Software Architecture Design Document:

Algorithmic Trading System

written by:

Stuart Gordon Reid

Systems Architect

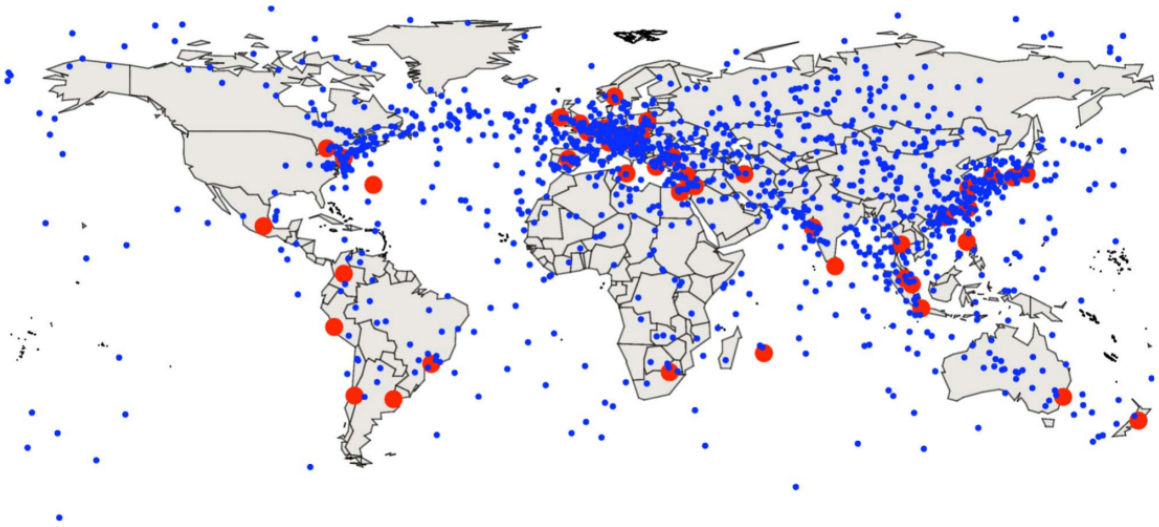
U1006942

for:

Open Source Algorithmic Trading Architectures (OSATA)

(27 November 2013)

V0.5



The blue points in the image above show the places where the network latency for algorithmic trading systems are minimized, and the red points show the locations of large international securities exchanges.

MIT open press: http://dspace.mit.edu/handle/1721.1/62859

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

Revision History

Date	Version	Description
October 10	V0.0	Began researching existing architectures
October 18	V0.1	Started summary of requirements document
October 20	V0.1	Complete summary of requirements document
October 20	V0.2	Began work on conceptual view of architecture
October 25	V0.2	Completed conceptual view of architecture
October 25	V0.3	Began work on structural view and tactics view of architecture
November 1	V0.4	Completed structural and tactics view of architecture
November 3	V0.5	Final version with conclusion, sources, and technologies added

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

Table of Contents

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

1. Introduction and Document Description

1.1. Purpose of this document

The purpose of this document is to define a software architecture for an algorithmic trading system. With respect to the algorithmic trading system, this document will:

- ➔ Describe the software architecture at multiple levels of granularity using UML diagrams and complementary bodies of text and
- ➔ Provide a requirements traceability matrix which illustrates how requirements captured in the requirements document are translated into architectural design decisions.

This document represents parts of an agreement between the Open Source Algorithmic Trading Architectures group, (OSATA), and Stuart Reid, the systems architect. As such, this document serves as a formal contract between the two aforementioned parties. The primary deliverable of this contract with the OSATA architecture.

1.2. Document conventions

This document follows the template prescribed by Dr Fritz Solms of the University of Pretoria
This diagrams in this document are created using the open source diagramming tool, Dia.

1.3. Requirements Traceability

For each tactic and architectural pattern employed reference is made back to the original requirements document for the OSATA algorithmic trading system architecture. The design of this proposed architecture was requirements driven.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

2. Executive summary of the requirements document

An algorithmic trading system has three high level functional requirements: make trading decisions, submit market orders, and manage orders after submission. Yet an algorithmic trading system which satisfies these three requirements may still fail to meet expectations. This is because algorithmic trading systems are also subject to strict non-functional requirements which are often overlooked at the onset of a software engineering project or a system sourcing and selection exercise. Trying to satisfy non-functional requirements ex post facto can become very costly. This article suggests relevant non-functional requirements to consider when building an algorithmic trading system or buying one off of the shelf. It also articulates the responsibility of an algorithmic trading system's architecture in satisfying those requirements.

2.1. Algorithmic trading

Algorithmic trading is often used by buy-side trading institutions as a means of reducing costs, remaining anonymous, improving productivity, executing trades faster and more consistently, and reducing the risk of significant market impact. Because of these benefits, the quantity of securities traded algorithmically has risen significantly since the early 2000's. This march of progress has given rise to a subset of high performance, low latency algorithmic trading systems called high frequency trading systems. Currently a debate over the market impact of these systems is being waged. Some industry players support algorithmic trading and others cite the flash crash as an indication that financial markets are becoming increasingly unstable. This changes how algorithmic trading system architectures should be designed and implemented.

2.2. System architectures

There is still no consensus regarding what a system's architecture is. This article defines a system's architecture as the infrastructure within which application components which satisfy functional requirements can be specified, deployed, and executed. Functional requirements are the expected functions of the system and its components e.g. make trading decisions. Non-functional requirements are measures through which the quality of the system can be measured e.g. make millions of trading decisions per second (performance) and log the audit trail for all trading decisions made (auditability).

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

2.3. Non-functional requirements

Whether initiating a software engineering project to build a new algorithmic trading system, or initiating a sourcing and selection exercise to buy an algorithmic trading system, the following eight non-functional requirements should be taken into consideration:

1. **Scalability** – measures the ability to continue performing under an expanding workload. An algorithmic trading system should be scalable when it comes to adding additional data feeds, the number of markets and instruments traded, the number of supported trading strategies, and the number of concurrent users.
2. **Performance** – measures the amount of work accomplished as compared with the time and resources required to do that work. An algorithmic trading system should be able to process data efficiently and accurately. It should also maintain a high network throughput
3. **Modifiability** – measures the ease with which changes to the system can be made. An algorithmic trading system should be modifiable when it comes to trading strategies, financial reporting, rules for data pre-processing, and message queue structures.
4. **Reliability** – measures the accuracy and dependability of a system to produce correct outputs for the inputs received. Because bugs in algorithmic trading systems could result in financial losses and fines, algorithmic trading systems should be reliable.
5. **Auditability** – measures the ease with which the system can be audited. Because of increasing regulatory pressure on algorithmic trading systems to comply with new laws and standards, they should be auditable from both from a financial and IT point of view.
6. **Security** – measures the safety of a system against criminal activity such as terrorism, theft, or espionage. Trading strategies are proprietary and represent valuable intellectual property; as such the algorithmic trading system should ensure their confidentiality and integrity. Large orders should also be obfuscated to avoid market impact and ensure anonymity.
7. **Fault tolerance** – measures the ability of a system to continue operating properly after a fault or failure. This is similar to reliability, except that the algorithmic trading system should continue to be reliable even after a fault. This is required in order to mitigate the risk of financial losses.
8. **Interoperability** – measures the ease with which the system is able to operate with a diverse range of related systems. This is important for an algorithmic trading system

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

which may be required to interface with order management systems, portfolio management systems, risk management systems, accounting systems, and even banking systems.

Even if an algorithmic trading system satisfies its functional requirements that system will not meet expectations unless it also satisfies its non-functional requirements. As an example consider an algorithmic trading system that makes great trading decisions but is completely inoperable with the organization's risk management and accounting systems. The cost of satisfying the interoperability non-functional requirement may be significant and if the repair were conducted poorly this could impact the performance, scalability, or even profitability of the algorithmic trading system.

2.4. Architectural constraints

In a perfect world the person or organization building or buying an algorithmic trading system would have a complete list of clearly articulated functional and nonfunctional requirements. They would also have enough time and resources to build a system which satisfies those requirements. However, in the real world the ability to satisfy requirements can be constrained. Real world constraints often include:

1. **Organizational constraints –** could include budget, resourcing, system dependency, vendor lock-ins, governance, and other constraints. Such constraints should be identified in advance and proactively managed to ensure the successful implementation of the algorithmic trading system.
2. **Geographic constraints –** the physical locality of the algorithmic trading system will impact system performance. When high performance is a strict requirement an option is to co-locate the algorithmic trading system with the exchange. However, the decision to co-locate may constrain the system in other ways e.g. infrastructure limitations.
3. **Regulatory constraints –** regulatory constraints vary between different countries and sometimes even within countries where state laws exist. Additional regulations governing specific exchanges and securities will also constrain the algorithmic trading system where applicable.

Where non-functional requirements are often universal, constraints are mostly localized. Therefore, constraints on the algorithmic trading system should be analyzed on a case-by-case.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

2.6. Additional requirements

In addition to functional and nonfunctional requirements there are also access and integration requirements. Access requirements define the way in which end-users interact with the algorithmic trading system e.g. traders, system administrators, and auditors. Integration requirements specify how the algorithmic trading system must interact with external systems e.g. exchanges, brokerages, and banks. These interactions are mostly controlled by standard protocols such as FIX, FAST, FIXatdl, XML, and SWIFT or application programming interfaces (API's) such as the Bloomberg Open API.

2.7. Conclusions

Algorithmic trading systems are complicated by strict non-functional requirements, numerous access and integration requirements, localized regulations which govern algorithmic trading, and challenging architectural constraints. Unless these are understood and taken into consideration in the architecture, an algorithmic trading system is unlikely to meet expectations.

2.8. Bibliography

1. Solms F, *What is software architecture?*, University of Pretoria, 2013
2. Cliff D and Northorp L, *The global financial markets: an ultra large scale systems perspective*, the future of computer trading in financial markets review, 2012
3. Avellaneda M, *Algorithmic and High frequency trading: an overview*, New York University & Finance Concepts LLC, 2011
4. Wissner-Gross, A and Freer, C. *Relativistic statistical arbitrage,* Massachusetts Institute of Technology, American Physical Society, 2011
5. Tellefsen Consulting Group, *Algorithmic Trading Trends and Drivers,* 2005

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3. Solution Architecture

3.1. Architectural Overview

As mentioned previously software architecture is defined in this document as the infrastructure within which application components providing user functionality can be specified, deployed, and executed. In order to communicate the architecture a number of complementary views of the envisages software architecture will be presented. These are detailed in the coming pages.

3.1.1. Key architectural views

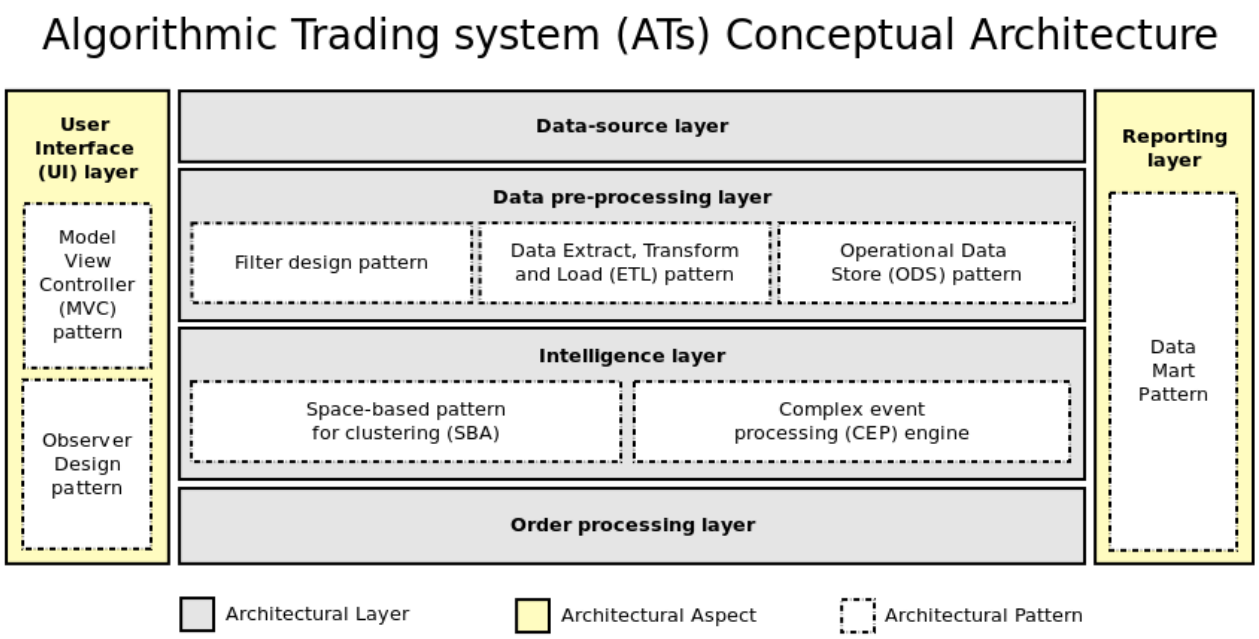
The architectural views presented in this document will include conceptual, structural, tactics, and a reference architecture view. Additional technologies and frameworks which should be taken into consideration when building an algorithmic trading system are also presented. The bodies of text for each of these views will be structured according to the conceptual architecture and described across multiple levels of granularity. This is done to ensure breadth and depth of clarity in presenting architectural decisions.

3.1.2. Requirements driven

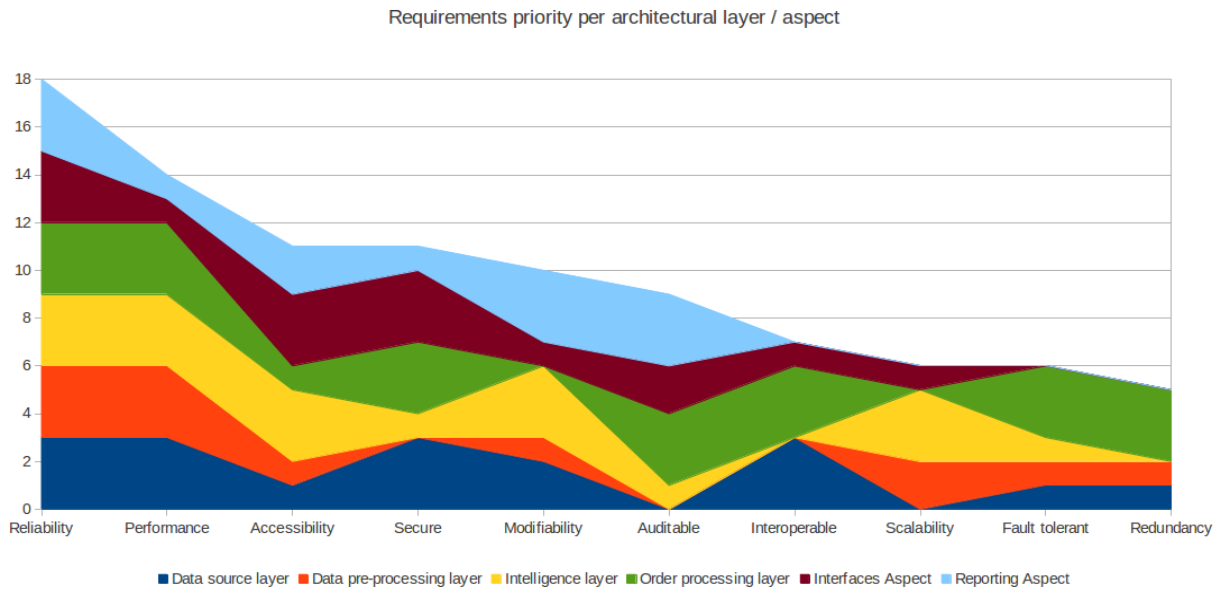
For each view key design decisions will be articulated, rationalized, and mapped back to specific requirements. In so doing, the process followed in producing the architecture is requirements driven. Additionally, at the conceptual view, radar diagram showing the priority of specific quality / nonfunctional requirements are shown. The reason for this is to guide the design decisions in the subsequent more detailed views to satisfy those requirements which are more important for that particular layer first.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2. Conceptual view



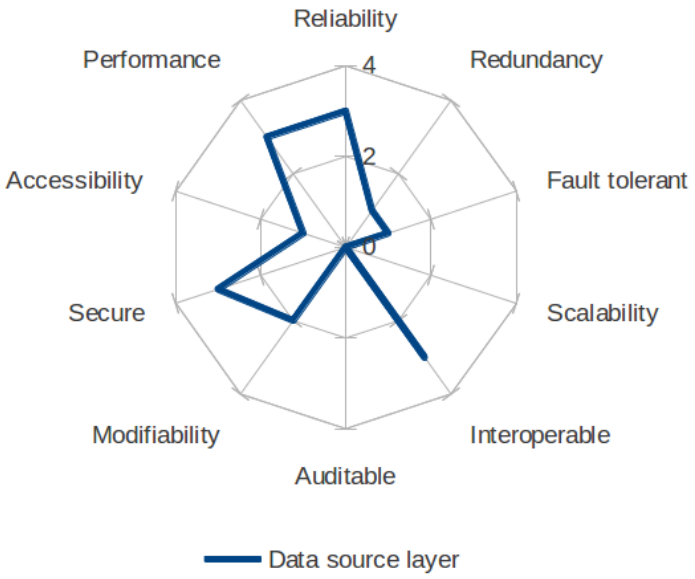
At a conceptual level the proposed architecture consists of four layers (which feed into one another) and two cross-cutting concerns across all four layers. This is illustrated in the previous diagram. For certain layers a set of core architectural patterns have been identified: This graph shows the priorities of each requirement per layer / aspect of the conceptual architecture:



Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.1. Data source Layer

This is a data staging area where data from various external sources is received and data is sent for pre-processing in the data pre-processing layer. Components on this layer manage the performance of the network connections from external data sources. Such architectural tactics are be described in the tactics view later in this document.



Quality requirements radar graph

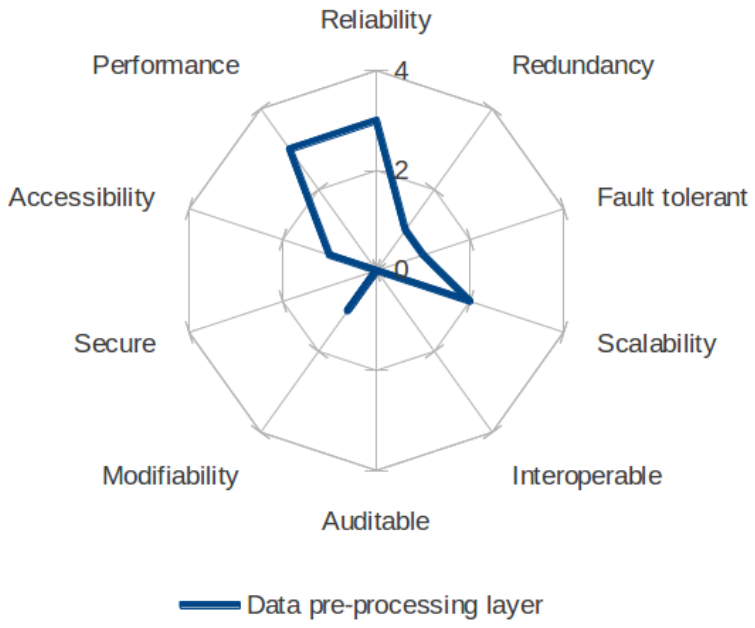
Reliability, performance, security, and interoperability are marked as high priorities for this layer. Modifiability is also marked as a priority for this layer, but it not a high priority as the rate of change experienced in this layer is expected to be less than in others.

3.2.2. Data pre-processing layer

In this layer data is received by the data source layer and prepared for analysis. In this layer three core architectural patterns have been identified. These architectural patterns and specific tactics are described in more detail later on in this document.

1. **Data filtering using the filter design pattern -** filters out irrelevant data. In the context of an ATs this is done to remove data about securities which are not traded. Filtered data is passed into the intelligence layer as a stream of real time events (RTE).
2. **Data extraction, transformation, and loading (ETL) -** extracts filtered data, transforms it into a standard format or schema, and loads that data into the end target. The component realizing this pattern must also derive new data and load it as well.
3. **Operational Data Store (ODS) -** stores relevant data and uses pre-defined business rules to ensure the integrity of that data. This data is then analyzed using a continuous querying languages (CQL) which is implemented as a stream. CQL is used in complex event processing to identify long term and complex events (CE).

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com



Quality requirements radar graph

Reliability and performance are marked as high priorities for this layer. Scalability and modifiability were also marked as priorities for this layer.

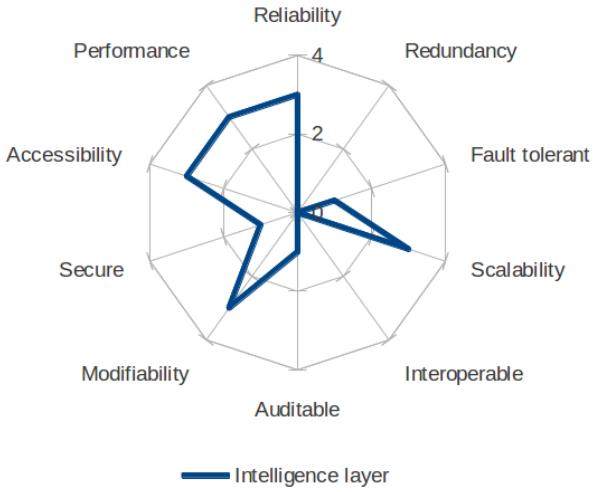
3.2.3. Intelligence layer

This is where the actual algorithmic trading happens. In order to satisfy performance and scalability requirements this layer makes use of the following patterns / tactics.

1. **Space based architecture (SBA) pattern -** a reference architecture which specifies an infrastructure where loosely coupled processing units interact through a shared associative memory. SBA is an event based architecture augmented with the concept of space (shared memory). In the context of algorithmic trading the shared associative memory is responsible for hosting and publishing new events. These could be real time or complex events. Events are picked up by different processing units, and depending on the event one or more processes could be launched to process the event. This either results in no action or a trading order being made.
2. **Complex event processing (CEP) engine -** allows multiple event driven trading strategies to be defined and executed. The process of defining new trading strategies should be as flexible as possible, as strategies could be:
    a. Based on real time events (RTE) or complex events (CE)
    b. Exchange specific e.g. NYSE strategies, JSE strategies, etc.
    c. Security specific e.g. All options, US Equities, Bonds, IT Equities, etc.
    d. Based on artificial intelligence (e.g. Neural Networks) or use simulations
    e. Based on sentiment indicators extracted from financial news
    f. Any other trading strategy or combination thereof

    The CEP engine is elaborated in the tactics architectural view section

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

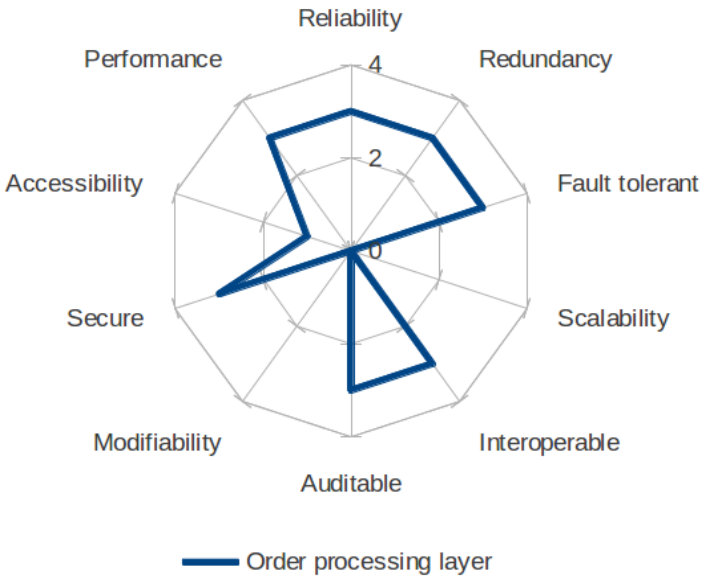


Quality requirements radar graph

Reliability, performance, accessibility, modifiability, and scalability are marked as high priorities for this layer.

3.2.4. Order processing layer

The order processing layer is responsible for receiving trading orders from the intelligence layer, submitting those orders to market exchanges, dark-pools, banks, and brokerages; tracking all open positions through responses from these entities; and logging all orders made onto a data recovery and backup server. In all likelihood this layer would be realized as a standalone order management system. This is described in more detail in the technologies and frameworks view.



Quality requirements radar graph

Reliability, performance, redundancy, fault-tolerance, auditability, interoperability, and security are all marked as high priorities for this layer.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

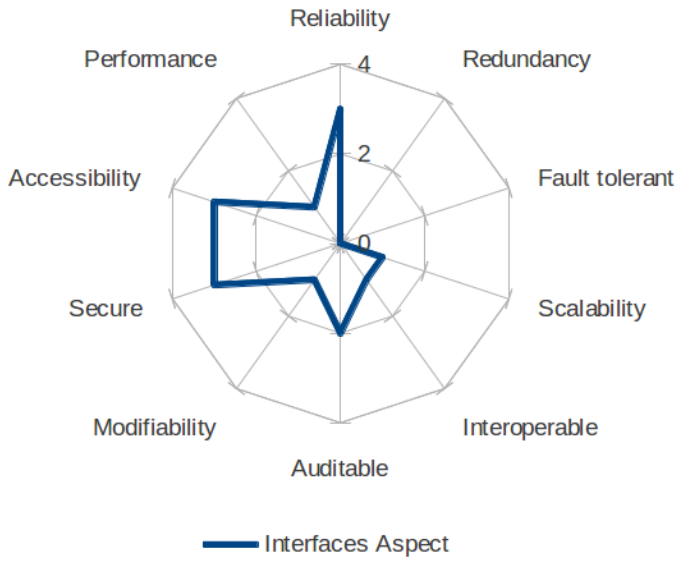
3.2.5. User Interface aspect

The user interface aspect is a cross cutting concern because interfaces will be required to configure and manage each one of the four layers. The following user interfaces are required

1. **Data source interface** - add additional data sources, remove existing data sources, and update sources if some change occurs.
2. **Data pre-processing interface** - define new filters for incoming data, configure the ETL processes, manage the ODS, and define new complex events using CQL.
3. **Intelligence later interfaces** - manage and monitor the processing environment, define new trading rules and strategies, and configure the automated trading system.
4. **Order management interfaces** - monitor submitted orders, track exchange responses, and add, remove, or configure connected exchanges, dark pools, and brokerages.

In addition to these interfaces there will also need to be interfaces which:

1. **Testing interface** - back-test new trading rules and strategies using historical data prior to deploying these onto a production environment
2. **Auditors interface** - generate financial reports which conform to IFRS standards, and list the IT general controls (ITGC's) in the system.



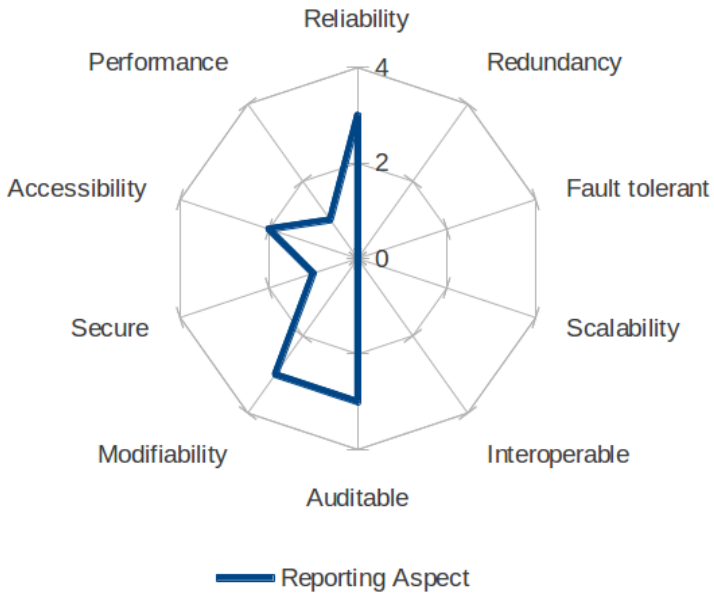
Quality requirements radar graph

Reliability, accessibility, and security are marked as high priorities for this cross cutting aspect. Scalability and auditability were also marked as priorities

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.6. Reporting aspect

The reporting aspect covers all of the aforementioned layers. This aspect should be configurable across each one of the layer through the interfaces mentioned in 4.2.5. The data mart architectural pattern was selected for this aspect to introduce a more flexible approach to reporting. The introduction of the “slice-and-dice” functionalities of a data-mart would allow reports which comply to specific regional reporting standards to be created.



Quality requirements radar graph

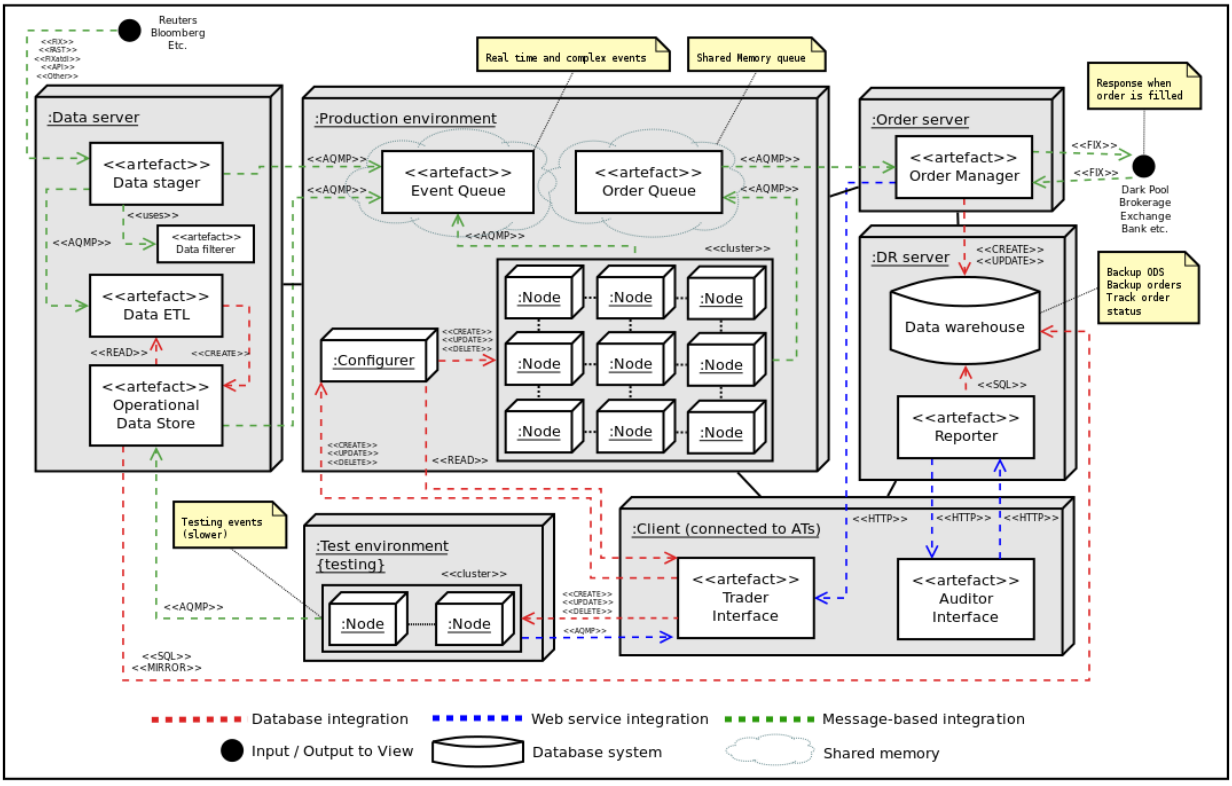
Reliability, auditability, and modifiability are marked as high priorities for this aspect.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.3. Structural view

The structural view of the algorithmic trading system architecture shows the deployment, components, and sub-components of the algorithmic trading system architecture at each level. Below is a diagram diagram showing the physical deployment of the algorithmic trading system onto physical nodes. Each artefact represents an instance of one or more components, these could be deployed as .jar or executable files onto the physical infrastructure.

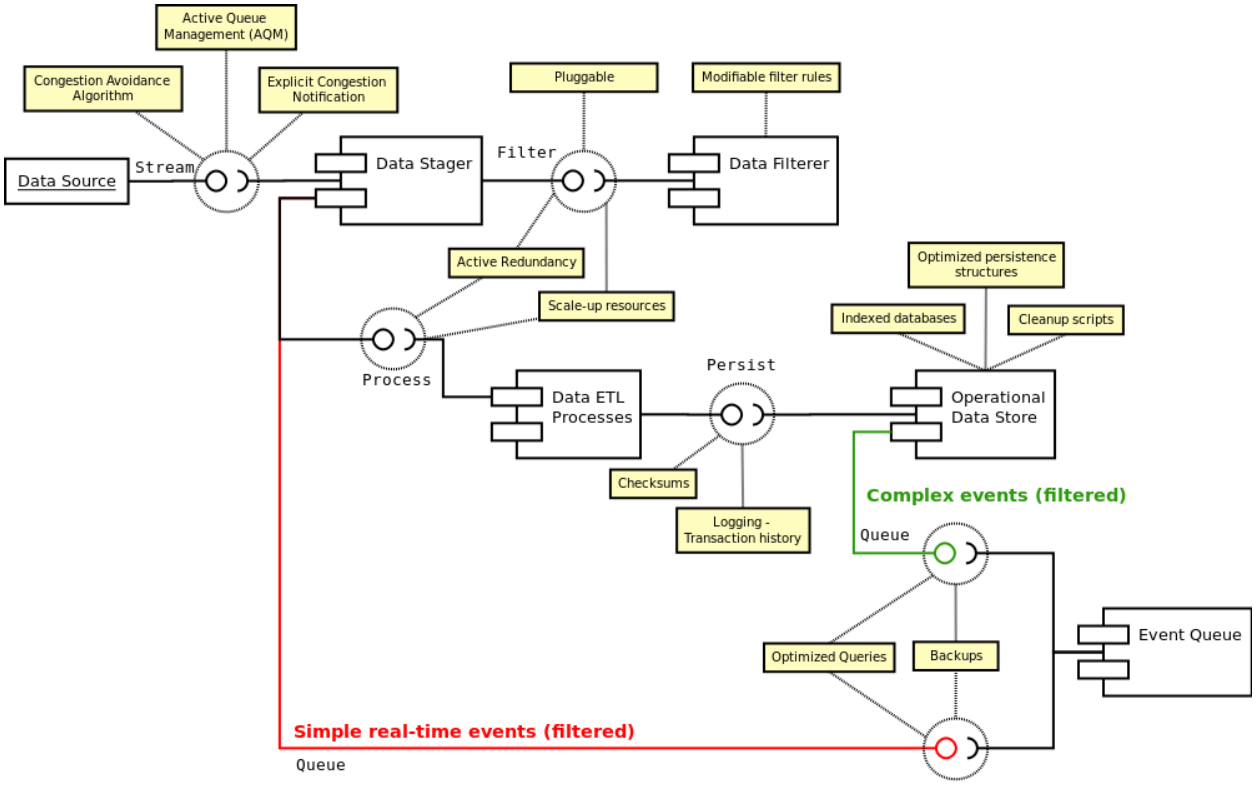
Algorithmic Trading system (ATs) High Level Deployment View



3.2.1. Data source and data pre-processing layer

The data source layer consists of the following components: a data stager which implements the data staging area architectural pattern, a data filter which uses the filter design pattern to distinguish between relevant and irrelevant data, a componentized set of processes called the data ETL (extract, transform, and load), and an operational data store (ODS) for near real-time processing of historical information using CQL to identify complex events.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com



The above diagram shows the component connector view of the data source and data pre-processing layers. The component diagram has been augmented with labels showing various tactics used to satisfying quality requirements on the connectors. These are discussed in the architectural tactics view for each layer. Here, we will simply describe each component

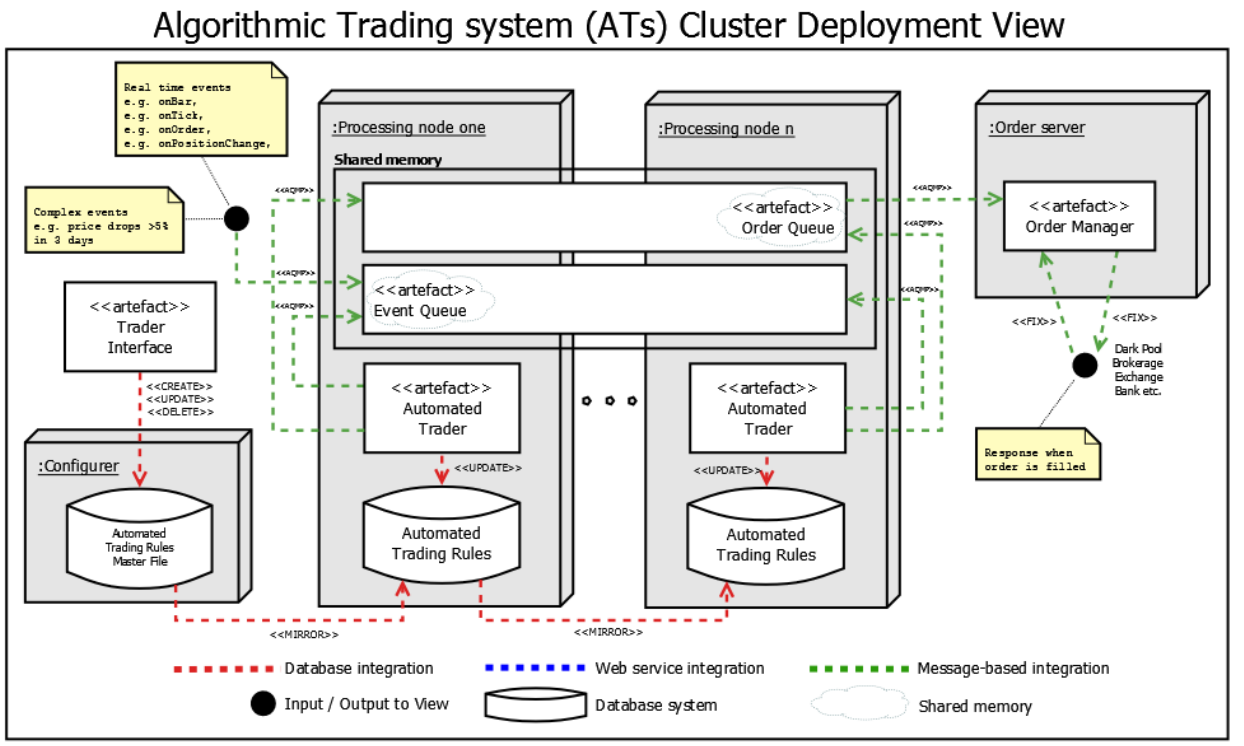
Component / Object	Description
Data Sources	These are not components but rather data streams which feed into the data staging area. These sources are likely to include Reuters, Bloomberg, FIX data service providers, news services, etc.
Data Stager	The data stager is a temporary storage area between the sources of data and a data warehouse or operational data store. Once data has been successfully filtered and loaded into the ODS it is deleted.
Data Filter	The data filter receives data from the data stager and filters that data according to user defined filtering rules. This is done to extract relevant information to improve performance. This component is pluggable in that it can be removed
Data ETL processes	This component encapsulates the processes and technologies needed to extract data from the data staging area, transform it into a standardized format, and load that data into the event queue as real-time events, and into the ODS for further analysis using CQL. This component would support database integration technologies protocols.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

Operational Data Store	An operational data store is an architectural pattern used for processing data quickly in near real-time. An ODS is designed to contain atomic data (such as real time events e.g. transactions or prices). It contains a limited history (only what data is needed). Befor data is removed from the ODS it is stored in Data Warehouse component (DW). Complex events are derived from the data in the ODS using an architectural tactic called continuous querying (CQL).
Event Queue	This event queue is an in-memory queue that sits in shared memory on the processing cluster (part of the intelligence layer).

3.2.2. Intelligence layer

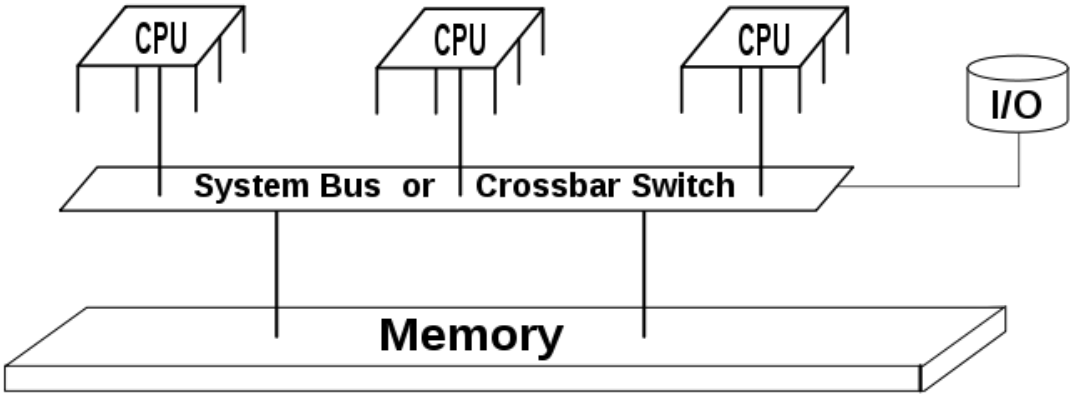
The intelligence layer is based on a Space Based Architecture (SBA). As mentioned in the conceptual view SBA is a reference architecture which specifies an infrastructure where loosely coupled processing units interact through a shared associative memory. SBA is an event based architecture augmented with the concept of space (shared memory). In the deployment diagram below the processing nodes are shown as physical nodes although these could be virtualized. The event queue component encapsulates the concept of space, and the automated trader components contain the processing logic.



Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.2.1. The event queue "component"

As already mentioned the event queue sits in the memory shared across all of the processing units. The diagram below illustrated the concept of shared memory structurally. In the diagram below, think of the "CPU"'s as the automated trader components as these contain the processing logic. As to whether this logic is executed on a CPU or GPU is left to the automated trader to determine.

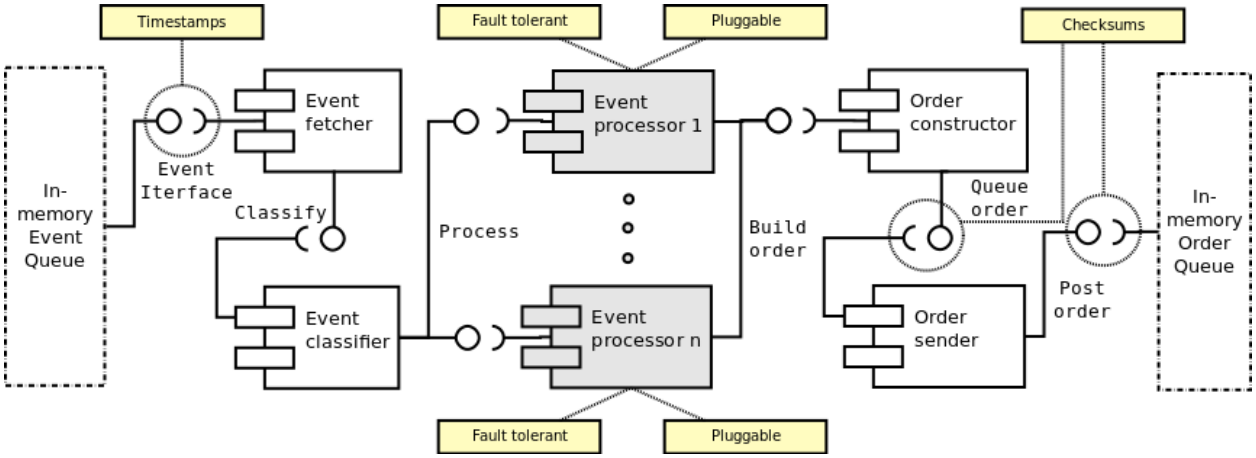


The event queue and the order queues are realized using the disruptor pattern. The disruptor pattern is a relatively new design pattern which has shown promising results in terms of performance and scalability in algorithmic trading. The disruptor pattern is described in more detail in the architectural tactics view of this document.

3.2.2.2. The automated trader component

The clustered automated trader components represent the heart of the algorithmic trading system. These are referred to as the model. The way in which this model is realized could vary significantly depending on the type of algorithmic trading being done and the computational techniques being used. As an example, the functionalities found in an automated trader responsible for trading bonds would be different to an automated trader responsible for trading equities. That having been said, the overall architecture of an algorithmic trading system is event driven, as such, regardless of what type of security is traded an automated trader must take events and convert them into orders. In satisfying this constraint the following base architecture of the automated trader component is suggested.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com



Component / Object	Description
Event fetcher	The event fetcher fetches data from the shared in-memory event queue (space) when it has event processor(s) sitting idle.
Event classifier	This component classifies the event which was fetched according to the type of event. This event plus the classification is then sent to an idle event processor to be analyzed.
Event processor(s)	Event processors contain the business logic for analyzing events. This business logic is updated through an in-memory (local) database of trading rules and strategies. These processors will use computational techniques to process events. Computational techniques used in algorithmic trading include, but are not limited to: • Decision trees based on events and or indicators • Neural networks trained to identify patterns • Simulations e.g. monte carlo simulation • Mathematical formulae for pricing securities e.g. derivatives pricing • Expert systems which codify human knowledge • Fuzzy logic systems used to make decisions • Many other computational finance techniques
Order constructor	Where an event matches a trading rule or strategy which results in an order being made, the relevant data is sent to the order constructor which passes it to the order sender.
Order sender	The order is put onto the in-memory orders queue.

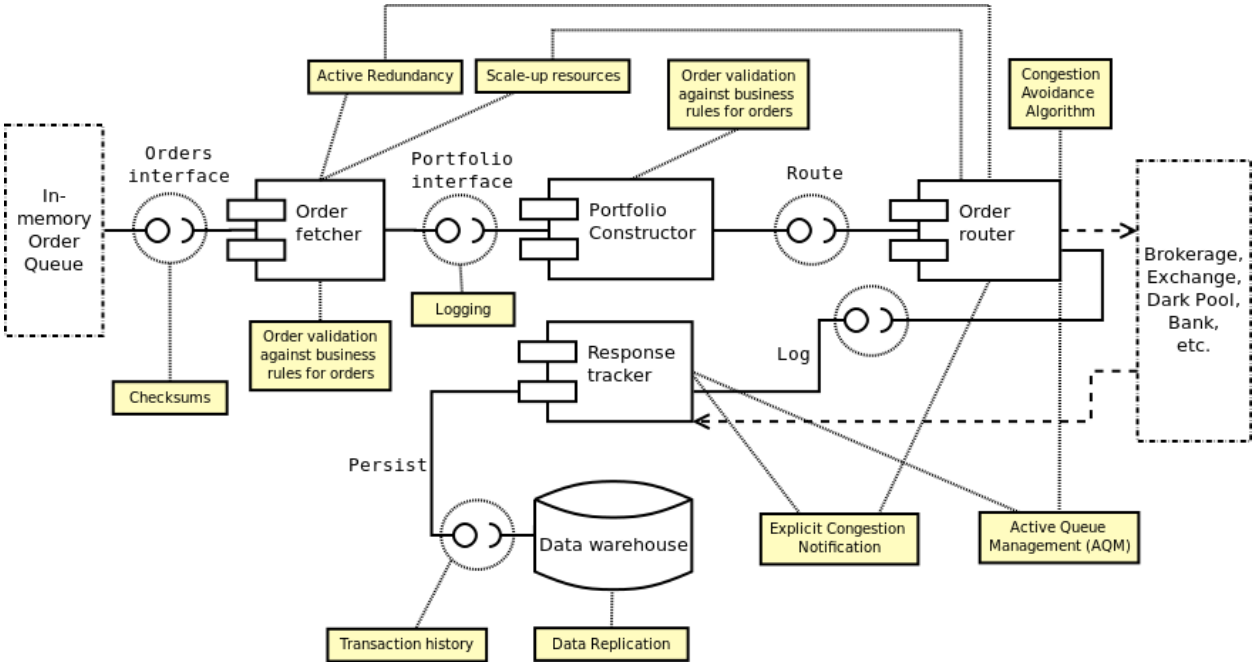
3.2.2.3. The automated trading rules database

As mentioned above the automated trading rules database sits in the local memory of the processing node. In here the business logic and configuration for the automated trading system is kept. A master database sits on a configuration node. This database then replicated its content across each of the other slave databases using a daisy-chain.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.3. Order processing layer

The order processing layer is interesting because it would likely operate out of an existing, customized order management system (OMS). This would probably be cheaper than building a new order management system but could constrain the performance of the architecture if the customized OMS does not meet the quality requirements of the algorithmic trading system. In essence the architecture of this layer is quite simple from a component perspective, however in order to meet the strict quality requirements of this layer, many architectural tactics would need to be employed across all sub-components:



This layer is also responsible for portfolio construction. This component needs to maintain a model of the portfolio currently being held by the algorithmic trading system. According to the algorithmic trading system review these techniques can be used in this component:

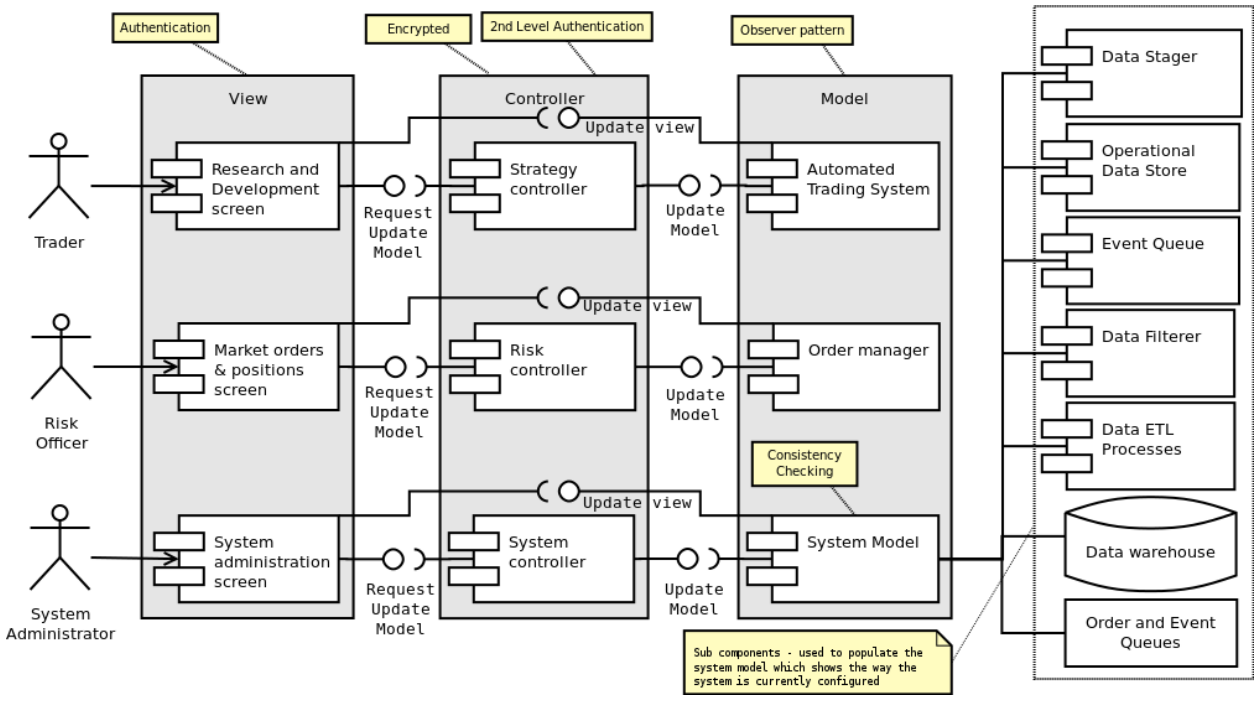
1. Rule based models
    a. Equal position weighting
    b. Equal risk weighting
    c. Alpha driven weighting
    d. Decision tree weighting
2. Optimization models

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.4. User Interface aspect

The user interface aspect is built on top of the Model View Controller (MVC) reference architecture. This architecture separates the representation of information from the user's interaction with it. In the context of algorithmic trading systems this decoupling is important to enforce security and access controls. The three components that make up the MVC are of course the model component, view component, and control component.

1. **View** - is a representation of the model which the user can see and, through the controller, interact with (e.g. update the model)
2. **Controller** - contains the business logic and relevant controls required to change the underlying model. It receives requests to change the model from the view component
3. **Model** - a representation of the underlying “model”. In this example it is considered to be a data defined state of the algorithmic trading system.



As you can see from the component diagram above, there are three instances of the MVC in the algorithmic trading system. These are aligned to the three distinct ways in which different users may interact with the system:

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

1. **Research and development screen -** this is used by traders and researchers to create, update and delete algorithmic trading strategies. New algorithms are created and back tested on the test environment, and then deployed onto the production environment. Likewise, algorithms in production can be modified, back-tested, and then updated.
2. **Market orders & positions screen -** this is monitored by risk officers and traders. They can manually adjust orders put through the system, manually close off risky positions, and in the event of system failure shut down the system using a kill switch.
3. **System administration screen -** this is used by administrators of the algorithmic trading system to configure the individual system components through a centralized system model. This model would act similarly to a configuration management system and ensure that configurations across the components were correct and consistent.

As mentioned previously the MVC architecture uses the observer design pattern. This is detailed in the tactics view of the user interface aspect. The other aspects shown pointing to the model, view, and controller grey blocks simply means that the tactics are used across all three instances. These include two step authentication and encryption

3.2.6. Reporting aspect

The reporting aspect governs the interaction between auditors and the system. The MVC reference architecture was not applied in this context because the auditor should not interact with the underlying model (data, logs, etc.) simply the presentation thereof. As such, the data-mart architectural pattern was selected. A data-mart is defined as the access layer between the users and the data warehouse. This data-mart should be realized using a flexible business intelligence reporting tool that allows the user to define their own reports and reporting structures. This flexibility will help with:

1. **Compliance constraints -** where specific standard reports are required and
2. **Interoperability -** through file-based integration with other systems

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.4. Tactics view

An architectural tactic is a means of satisfying a quality requirement. As such it bridges the quality requirements document and the architectural design. Specific tactics were selected and 'applied' to components and connectors to meet one or more nonfunctional requirements.

3.2.1. Data source and data preprocessing layers

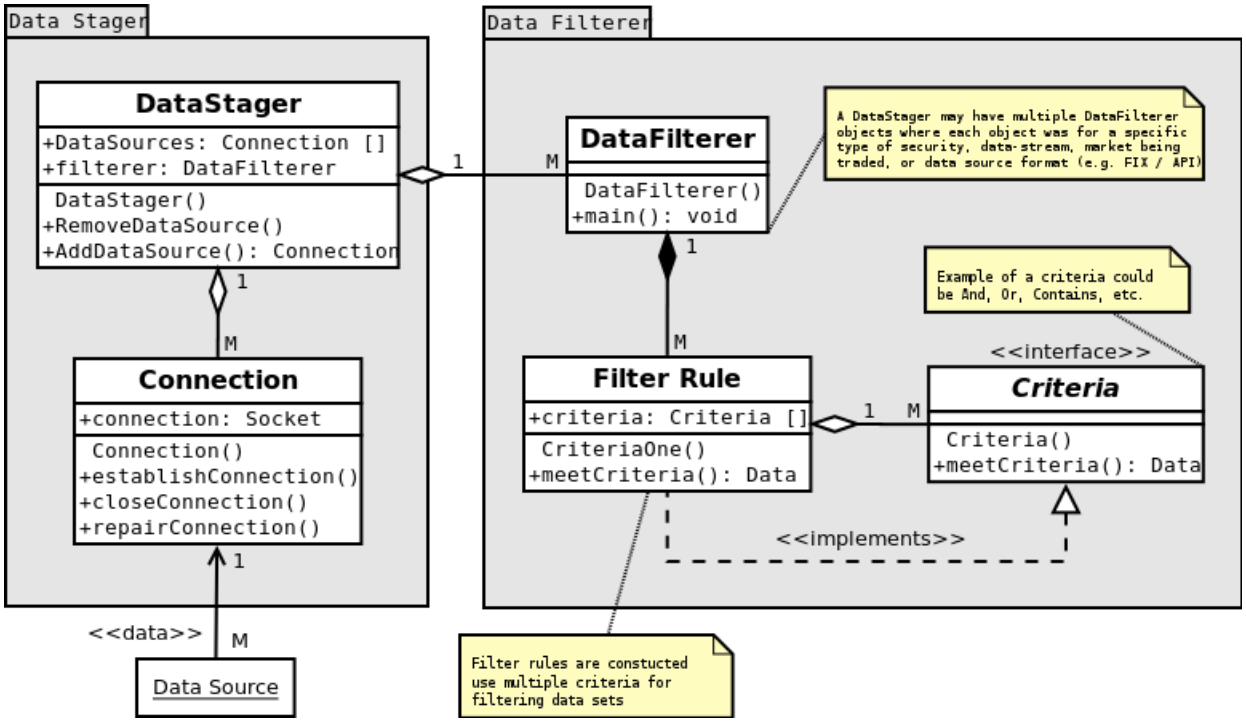
3.2.1.1. Data stager / data staging area component

Design decision	Nonfunctional Requirements
Use a congestion avoidance algorithm on TCP connections - TCP uses a congestion avoidance algorithm that uses tactics from schemes such as the additive increase/multiplicative decrease (AIMD) scheme and the slow-start scheme	Performance Reliability
Use active queue management (AQM) on incoming queue's - AQM is the arbitrary reorder of network packets inside the transmit buffer of a network interface controller. It reduces network latency and improves performance of queues TCP/IP.	Performance Reliability
Implement an explicit congestion notification on incoming queue's - Explicit Congestion Notification (ECN) is an optional extension of TCP IP. Standard TCP/IP networks signal congestion by dropping packets this avoids that but must be supported by the network.	Reliability Fault tolerance
Make the data filter pattern pluggable (easily removed) - If all data is to be processed the data filter would be an overhead. If only a small portion of data is to be processed then the data filter would improve performance through pre-processing	Modifiability

3.2.1.2. Data filter component

Design decision	Nonfunctional Requirements
Deploy data filter component using active redundancy - Active redundancy is a design concept that increases operational availability and that reduces operating cost by automating most critical maintenance actions.	Fault tolerance Performance Reliability
Make sure resources can scale upwards - Resources used should be commodities with additional physical "space" for improved capacity and processing power	Performance Scalability
Enable modifiable filter rules through filter design pattern - This is detailed on the next page	Modifiability

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com



3.2.1.3. Data ETL component

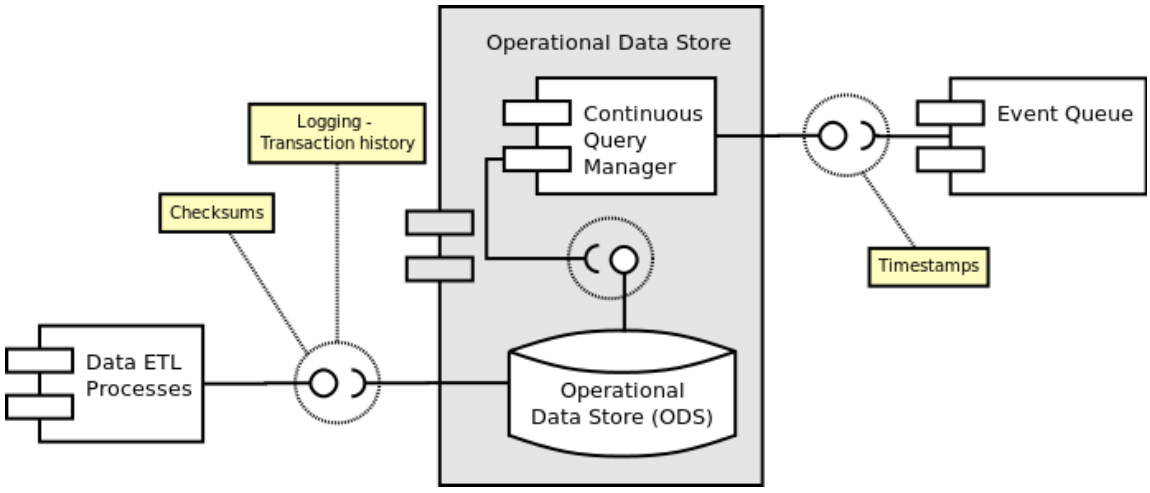
Design decision	Nonfunctional Requirements
Encapsulate the ETL data patterns into a component to bridge the data stager and operational data store - See http://en.wikipedia.org/wiki/Extract,_transform,_load for more info	Reliability
Deploy data ETL component using active redundancy - Previously explained	Fault tolerance Performance Reliability
Make sure resources can scale upwards - Previously explained	Performance Scalability
Generate checksums for data send into the ODS - A checksum is a small-size datum computed from an arbitrary block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage	Reliability Fault avoidance
Maintain transaction log of all data sent into the ODS - A transaction log is a history of actions executed by a database management system to guarantee ACID properties over crashes or hardware failures. Physically, a log is a file of updates done to the database, stored in stable storage.	Recoverability Auditability

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.1.4. Operational Data Store

Design decision	Nonfunctional Requirements
Generate checksum from data received and compare to checksum sent from the ETL to identify any data issues - Previously explained	Reliability Fault avoidance
Index databases for fast querying - This is a data structure that improves the speed of data retrieval operations on a database table. Indexes are used to quickly locate data without having to search every row in a database table	Performance
Optimize persistence structures for fast querying - The structure of database tables has a large impact on performance, there should therefore be optimized to improve performance	Performance
Run hourly backup scripts of all data from the ODS into data-warehouse	Fault tolerance Autiability
Run daily clean-up scripts which remove irrelevant data provided it is backed up onto the data warehouse - Only a certain 'window' of data will be used by the automated trader components in the intelligence layer. Therefore, making sure that the ODS has only 'just enough' data will improve performance of queries	Performance
Employ continuous query language (CQL) for ODS analysis - This is detailed below	Functional requirements Performance

Continuous querying is a technique which has been used in a number of complex event processing systems e.g. GemFire[1]. With CQL a query is registered with the server. The query then runs continuously against server cache changes, and the user receives events for any changes to the query result set.



[1] For more information see: http://community.gemstone.com/display/gemfire/Continuous+Querying

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

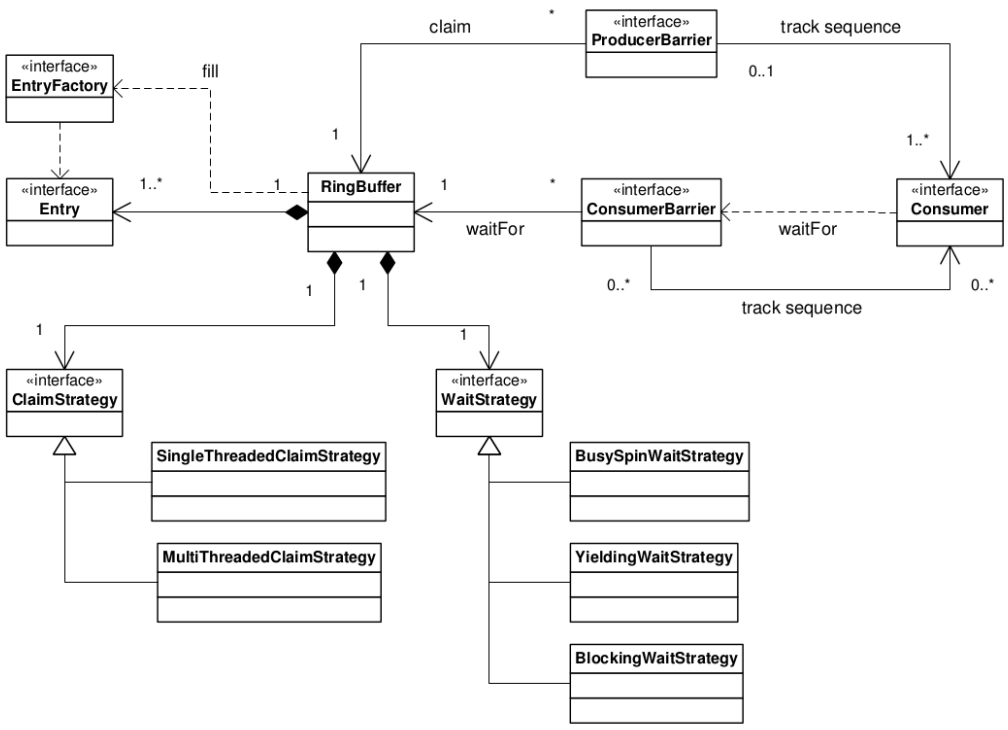
3.2.2. Intelligence layer

3.2.2.1. In memory queue's

Design decision	Nonfunctional Requirements
Deploy the queues between components in-memory	Performance
Use disruptor pattern for high performance and scalability - This is detailed below for more information see http://martinfowler.com/articles/lmax.html	Performance Reliability

The disruptor pattern is a relatively new concurrency design pattern which provides the ability to pass a message onto other threads. Three characteristics of the disruptor pattern differentiate it from other concurrency design patterns (e.g. BlockingQueue):

1. User of disruptor defines how events are stored. This is achieved by extending the entry class and using a factory to do pre-allocation. This allows memory reuse or the entry could contain a reference to an object
2. Two phase process for posting events:
    a. Claim a slot in the ring buffer this provides the user with an entry
    b. Entry is committed making it visible to consumer threads
3. Consumer is responsible for tracking messages that have been consumed from the ring buffer



[2]

[2] This class diagram is from the LMAX architecture paper:

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.2.2. Automated trader component

Design decision	Nonfunctional Requirements
Annotate all events with timestamps - Time stamps show the exact time an event was created. This is done so that expired events can be kicked out by the processors, this is done to avoid trading on outdated data. The number of events 'kicked out' should be monitored as an indication of performance	Reliability
Each event processor 1 through n through be fault tolerant	Reliability Fault tolerance
Each event processor 1 through n should be pluggable - Each event processor should be able to handle all types of events and should be deployed on commoditized hardware to make scalability of the cluster as easy as possible.	Scalability Performance
Orders are annotated with checksums for accuracy - Previously explained	Reliability Fault avoidance

3.2.2.3. In-memory trading rules database

Design decision	Nonfunctional Requirements
Use in-memory database to store and update rules for event processors in the automated trading systems - An in memory database relies on main memory for data storage. Main memory databases are faster than disk-optimized databases since the internal optimization algorithms are simpler and execute fewer CPU instructions. Accessing data in memory eliminates seek time when querying the data, which provides faster and more predictable performance than disk	Performance Reliability Accessibility
Event processors can use in-memory database for calculations - The database can be used to do queries on data	Performance Ease-of-use

3.2.3. Order processing layer

3.2.3.1. Order fetcher component

Design decision	Nonfunctional Requirements
Evaluate orders using annotated checksums - Previously explained	Reliability Fault avoidance
Validate orders against business rules for orders e.g. max / min volume quantities and or amounts traded - Specific trading rules which define the "do's and dont's" for the algorithmic trading system should be defined to evaluate orders and remove ones which are non-compliant with the rules	Reliability Fault tolerance Security
Deploy the order fetcher component using active redundancy - Previously explained	Fault tolerance Performance Reliability
Enable the order fetcher to easily scale up computing resources	Performance

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

- Previously explained	Scalability

3.2.3.2. Order router component

Design decision	Nonfunctional Requirements
Deploy the order router component using active redundancy - Previously explained	Fault tolerance Performance Reliability
Enable the order router to easily scale up computing resources - Previously explained	Performance Scalability
Make use of active queue management to improve performance - Previously explained	Performance Reliability
Make use of congestion avoidance algorithms on TCP connections - Previously explained	Performance Reliability
Enable explicit congestion notification - Previously explained	Reliability Fault tolerance

3.2.3.2. Portfolio construction component

Design decision	Nonfunctional Requirements
Order validation against standard business rules - previously explained	Reliability Fault tolerance

3.2.3.3. Response tracker component

Design decision	Nonfunctional Requirements
Make use of active queue management to improve performance - Previously explained	Performance Reliability
Make use of congestion avoidance algorithms on TCP connections - Previously explained	Performance Reliability
Enable explicit congestion notification - Previously explained	Reliability Fault tolerance

3.2.3.4. Data warehouse

Design decision	Nonfunctional Requirements
Maintain a transaction history for the database - Previously explained	Fault tolerance Auditability
Enable full replication of data in the data-warehouse - To avoid loss of information in the data-warehouse, the database(s) should be replicated onto additional hardware	Fault tolerance

3.2.3.5. Additional Consideration

It is most likely that the order management layer for the algorithmic trading system will be satisfied by the organizations presumably pre-existing order management system. As such, it is important to assess the suitability of that external order management system in terms of it's ability to satisfy both the functional and nonfunctional requirements.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.4. User Interface aspect

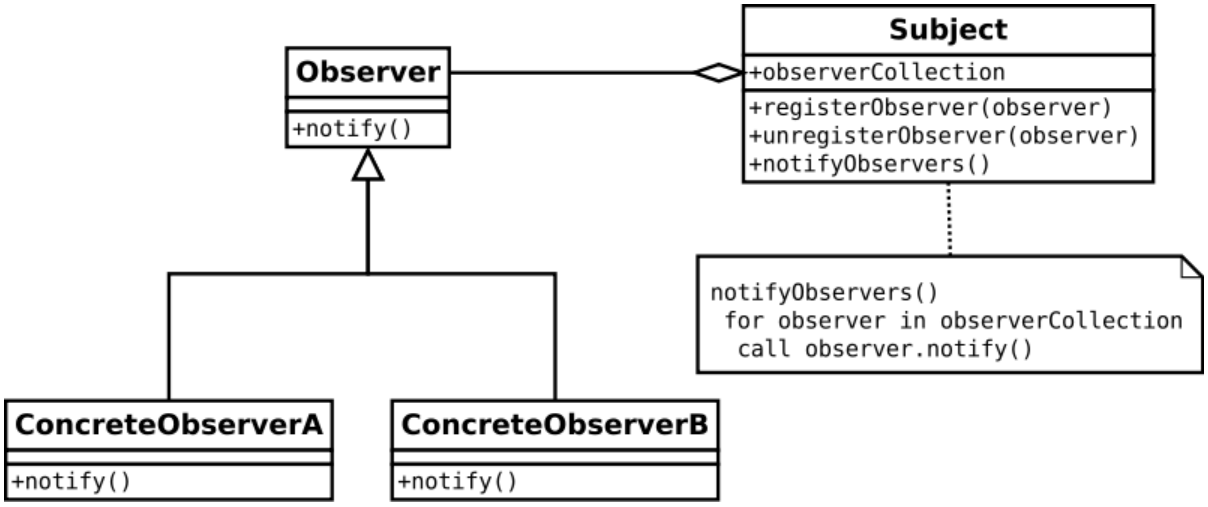
3.2.4.1. View components

Design decision	Nonfunctional Requirements
Use first stage authentication for users connecting to the ATs - The user is authenticated with the view component of the algorithmic trading system using a unique username and password. Then the view itself is authenticated with the controller using a separate unique name and password. This is enabled by the choice of MVC because of the decoupling between the user and his / her interaction with the system	Security Accessibility
Use encrypted channels between the end user and the ATs - Encryption is the process of encoding messages in such a way that third parties cannot read it, but only authorized parties can. This is done to mitigate the security risks of connective over a network to the ATs	Security

3.2.4.2. Model components

Design decision	Nonfunctional Requirements
Make use of the observer design pattern - This is detailed below	Performance

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. In the context of an algorithmic trading system, the system model and order manager will notify the observers (views) of any changes.



Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.4.3. Controller components

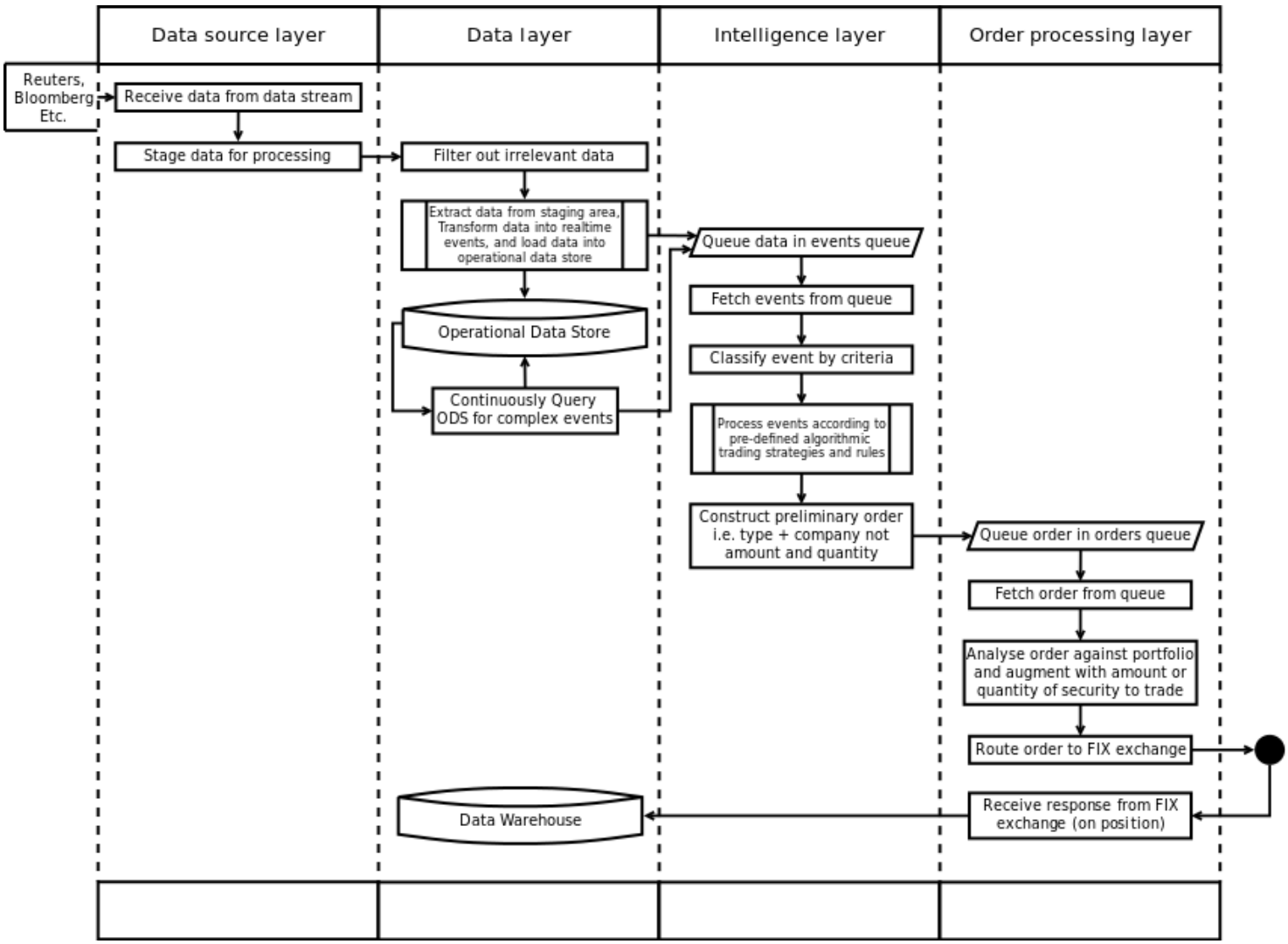
Design decision	Nonfunctional Requirements
Use second stage authentication between controllers and models - The user is authenticated with the view component of the algorithmic trading system using a unique username and password. Then the view itself is authenticated with the controller using a separate unique name and password. This is enabled by the choice of MVC because of the decoupling between the user and his / her interaction with the system	Security
Use encrypted channels between controllers and models - Described previously	Security
Check consistency between system model and system components - Checking the consistency of the system model and the system components at particular points in time could prevent faults	Reliability Security

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.5. Behavioral view

3.2.1. End-to-end system flow chart

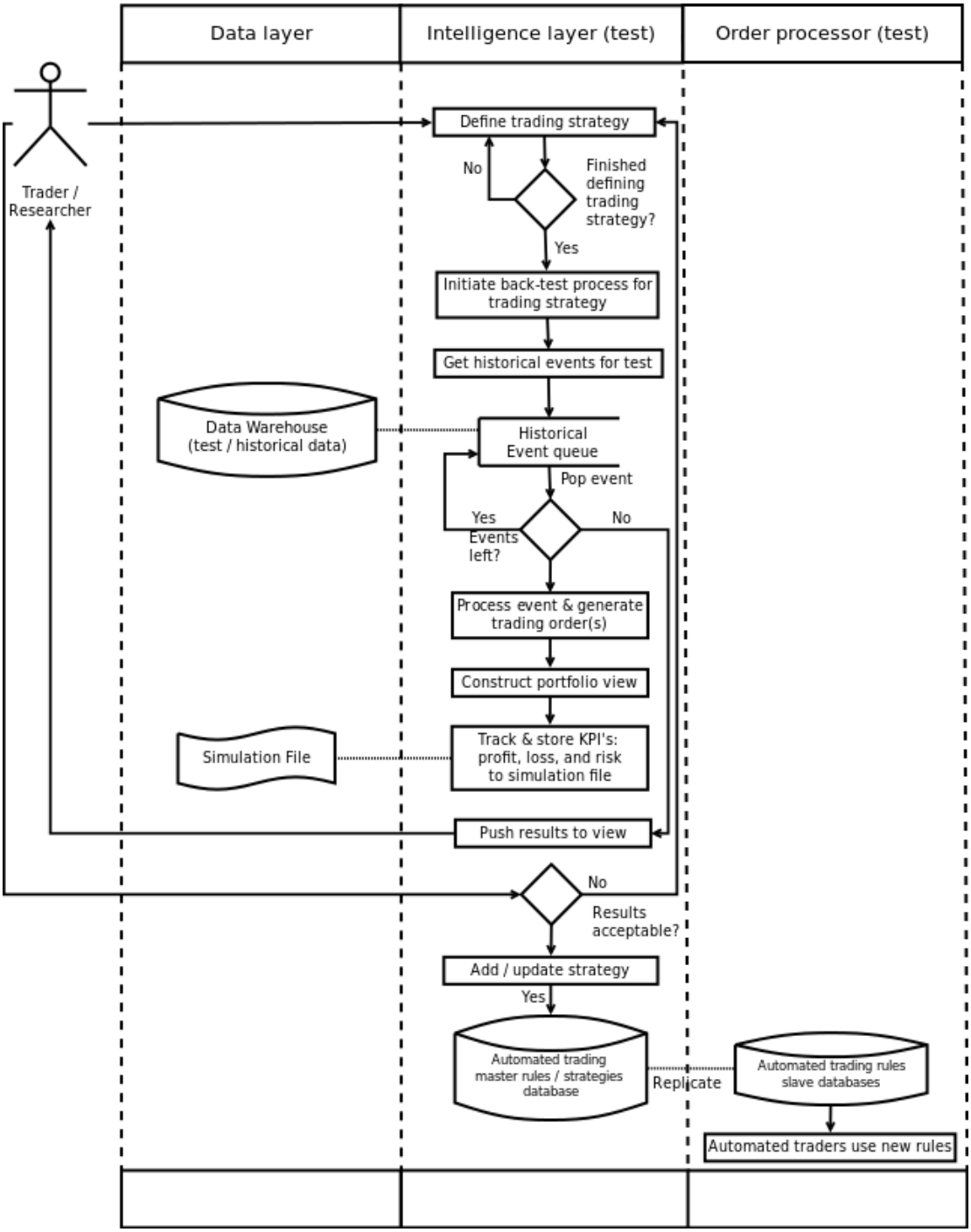
The below swim-lane diagram shows the core activities in each layer of the architecture and how they interact. In this diagram the event-driven nature of the algorithmic trading system shines through quite nicely. A more detailed breakdown of each process into particular sequence diagrams between components is beyond the scope of this assignment as it begins to border on application design.



Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.2. User perspective: research and development

One aspect of the architecture which until this point has not been mentioned in much detail is the role of the testing server and the way in which a person developing algorithmic trading strategies would interact with the algorithmic trading system. The below diagram illustrates this.



Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.6. Reference architectures used

In the architecture of the algorithmic trading system a number of reference architectures have been used. Reference architectures are 'template' architectures which can be used for a wide range of different systems with similar architectural requirements.

3.2.1. Intelligence layer - SBA

The intelligence layer makes use of the space based architecture (SBA) pattern deployed within a clustered environment. SBA is a reference architecture which specifies an infrastructure where loosely coupled processing units interact with one another through a shared associative memory called space. SBA was chosen the hope of achieving near-linear scalability of processing resources. SBA is based on the blackboard pattern and uses many of the principles of Representational State Transfer (REST), service-oriented architecture (SOA) and Event-driven architecture (EDA), as well as elements of grid computing.

3.2.1.1. Components of SBA

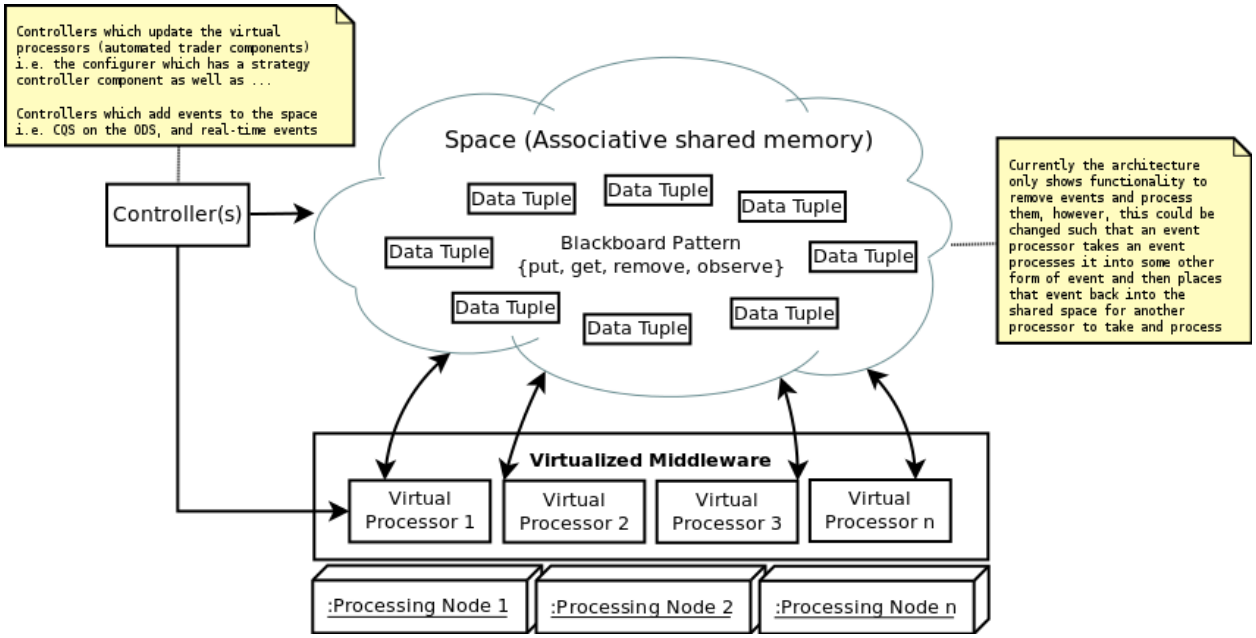
Applications are built out of a set of self-sufficient units, known as processing-units (PU). These units are independent of each other, so that the application can scale by adding more units. Processing units can interact with space by putting, getting, removing, and observing. In the context of the algorithmic trading system architecture, space is the shared in-memory event queue. The processing units are called processing nodes and contain automated trader components which contain the business logic for processing events. These are deployed in a clustered environment which requires a virtualized middleware. In the section on additional considerations and technologies we propose the use of the Apache River project for this.

3.2.1.2. Link back to requirements

The SBA reference architecture was selected because it has been used to develop stateful, high performance applications using the tuple-space paradigm which are also nearly linearly scalable. In other words, SBA is a good reference architecture for satisfying strict scalability, reliability, performance, and flexibility non functional requirements which aligns well to the nonfunctional requirements of an algorithmic trading system.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.1.3. Diagram of SBA reference architecture (not UML)



The diagram above shows the conceptual way in which the SBA pattern works. As stated in the comment, the currently ATs architecture only mentions taking events out of space and processing then, however this could be extended such that a processor can take an event out of space, process it to a point and then place it back onto space to be processed further by a different type of processors. This works in theory, however, I suspect it may be more efficient to have each processor fully capable of processing an event to completion regardless of how many processes that may spawn.

3.2.2. User Interface aspect - MVC

The reference architecture used in the user interfaces aspect is the Model View Controller (MVC). MVC is used to introduce a controller between the user and the system so as to reduce the risk of human error through appropriate controls thereby improving the security of the overall architecture. Security is a high consideration in the user interfaces aspect because of the high susceptibility of an algorithmic trading system to fraud. If a hacker were to gain access to the system they could not only cause system damage but also financial damage to the organization using the algorithmic trading system.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.2.1. Diagram of the Model View Controller

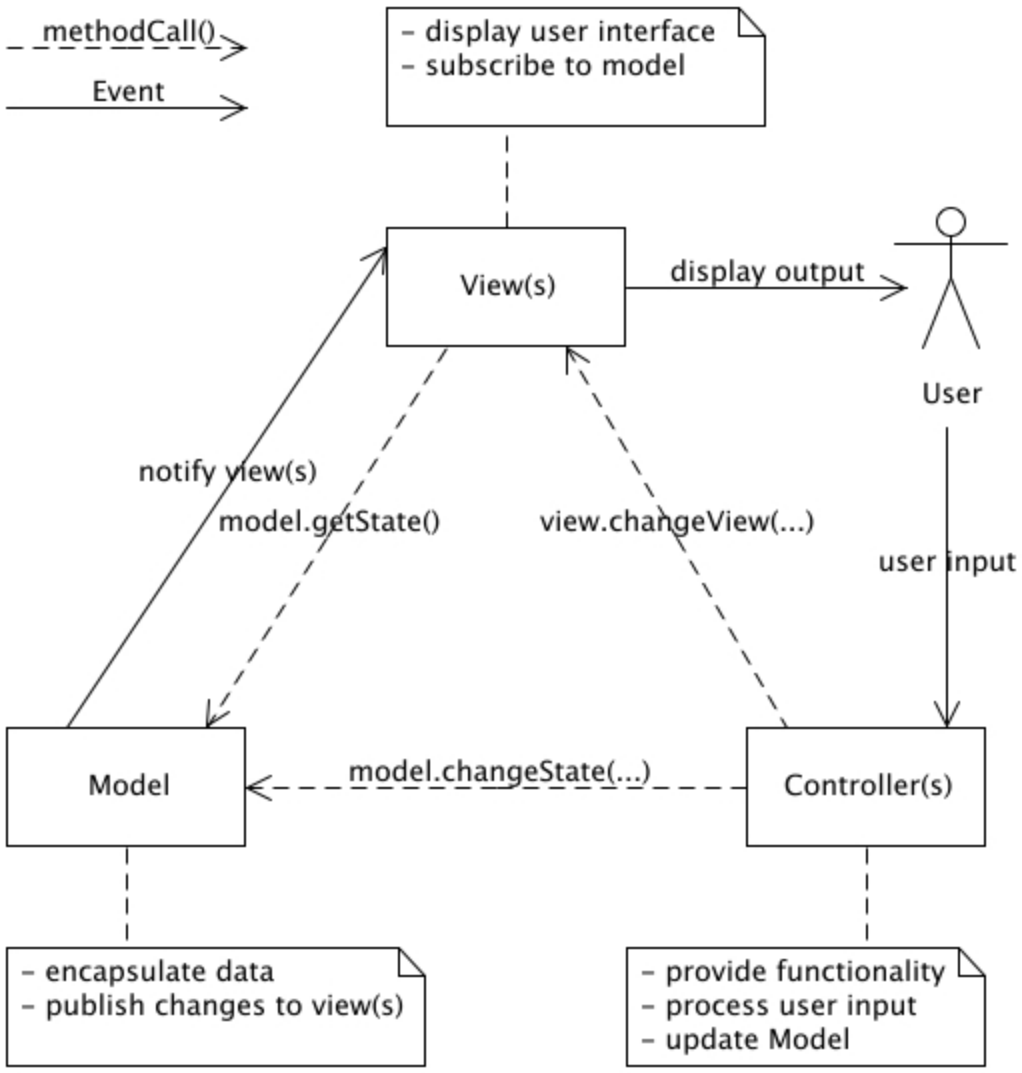


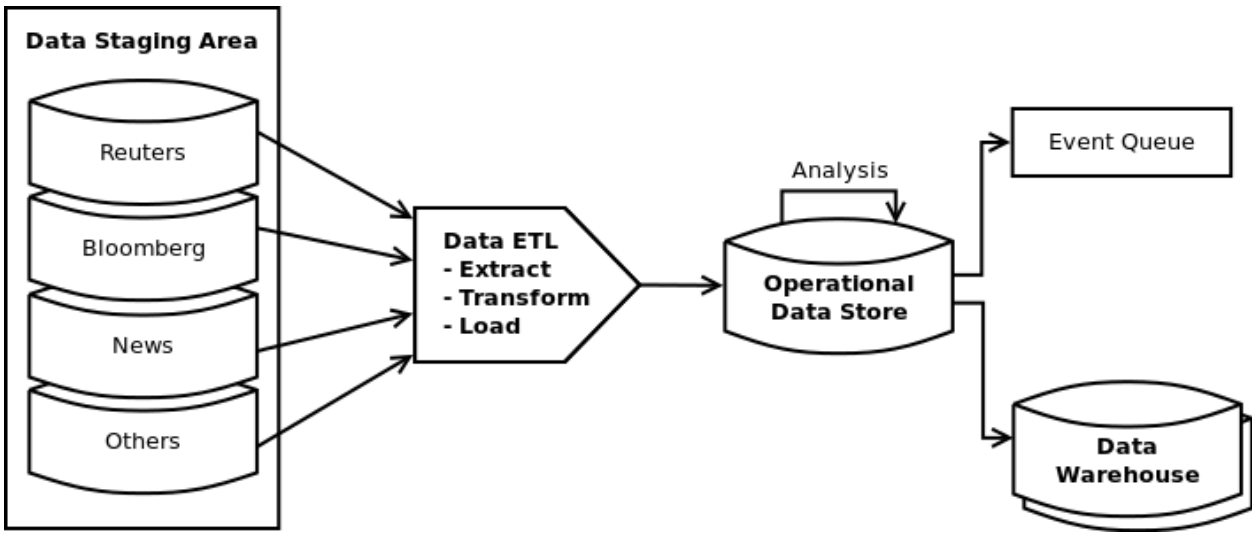
Image source:
http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/images/mvc3.jpg

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

3.2.3. Data source Layer - ETL + ODS + DW

Data in the algorithmic trading system is managed using standard components and patterns found in enterprise systems. The possibility exists that this may be insufficient to meet the data requirements of the system. In this event, a distributed file system such as hadoop could be used to replace the operational data store and the data warehouse. The current architecture is shown below, and hadoop is discussed in further detail in the additional considerations and technologies section.

3.2.3.1. ETL, ODS, and DW Pattern (not UML)



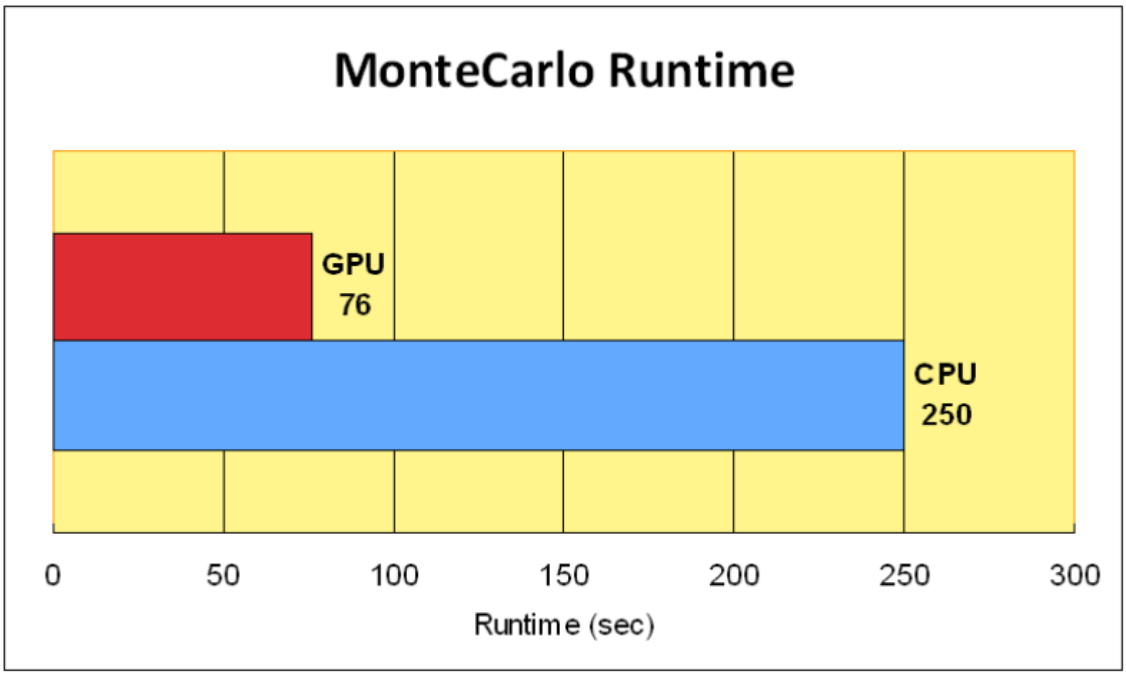
Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

4. Additional architectural considerations and technologies

The following technology should serve as considerations on a case-by-case basis when using the algorithmic trading system architecture.

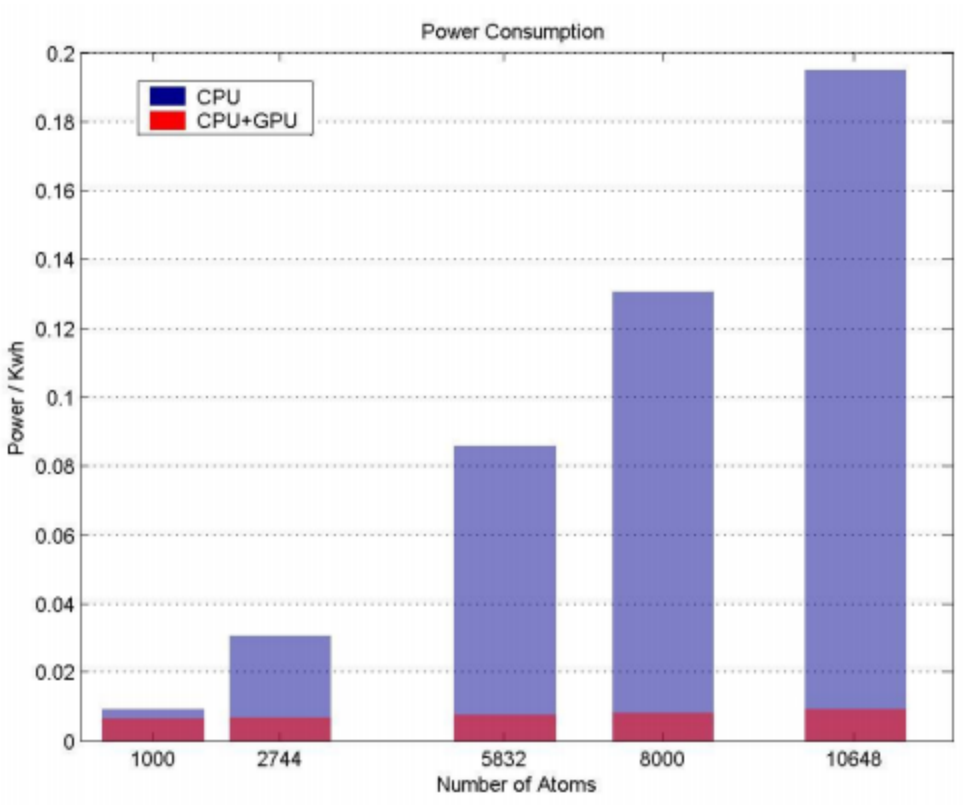
4.1. OpenCL and CUDA support for Quantitative Finance

Much research has gone into investigating the applications of CUDA and OpenCL technologies in quantitative and computational finance modelling. These technologies enable systems to make use of the graphics card to do calculations in addition to the CPU. Research has shown impressive performance increases in financial modelling problems that can be parallelised at reduced operating costs. Example applications include pricing of derivatives using monte carlo simulations, calculating co-variances in large two-dimensional matrices, and enabling artificial intelligence algorithms. Below are some results:



This diagram shows the speedup achieved in performance in using a GPU for Monte Carlo simulations over a traditional CPU

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com



This diagram shows the remarkable improvements in power consumption

4.2. AlgoTrader deployed as the automated trading component

AlgoTrader is an open source trading platform that allows customers to automate complex, quantitative trading strategies in forex, options, futures, stocks, ETFs and commodities markets. It has a robust, open-source architecture, allowing customization for customer-specific needs. In order to reduce efforts associated with implementation times, it may be worthwhile to investigate deploying instances of AlgoTrader on each processing node in the Intelligence layer.

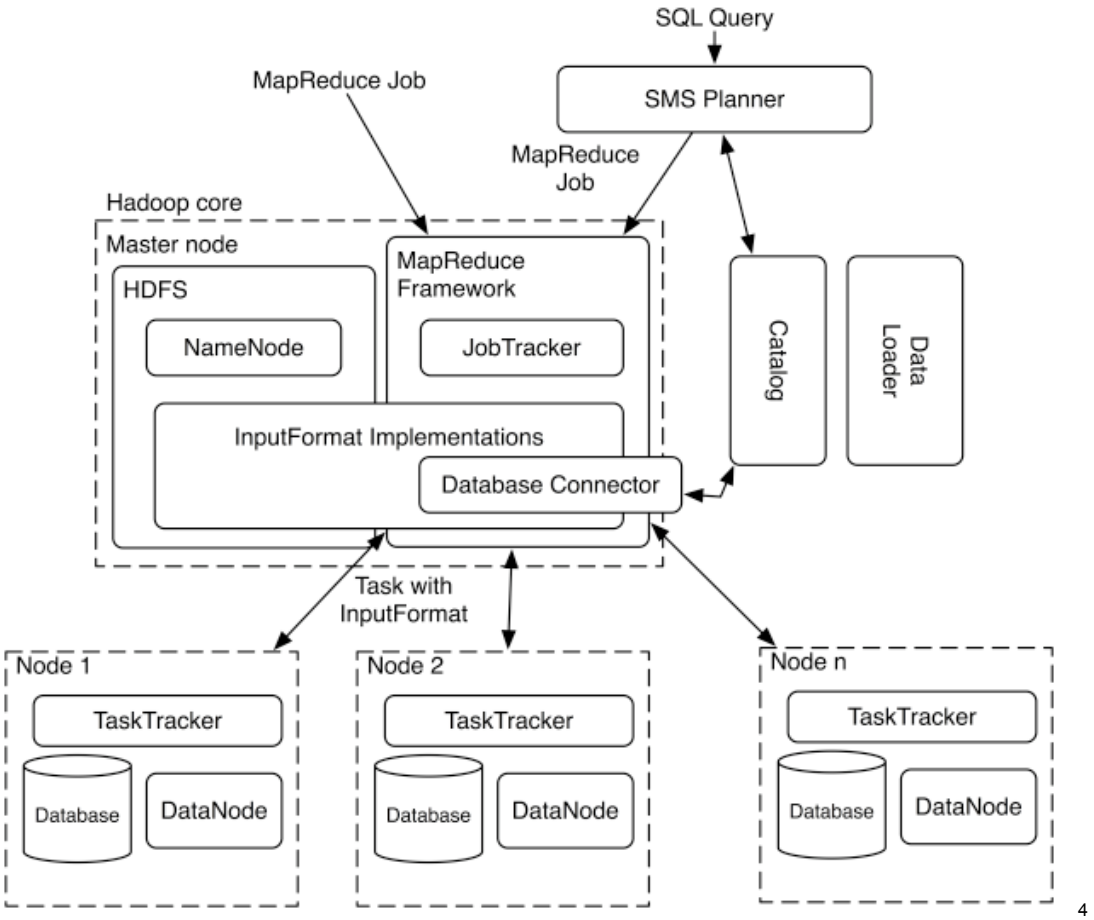
4.3. Distributed file systems for pervasive logging requirements

If data requirements cannot be met by the proposed architecture then the space based clustered architecture could be augmented to include a distributed file system such as Hadoop. A similar problem has been experienced historically by Deep Value, a proprietary algorithmic trading platform who also turned to Hadoop. Hadoop is an open-source software framework for storing and large scale processing of data-sets on clusters of commodity hardware. Hadoop consists of the following main components[3]:

[3] Source: wikipedia http://en.wikipedia.org/wiki/Hadoop

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

1. **Hadoop Common** - contains libraries and utilities needed by other Hadoop modules
2. **Hadoop Distributed File System (HDFS)** - a distributed file-system that stores data on the commodity machines, providing very high aggregate bandwidth across the cluster.
3. **Hadoop YARN** - a resource-management platform responsible for managing compute resources in clusters and using them for scheduling of users' applications.
4. **Hadoop MapReduce** - a programming model for large scale data processing. This based on the MapReduce algorithm which takes large problems and breaks them down into smaller component.



[4]

The hadoop architecture

Note however that Hadoop is typically only used for batch processing which may present a challenge to achieving the high performance processing required for algorithmic trading.

[4] Source of image: http://hadoopdb.sourceforge.net/guide/hadoopDB-arch.png

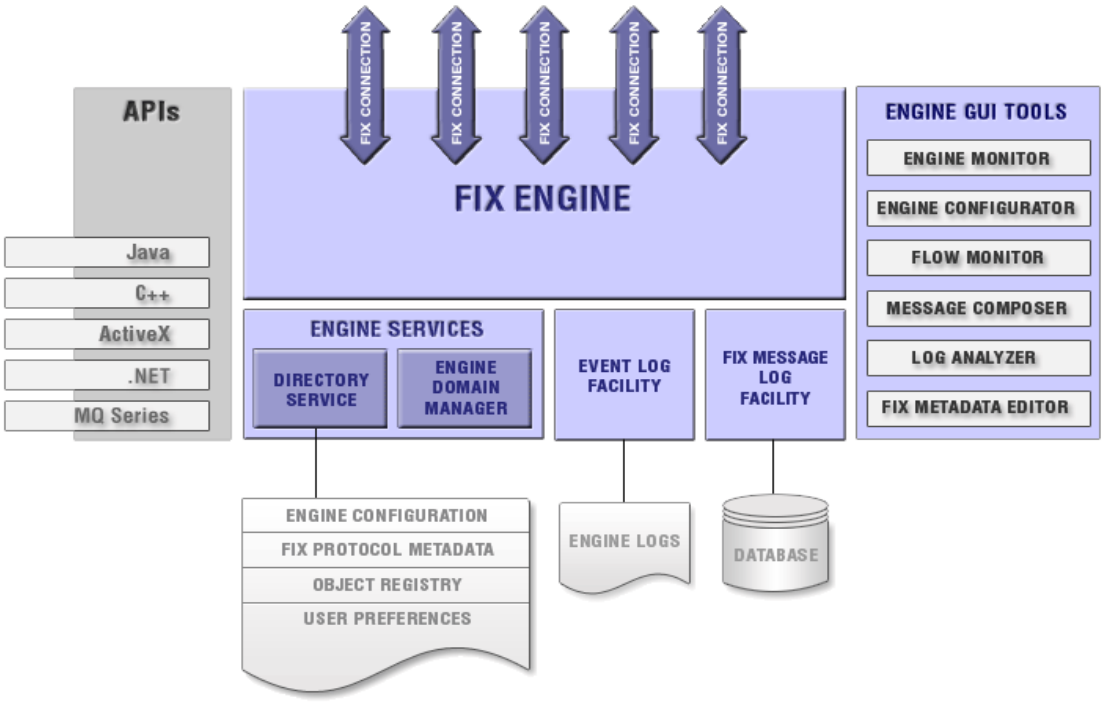
Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

4.4. Application Programming Interfaces

While difficult to illustrate in UML2.0, each one of the components discussed in this document should expose an Application Programming Interface (API). This should be done to try and improve the overall interoperability of the algorithmic trading system with other commonly found enterprise systems e.g. accounting systems, risk management systems, and portfolio management systems.

4.5. FIX, FIXatdl, and FAST

In satisfying integration requirements between exchanges and the algorithmic trading system, the FIX, FIXatdl, and FAST protocols should be supported by the order management system. The standard way of doing this is to purchase a FIX engine, which is responsible for connectivity management, session maintenance, and FIX message handling. The FIX Engine supports multiple connections to different exchanges, can be scaled with a deployment, and is designed with message throughput and performance in mind. The high level architecture of this technology is shown below:



Other open source implementations of the FIX protocol do exist and may be worth investigating is there are cost constraints on the project.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

4.6. Apache River / JINI Project

When building the intelligence layer a potential project which could be leveraged is the Apache River project, sometimes referred to as the JINI Project. River is a toolkit to build distributed systems with. The basic communication between client and service is based on RPC. It contains a number of base services for distributed event processing, locking, leasing, and transactions. The Apache River project is java based and at a high level consists of the following components:

	Infrastructure	Programming Model	Services
Base Java	Java VM	Java APIs	JNDI
	Java RMI	JavaBeans	Enterprise Beans
	Java Security	...	JTS
Java + Jini	Discovery/Join	Leasing	Printing
	Distributed Security	Transactions	Transaction Manager
	Lookup	Events	JavaSpaces™ Service

Figure AR.2.1: Jini Architecture Segmentation

An architecture specification is available at http://river.apache.org/doc/specs/html/jini-spec.html

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

5. Conclusion

The proposed architecture has been designed to satisfy the generic requirements identified in the OSATA architectural requirements document. In general algorithmic trading systems are complicated by three factors which vary differ with each implementation:

1. Dependencies on external enterprise and exchange systems
2. Challenging nonfunctional requirements and
3. Evolving architectural constraints

The proposed software architecture would therefore need to be adapted on a case-by-case basis in order to satisfy specific organizational and regulatory requirements. As well as overcome specific regional constraints. That having been said, the proposed architectural patterns, tactics, and technologies act as a foundation for designing a software architecture which would cater to these specific requirements.

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com

Sources

In general Wikipedia served as a constant input into this architecture document and other web-based sources. The following sources all contained information that was used in the design of this architecture.

Barbarians at the Gate HFT - http://queue.acm.org/detail.cfm?ref=rss&id=2536492#!
Map reduce Algo - http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/
Distributes algos in NoSQL DB - http://highlyscalable.wordpress.com/2012/09/18/distributed-algorithms-in-nosql-databases/
In Stream Big Data processing - http://highlyscalable.wordpress.com/2013/08/20/in-stream-big-data-processing/
GenFire GIFS (Intel) - http://www.gemstone.com/pdf/GIFS_Reference_Architecture_Trading_Systems.pdf
Algo trader CEP - http://www.algotrader.ch/algotrader/architecture/
Algo trader UML - http://doc.algotrader.ch/uml/AlgoTrader-UML-Model.pdf
CISCO Algo Speed HFT - http://www.cisco.com/web/strategy/docs/finance/c22-658397-01_sOview.pdf
CISCO Networks for HFT - http://www.cisco.com/web/strategy/docs/finance/c11-600126_wp.pdf
CISCO The right road to HFT - http://www.cisco.com/web/strategy/docs/finance/High_Frequency_Trading_special_report.pdf
CISCO Microbursts issue - http://www.cisco.com/web/strategy/docs/finance/Dow_Jones_Article_June_2010.pdf
In memory - http://stackoverflow.com/questions/3045164/choosing-a-distributed-shared-memory-solution
Disruptor pattern - http://trading.aurorasolutions.org/over-6-million-transactions-per-second-in-a-real-time-system-an-out-of-the-box-approach/#!
Commodity Computing - http://en.wikipedia.org/wiki/Commodity_computing
In memory data grids -http://www.infoq.com/articles/in-memory-data-grids
TIBCO CEP - http://www.tibco.com/products/event-processing/complex-event-processing/default.jsp
Google Rockteady CEP - http://google-opensource.blogspot.com/2010/09/get-ready-to-rocksteady.html
Esper CEP -http://esper.codehaus.org/
FIX Best practices -http://www.fixtradingcommunity.org/mod/file/view.php?file_guid=43018
CUDA for CF evaluation - http://cse.unl.edu/~che/slides/cuda.pdf
Using GPU's for CF - http://www.nag.com/Market/nagquantday2009_UsingGPUsforComputationalFinance_Mike_Giles.pdf
Introduction - http://www.jcuda.de/tutorial/TutorialIndex.html
Wikipedia CUDA - http://en.wikipedia.org/wiki/CUDA
SQL and CUDA - http://www.cs.virginia.edu/~skadron/Papers/bakkum_sqlite_tr.pdf
BOINC - https://boinc.berkeley.edu/wiki/GPU_computing
TREX CEP - http://home.deib.polimi.it/margara/papers/trex.pdf
GPU Clusters - http://www.ncsa.illinois.edu/~kindr/papers/ppac09_paper.pdf
Petascaling commodity onto exascale with GPU's - http://www.nvidia.com/content/GTC/documents/SC09_Matsuoka.pdf
R on GPU - http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/
Architecture - http://hadoop.apache.org/docs/stable/hdfs_design.html
Yahoo tutorial - http://developer.yahoo.com/hadoop/tutorial/module2.html#rack
Hadoop + CUDA - http://wiki.apache.org/hadoop/CUDA%20On%20Hadoop
Map reduce - http://wiki.apache.org/hadoop/MapReduce
Stack IQ - http://hortonworks.com/wp-content/uploads/2012/06/StackIQ_Ref_Architecture_WPP_v2.pdf
Linux Clusters (Network tech) - http://www.tldp.org/HOWTO/Parallel-Processing-HOWTO-3.html
Cascading - http://en.wikipedia.org/wiki/Cascading and http://www.cascading.org/
Terracotta (in memory + Hadoop) - http://terracotta.org/
HDFS Scalability - http://developer.yahoo.com/blogs/hadoop/scalability-hadoop-distributed-file-system-452.html
Hadoop scalability - http://www.manamplified.org/archives/2008/11/hadoop-is-about-scalability.html
In memory accelerator for Hadoop (100x faster) - http://www.gridgain.com/products/in-memory-hadoop-accelerator/
HBase home - http://hbase.apache.org/
Benchmark TIB/RV vs SonicMQ - http://www.cs.cmu.edu/~priya/WFoMT2002/Pang-Maheshwari.pdf
JMS vs RabbitMQ - http://java.dzone.com/articles/jms-vs-rabbitmq
AMQP, MQTT or STOMP - http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html
AMQP Uses (JPM) -http://www.amqp.org/about/examples
MQ Shootout -http://mikehadlow.blogspot.com/2011/04/message-queue-shootout.html
Rabit MQ - http://www.rabbitmq.com/documentation.html
MVC - http://www.devx.com/assets/articlefigs/13110.jpg

Contact: http://www.stuartreid.co.za or stuartgordonreid@gmail.com