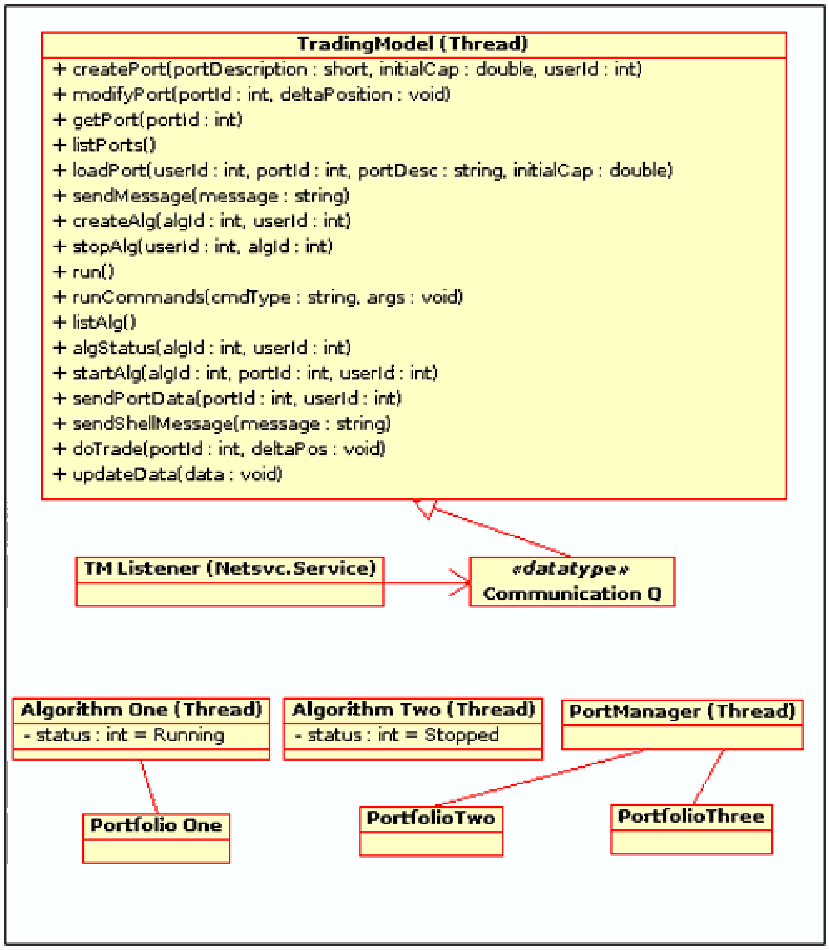

Figure 4: Trading Model UML

3.5 Matrix

As stated above, the *Matrix* processes trade data published by the *QuantWorld*. Its structure is similar to the *TradingModel*: a *listener* object subscribes to the data feed and inserts trades onto the queue. The *Matrix* Thread then distributes trade data among the data analysis algorithms that, in turn, process it and publish their results via 'data' channel. The format for submission is a dictionary that contains two fields, 'ticker' and 'type', and an arbitrary number of data values. As there are no shared objects, the implementation is fairly straight forward. The interface for inserting/removing algorithms is similar to *TM*'s. Specifically, commands are published with destination 'Matrix' that specify the Python code for the algorithm and the owner's *userId*. A new thread

is created with access to the incoming data and the algorithm information is backed into the database.

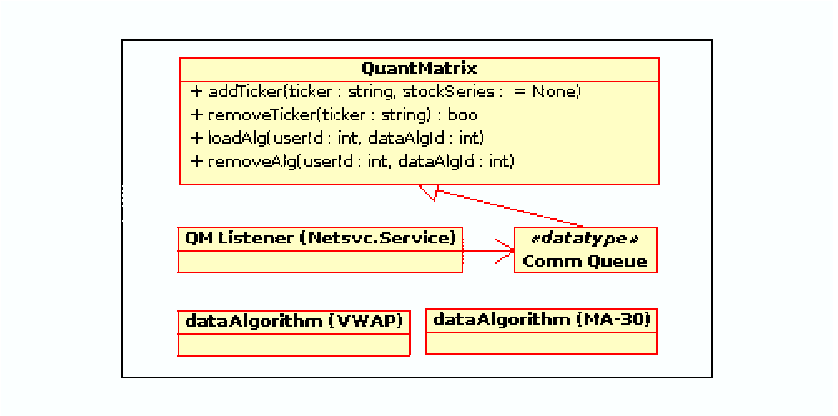

Figure 5: Matrix UML

3.6 ControlCenter

ControlCenter is the only component that is not derived from a thread. Its purpose is to provide access to the system and maintain 'sanity' of the components. After it starts up the exchange server, messaging dispatcher, and threads running the components, it communicates with them only via publisher/subscriber architecture. On the side of *UserInterface*, it creates an HttpDaemon and exports several methods via RPC:

command (user, func, args) - central method that matches request with an operation and dispatches command to the component

checkConnection (user) - returns status of the user within the system

setupWorkspace - initializes user's workspace: loads the list of stocks tracked through UI, loads portfolios and algorithms

retrieveMessages - returns internal system messages and resets the messaging queue

retrieveShellMessages - returns messages labeled for shell output (results from user requests to components) and also clears the queue

retrieveStockData - returns latest stock information as specified by the workspace

retrievePortData - returns user's portfolio and algorithm information

3.7 Data Server

3.7.1 Functionality

The *DataServer* stores all changes to the system state into MySQL server. This includes all communication except system and shell messages. Since the

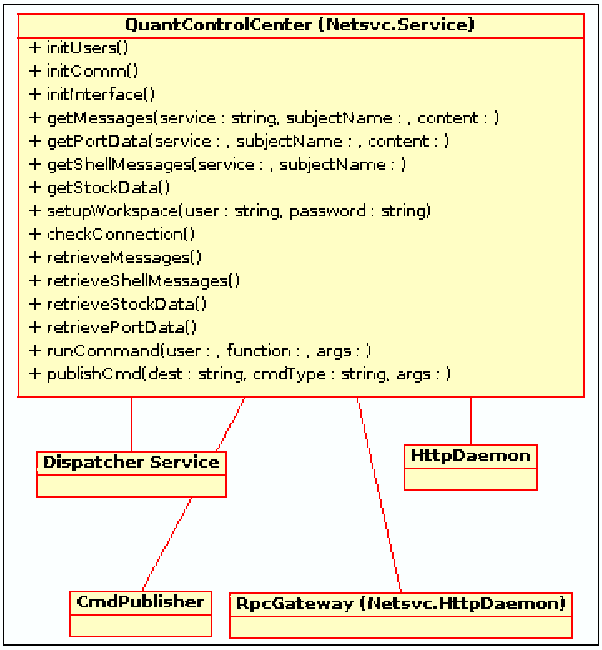

Figure 6: ControlCenter UML

MySQL server is, in fact, a server, it would be redundant to provide an interface within *DataServer* for information retrieval, as the components maintain their own connections.

3.7.2 Database Schema

3.8 Implementation

Implementation of this project is the result of many trials and errors. I am very glad that I have chosen Python as the primary language, it has provided elegant ways of connecting numerous different components. Netsvc Python package is a big recent change: initially the inter-component communication resided on several producer/consumer oriented queues. Race conditions and deadlocks became an issue and Netsvc publisher/subscriber model really made the difference. R-environment for statistical computing, and RSPython - its interface to/from Python, greatly expanded functionality of the system.

4 Application

4.1 Introduction

One of the interesting features of Quant is the separation of data handling and trading algorithms. In the following example, we will incorporate several components of Quant to solve a relatively common task in the world of finance:

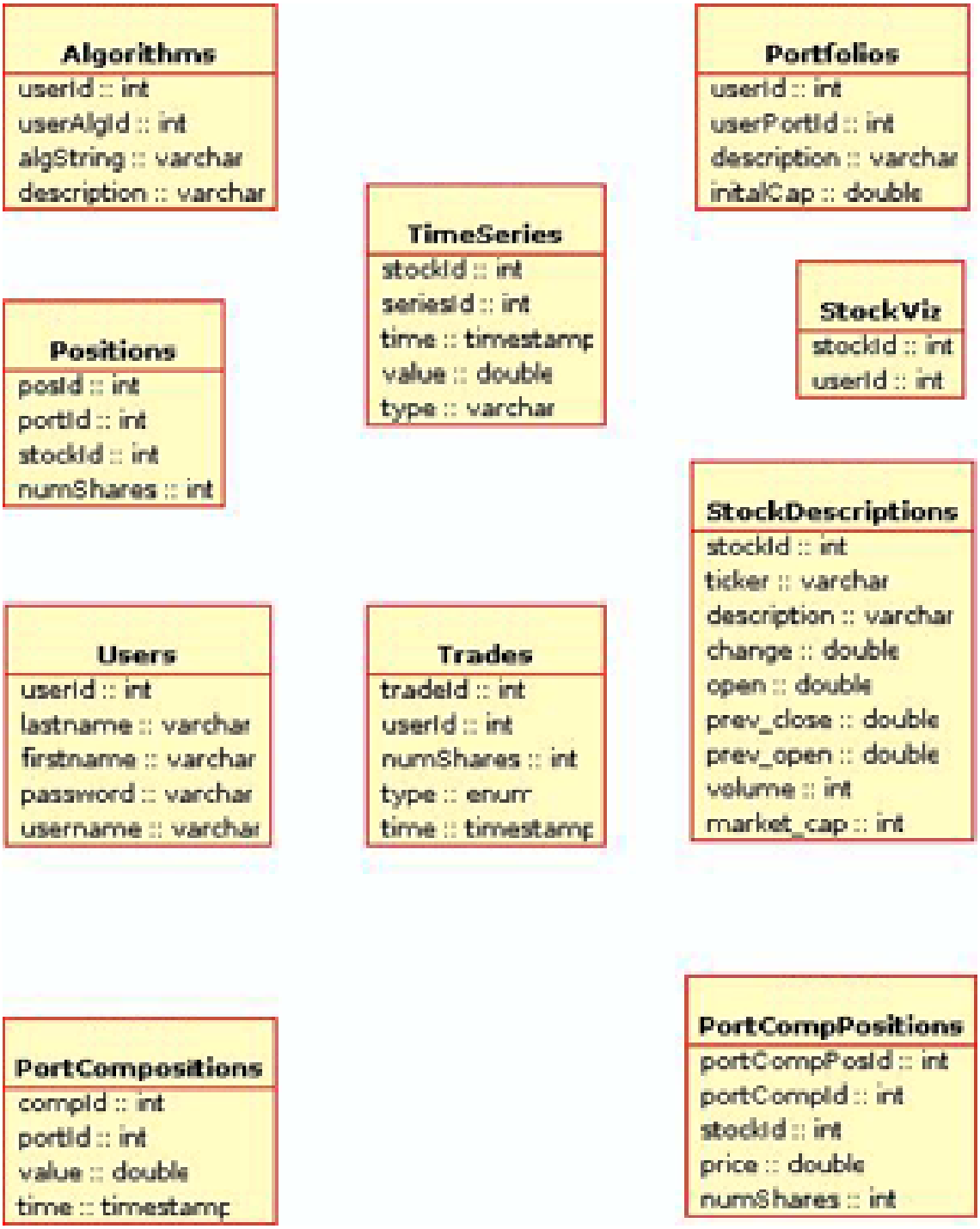

Figure 7: SQL Schema

portfolio optimization. Two data sources are available to us: (1) the historical prices (courtesy of *Yahoo! Finance*) for volatility calculation, and (2) real-time trade data supplied by *QuantWorld* for the current returns and correlations. Since we're given heterogeneous (inhomongeneous) time series, i.e. irregularly spaced, we will compute volume-weighted average price (VWAP) for normalization of each series (this will provide us with a common point of reference for all of the stocks in the portfolio). The next step would be to take the historical prices and run Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model to compute conditional variance (risk) for the series. Once we have both of these results, we will run quadratic optimization to determine the best weights' allocation for our portfolio.

4.2 Optimal Risky Portfolios

Our goal is to efficiently construct a diversified portfolio of equity securities. By efficiently, we mean deriving the most out of the risk we infer from making our investments. There are two sources of risk to our portfolio: market (or systematic) and firm-specific. Every security has exposure to systematic risk, it is inherent to all stocks traded in the market. The effect of diversification, however, is derived from combining securities whose firm-specific risks offset each other. For example, when oil futures' prices rise, an oil company's stock price will rise, while high-tech's may fall. This effect is the foundation of diversification and in the next several sections we will look at tools allowing us to describe composition of portfolios such that we minimize the non-systematic risk and attain highest-possible return per unit of risk. The concept of diversification was presented by Harry Markowitz in his "Portfolio Selection" article in *Journal of Finance, 1952*[15] as part of his *Modern Portfolio Theory*.

The model started out by making numerous unrealistic assumptions:[3]

1. Potential investments are derived from a probability distribution of expected returns.
2. Wealth has a diminishing marginal utility.
3. Decisions are made based on *expected* return and risk.
4. Risk is a function of variance of past returns.
5. Investors are rational: from a set of investments at the same risk level, they will prefer one with the highest return.

We will now proceed by examining the basic units we will use to further develop our models. Average of the underlying stocks' returns measures return of the overall portfolio.

$$r_p = w_1 r_1 + w_2 r_2$$

Non-linearity of the portfolio's risk (as shown below for a two-asset portfolio) will provide us with the key for diversification:

$$\sigma_p^2 = w_1^2 \sigma_1^2 + w_2^2 \sigma_2^2 + 2 w_1 w_2 Cov(r_1, r_2)$$

The last term in the equation will be negative for stocks that historically offset each other's risks, that is, to some extend, they move in opposite directions. Therefore, overall portfolio risk will be less than just the weighted average of the two. The effect of their combination is illustrated in *Figure: Effects of Diversification*, where there are several portfolio opportunity sets for different levels of correlation. In theoretical finance, the more risk an investment entails, the higher return the investor will seek and vice versa. The ratio of return to risk varies with different assets; that taken together will provide us with ratio for our portfolio. Our goal is to derive weights, or fractions of overall capital invested in individual assets, such that we derive the most return per unit of risk. US Government Treasury Bills, or T-bills, are considered risk-free assets, and they symbolize the minimum absolute return an investor should seek. We will come back to this concept when we discuss CAPM, a model that addressed combination of the risk-free asset and MPT's efficient portfolio.

Efficient frontier is a set of portfolios, such that, for any risk level, the portfolio with the highest return is included in this set. "Alternatively, the frontier is the set of portfolios that minimize the variance for any target expected return" [4]. Given our set of securities, MPT suggests forming a portfolio that would lie on the efficient frontier.

One of the limitations of MPT is the reliance on *expectation* of securities' returns. The issue is the assumption of efficient markets. We do not have any information that is not already available to the general public. Thus, only the historical data is available to us, but we must form expectations of the future returns.

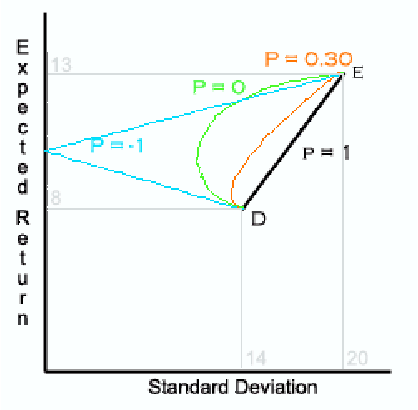

Figure 8: Effects of diversification in a simple two-asset portfolio

4.3 CMT & CAPM

Capital Markets Theory was introduced by William Sharpe (Jack Treynor, John Litner, and Jan Mossin also arrived at the theory independently) in 1964 and was based on Markowitz's MPT of twelve years earlier. The model starts out with the same set of assumptions as MPT and adds severl others [3]:

1. Investors form their portfolios on the MPT's efficient frontier.

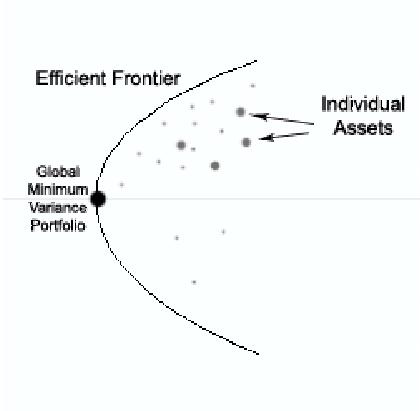

Figure 9: Forming an efficient frontier

2. As noted above, there is an asset with zero risk (T-Bill), and investors may borrow and lend at its rate.

3. Investors have homogeneous expectations of future returns.

4. All investors share the same investment horizon.

5. The operating microstructure has no transaction costs.

6. The interest rates (that dictate, among other things, the risk-free rate) are constant.

7. All investments are properly priced (no arbitrage opportunities).

As stated above, MPT dictates that the only portfolios we would like to form are on the efficient frontier. But how would one account for the risk-free asset that is visualized on the return (vertical) axis? As it turns out, if we think about how portfolio variance and return are computed and combine our efficient portfolio with a risk-free asset, a new efficient frontier will form. Let i denote our portfolio on the efficient frontier. If we combine it with a risk-free asset, the return will equal:

$$E(R_{port}) = w_{RF}(RFR) + (1 - w_{RF})E(R_i),$$
$$\text{and } \sigma^2_{port} = w^2_{RF}\sigma^2_{RF} + 2w_{RF}(1 - w_{RF})r_{RFi}\sigma_{RF}\sigma_i.$$

But, by definition, risk-free rate has zero variance and its covariance with any asset would be zero, we therefore derive the following definition:

$$\sigma_{port} = (1 - w_{RF})^(2)\sigma^2_i$$

This linear relationship provides us with means of stepping out of our efficient frontier by shifting capital in/out of our efficient portfolio and investing/leveragin the risk-free asset. We still have to keep in mind the return-to-risk ratio. When we look at all of the possible lines or opportunities of combining efficient portfolios with the risk-free asset, we will notice that there is one that

will dominate all others in terms of that ratio. This line is referred to as *Capital Market Line* (CML) and is formed by two points: the risk-free rate and the tangency portfolio on the efficient frontier. The latter is called the *Market Portfolio* and since the markets are in equilibrium, it combines all assets in the world (including those not traded).

If we wish to increase the return of our overall portfolio, we simply borrow at the risk-free rate and reinvest the proceeds into the *Market Portfolio*. The result will dominate the portfolio on the efficient frontier. Even though this notion of a portfolio combined of all assets in the world, and that is further complicated by the assumption of markets in equilibrium, we will later use it to derive a model we can easily implement.

Capital Asset Pricing Model (CAPM) understands that every asset should be as close to the CML as possible. And derives the source of risk of an asset as its measure of variances with the *Market Portfolio*, β.

$$E(r_i = r_f + \beta_i[E(r_M - r_f]$$
$$\text{where: } \beta = Cov(r_i, r_M)/\sigma_M^2$$

Beta can be viewed as a systematic measure of risk[3]. Since covariance of an asset with itself is just the variance, β of the *Market Portfolio* is 1. Assets with β above 1 are considered more volatile than the market and vice versa. This linear relationship forms the *Security Market Line* and it provides a measure of the market-required rate of return. An analyst may compare this expected return against his own expectations to determine whether the asset is priced correctly. The importance of CAPM to our cause is the relationship between risk and expected return of a security, and in the next section we will tie in past returns to arrive at a solution to our goal of portfolio optimization.

4.4 Index Models

Risk is probability that investor's return will deviate from the expectation, as commonly measured by standard deviation of the returns. We will start out by discussing risk structure of an equity security. Under the Single-Index Security (SIS) Model, risk is divided into two components: systematic and firm-specific. Fluctuations in the market (as measured by S&P500, for example) as a whole dictate performance of individual stocks. This relationship is described as exposure to systematic risk and is part of an investment in every security varying with the measure of sensitivity, β. SIS seeks to derive an estimate of expected return for a given secruty by combining all macroeconomic factors into one component, a measure of the market. The equation defines expected return as a linear regression on the market measure (S&P 500).

$R_i = \alpha_i + \beta_i R_M + e_i$

R_i - rate of return on security

α_i - the stock's expected return if the market is neutral, that is, its excess return is zero

$\beta_i R_M$ - component of the excess return attributed to the overall market

e_i - unexpected component due to events specific to this security only

This equation looks very similar to the one describing excess return using CAPM. In fact, we can combine the two in order to form a model that would use the historical data. The results will be the necessary expected returns for the *Capital Market Theory*.

We looked at the progression from MPT and its efficient portfolios into CAPM. In our algorithm examples, we will demonstrate how to use our trade data to derive optimal weights for the portfolio.

4.5 Data Filtering and VWAP Calculation

The trade information we receive from QuantWorld will contain 3 components: the purchase price, the number of shares purchased, and time of the trade. The user may define custom algorithms that would process and then publish the results to the trading algorithms, that in turn use them to make decisions. The following is an example of publishing custom information onto the network, VWAP, specifically:

```
for stock in self.timeSeries.keys():
    vwap = qLibVWAP (self.timeSeries[stock])
    data = {'ticker':stock, 'time':self.currentTime, 'type':'vwap'}
    self.publishData (data)
```

Figure 10: Data Analysis Algorithm

4.6 GARCH-based Volatility Calculation

Intuitively, even from a brisk examination of financial time series, such as S&P 500 returns, we understand that during some periods in time, investments in some assets contain greater risk than others. "That is, the expected value of the magnitude of error terms at some times is greater than at others" ([14]). However the previously standard model of least squares dealt with homoskedasticity or assumption that "the expected value of all error terms, when squared, is the same at any given point" [14]. Our intuitive idea is the basis for the opposite. Heteroskedasticity takes into account variance in the actual error terms (see Figure 5). Before the introduction of the ARCH model (1982), rolling standard deviation was the primary tool for future return predictions based on data of past 22 days (month's work days). ARCH took that idea of equally-averaged squared residuals of the past month, and turned it into a process where the weights would be subject for an estimate. In 1986 Bollerslev, student of Engle, proposed Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model that changed the weighting scheme from the past month to one asymptotically approaching zero. The resuling model is easy to estimate and is an effective predictor of return variance. If we consider the regression $r_t = m_t + \sqrt{h_t}\epsilon_t$, where r_t is the current return, m_t is the mean, h_t is current variance and ϵ is standard error, we can express the next period's variance using the following equation (with constants ω, α, β to be estimated):

$$h_{t+1} = \omega + \alpha(r_t - m_t)^2 + \beta h_t = \omega + h_t\epsilon_t^2 + \beta_t$$

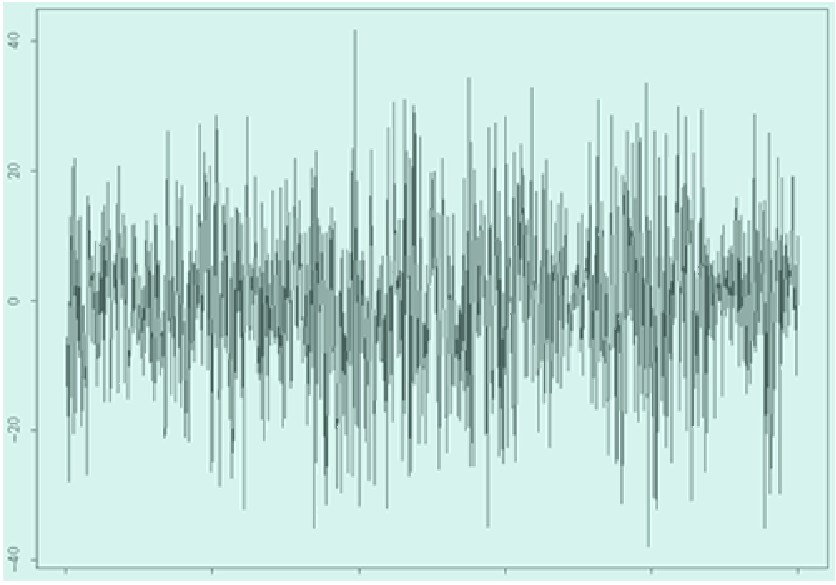

Figure 11: Presence of heteroskedasticity

Several issues arise when applying GARCH models to intra-day high-frequency data: because of difficulties of extrapolating coarse volatilities (caused by long-term traders vs. fine) from the data, methods estimating the GARCH parameters rarely converge, and in most cases when they do converge, the results are statistically irrelevant. There have been many publications on applications to intra-day data, however they are beyoung the scope of this report and we will implement the model on the daily volatilities incorporating several-months data. The following code will illustrate estimation of GARCH(1,1) parameters for an individual security:

```
class CustomPortfolioOptimizer (QuantAlgorithm):
    def calcGARCH (self):
            if self.lastCalcData == self.today():
                    return
            self.lastCalcData = self.today()
            for stock in self.myAssets.keys():
                    histData = self.retrieveHistoricData (stock)
                    self.GARCH_Var[stock] = qLibGARCH (histData)
```

Figure 12: Calculation of GARCH parameters

The internal library already provides for us 'qLibGARCH,' function running the GARCH method. But here we take the first step for creating a functional trading algorithm: we create a new class derived from 'QuantAlgorithm' and write a method 'calcGARCH' we will use later.

4.7 Complete Algorithm

We have seen the theory behind efficiently weighted portfolio, how to derive expected returns from the historical data, and how to calculate the volatility.

With GARCH calculation defined above, the following is an implementation of a portfolio optimizing algorithm:

```
class CustomPortfolioOptimizer (QuantAlgorithm):
    def calcGARCH (self):
            pass
    def runAlgorithm (self):
            self.calcGARCH()
            minPos = [0.1] * len(self.myAssets.keys())
            maxPos = [0.9] * len(self.myAssets.keys())
            riskFree = 0.035
            newWeights = qLibPortOptimize (self.myAssets, riskFree, minPos,
                                           maxPos, self.GARCH_Var)
            self.adjustWeights (newWeights)
```

Figure 13: Portfolio Optimization

5 Future Work

There are several projects I am interested in pursuing after I finish Quant: (1) connecting Quant to an artificial stock market, such as the Santa Fe version, and (2) researching the formal description of financial contracts (options, swaps, etc.) using a functional approach and building an automated system to screen for arbitrage opportunities.

6 Acknowledgements

I would like to thank Dr. Greg Lavender for his continuing guidance, expertise, and support that made this project a possibility, and Deutsche Bank's Index Arbitrage and Quantitative Strategies Groups for the invaluable experiences.

References

[1] Lutz, Mark. Programming Python. Sebastopol, California: O'Reilly Media Inc., 2003.

[2] Dacorogna, Michel, Gencay, Ramazan, Mueller, Ulrich, Olsen, Richard, and Pictet, Olivier. Introduction to High Frequency Finance. San Diego, California: Academic Press, 2001.

[3] Reilly, Frank, and Brown, Keith. Investment Analysis & Portfolio Management. Mason, Ohio: Thompson South-Western, 2003.

[4] Bodie, Zvi, Kane, Alex, and Marcus, Alan. Investments. New York: McGraw Hill, 2002.

[5] Harris, Larry. Trading and Exchanges. New York, New York: Oxford University Press, 2003.

[6] Lutz, Mark and Ascher, David. Learning Python. Sebastopol, California: O'Reilly Media Inc., 2003.

[7] ASPN Python Cookbook. 05 May 2006. <http://aspn.activestate.com/ASPN/Cookbook/Python>

[8] Introduction to R Statistical Package. Venables, W. N. , and Smith, D.M. <http://cran.r-project.org/doc/manuals/R-intro.pdf>

[9] TSeries: Package for time series analysis and computational finance. Trapletti, Adrian, and Hornik, Kurt. <http://cran.r-project.org/src/contrib/Descriptions/tseries.html>

[10] MySQL Commands. 05 May 2006. <http://www.pantz.org/database/mysql/mysqlcommands.shtml>

[11] Insert GARCH material

[12] Probably some other r-package documentation

[13] Kendrick, David, Mercado, Ruben, and Amman, Hans. Computational Economics Modeling. Austin: UT Austin Press, 2005.

[14] Kritzman, Mark. Portable Financial Analyst. New York: Wiley Finance, 2003.

[15] Engle, Robert. 2001. GARCH 101: The Use of ARCH/GARCH Models in Applied Econometrics.

[16] Markowitz, Harry. Portfolio Selection. The Journal of Finance, 1952.