A Componentized Architecture for Dynamic Electronic Markets

Benny Reich Israel Ben-Shaul

Department of Electrical Engineering
Technion — Israel Institute of Technology
Technion City, Haifa 32000, Israel
{reich@tx,issy@ee}.technion.ac.il

Abstract

The emergence and growing popularity of Internet-based electronic market-places, in their various forms, has raised the challenge to explore genericity in market design. In this paper we present a domain-specific software architecture that delineates the abstract components of a generic market and specifies control and dataflow constraints between them, and a framework that allows convenient pluggability of components that implement specific market policies. The framework was realized in the GEM system. GEM provides infrastructure services that allow market designers to focus solely on market-issues. In addition, it allows dynamic (re)configuration of components. This functionality can be used to change market-policies as the environment or market trends change, adding another level of flexibility to market designers and administrators.

1 Introduction and Motivation

Inspired by the growth of the Internet, electronic commerce (EC) has grown rapidly over the last few years and has become a major Internet application domain. EC encompasses various profit-oriented activities such as purchase of goods and services over the network, banking, and business-to-business trade. EC has grown tenfold in 1997 ($8 billion), and Forrester [1] researchers predict that in year 2002, a staggering amount of $327 Billion will be traded between companies.

One particular EC segment that has gained interest recently is Internet-based markets, in which traders buy and sell assets in exchange for money or other valuable commodities. More specifically, by Internet-based electronic markets we mean that: 1. The market is directly accessible through the public Internet. This is in contrast with other markets that allow indirect access through an Internet-based brokerage agency but are themselves accessible only through a private network. 2. The market is entirely automated, with no human market-makers, as in conventional stock-exchanges and auctions. 3. Traders, i.e., the individuals that access the market, may be human or automatic.

We intentionally refrain from specifying exactly what kind of assets or exchanged commodities are used. Indeed, market mechanisms have been shown to be effective in a wide range of domains, including predicting elections (e.g., IOWA stock market experiment [2]), resource allocation in distributed systems (e.g., Challenger [3], Popcorn [4]), market-places for agents (e.g., Artificial Markets Project [5], Kasbah [6]), as well as more traditional applications such as on-line auctions (e.g., AuctionBot [7]).

Network-based markets are spreading fast in the commercial sector too, both among companies that extend their traditional markets to operate over the network as well as among new companies with network-only auctions, (e.g., OnSale auctions [8], eBay auctions [9]). In addition to traditional market models, these companies also invent new models that are effective for online auctions.

Besides the variety in application domains, various types of market mechanisms [10, 11] exist. They can be classified into two basic types: single-sided auctions and double-sided auctions [12]. In a single-sided auction there is a single buyer and many sellers or a single seller and many buyers. Examples of such auctions include the English open outcry auction in which the auctioneer calls a price and bidders raise their hands for acceptance; the Dutch auction, in which the auctioneer starts at a high price and lowers it until someone accepts; or the Vickrey [13] auction where bids are sealed and the highest bidder wins at the second bidder's price. In a double-sided auction there are many buyers and many sellers, and every buyer can be also a seller. Examples include the continuous double auction, in which bids and asks are matched in the order received, and the sealed (clearinghouse) double auction, where bids and asks are collected for a predetermined time interval and are matched at the end of the interval according to price and arrival order.

From the economic point-of-view, there are critical differences between the various markets in terms of

economic performance and the ability to analyze and predict market behavior. However, we claim that from the *system point-of-view*, it is possible to identify and abstract a high-level generic structure, or architecture, that is common to all market applications. The key to a flexible market framework is to combine the generic architecture with fine-grained decomposition into components that can be independently programmed, where each component tailors a distinct aspect of the market mechanism. As an example, a single-sided auction can be treated as a double-sided auction with the number of sellers restricted to 1; a continuous auction can be treated as a special case of a clearinghouse auction with a clearing period set to to be every time a new bid/ask is entered; and so forth.

The Global Electronic Market (GEM) system addresses these issues by providing a generic market framework and infrastructure, along with specifications of component interfaces that need to be implemented and plugged into the framework in order to instantiate an operational market.

Another desirable property for a market framework is the ability to support *dynamic configuration* of its policies, i.e., to enable runtime modifications and/or replacement of components, both for experimentation as well as for providing a mechanism for adapting behavior to changes in the environment. This capability is particularly important for Internet-based markets, in which both system-parameters (e.g., network bandwidth and host loads) as well as economic parameters (e.g., changes in trading volume) are not predictable and thus may require dynamic adjustments.

We emphasize the need for programmability of the framework, as opposed to parameterization. The latter approach is used by the AuctionBot [7] project, which addresses some of the same issues addressed by GEM by offering a general auction architecture that can be parametrized to provide different auctions. Instead of having general rules that are implemented by a fixed set of parameters, GEM defines a loosely coupled framework in which the market mechanism is distributed among many autonomous programmable components. This allows for more fine-grained modifications of the market mechanism. It is important to note, however, that the components themselves are not intended to be reusable across various frameworks, but are components with pre-determined interfaces, chosen according to their role in the market architecture.

Finally, as its name suggests, the long-term goal of the GEM project is to enable construction of world-wide electronic markets. Increasing the number of traders and the trading volume has the potential to improve economic efficiency. However, a global centralized market over the Internet is not scalable system-wise. Thus, in addition to a single market architecture, GEM provides a decentralized architecture that allows traders to trade in their (physically) local market, perhaps even with each market employing different (but compatible) policies, and an infrastructure that provides inter-market communication and synchronization for approximating global market behavior. However, the decentralized market architecture is beyond the scope of this paper (see [14]). Here we focus on a centralized market (although the traders may be arbitrarily distributed of course), and mention market distribution only when it is affecting a particular design choice.

GEM has been fully implemented in Java 1.1 (except for a Swing-based user-interface that is part of 1.2) and an on-line market demo is available in http://www.dsg.technion.ac.il/gem. The rest of this paper is organized as follows: Section 2 presents a high-level view of the market architecture. Section 3 describes a detailed design of the market framework and discusses rationale for it. Section 4 highlights implementation issues, and Section 5 summarizes our contributions and points to future work.

2 The Generic Market Architecture

The high-level view of the framework is illustrated in Figure 1. We use the term "framework" in its classical object-oriented meaning, i.e., as a set of cooperating classes that makes up a reusable design for a specific class of software [15]. The framework consists of two main layers that clearly separate "system" and "economic" concerns. The lower system level is largely *market-independent*, providing general infrastructure services to the upper layer. These include security services for authentication of traders and encryption of their requests, persistence services for storing orders and for various logging services, and trader-market communication services. This layer leverages existing technologies (see Section 4), and its internals are of less concern in this paper.

The upper layer is *market-specific*, i.e., it requests services from the infrastructure layer but is otherwise concerned only with the operation of the market following its currently programmed policies. This separation allows to switch infrastructure implementations without affecting market-level code. For example, a particular implementation may use IIOP/CORBA [16] networking services, that complies with the communication API, instead of (the currently implemented) RMI [17]. The separation between system and market layers resembles the layered approach taken in Eco [18], another proposed framework for EC applications. Eco defines three

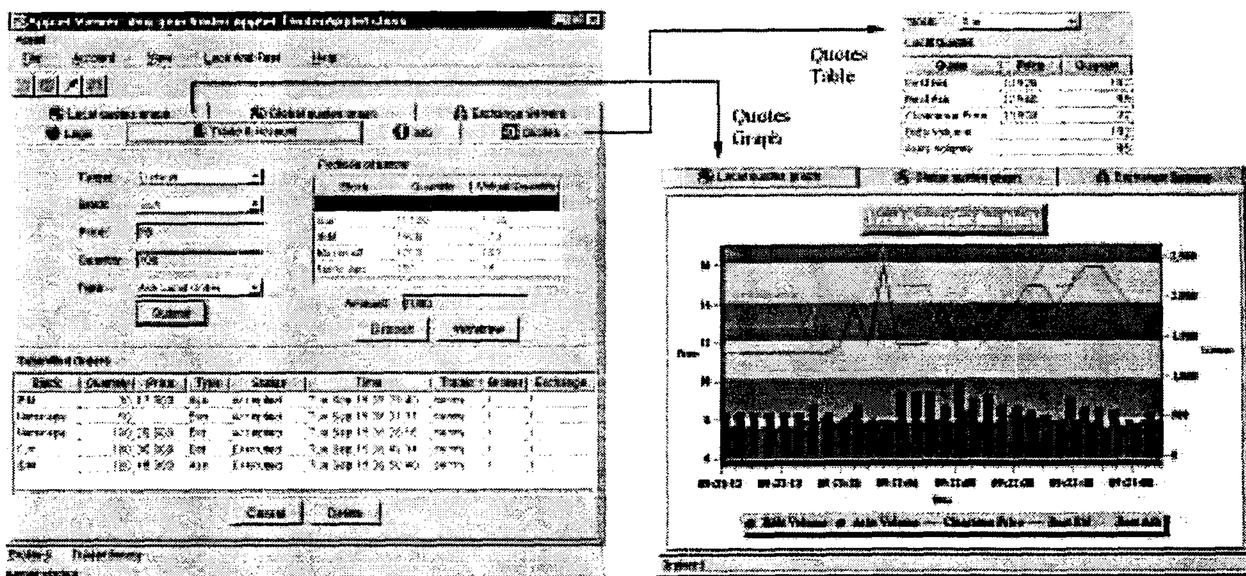

Figure 3: Trader Applet

interface for submission of orders, and the right side shows account information. The bottom shows the status of submitted orders. Other panels of the Applet show a table and a graph of the current quotes.

The Market Maker implements a general matching algorithm that decides which orders should be executed. This algorithm is good both for continuous as well as clearinghouse auctions, and can match multiple orders or only two orders. It is based on the algorithm used in AuctionBot [21], but is more generalized and improved. It uses 4 sets: 2 sets for unmatched asks and bids and 2 sets for matched. Orders are moved between these sets when placed. On execution the matched sets are already ready (except splitting last orders when quantity doesn't match). Since the best bid is the one with the maximal price and the best ask is the one with the minimal price, we represent asks as a negative value so the best ask is also the one with the maximal price (smallest negative). For more details, see [14].

4.2 System-Layer

In general, GEM is heavily based on Java [22]. The obvious portability advantage, dynamic loading, multi-threading and synchronization support, which are part of the basic language, made it a convenient platform to implement the inherent complexity and concurrency in Market applications. For example, for each stock we instantiate a separate set of components according to their policy, and each stock runs in a separate thread (although in practice it is possible and may be worthwhile sometimes to combine several stocks that are traded with the same market policies into a single thread, to reduce the consumption of system resources).

The wealth of Java-related technologies and utilities, has also made it possible to build a robust system with relatively limited resources. Some examples follow. The trader-broker-market communication is done through Java RMI. Thus, as a system-layer service, market designers need not be concerned with communication, and can use the primitives provided to them in the communication API of GEM. Persistence is supported by two mechanisms. A simple object-oriented database that comes with Voyager [19] (which is itself built on top of Java's serialization) is used to keep information on standing orders that are still in the market. Persistence here is mostly to ensure recovery from unexpected failures. Long term information on accounts, history of executed orders and price quotes are kept in a relational database (using Java Database Connectivity). This allows external (non-GEM) queries to be made against the database, for statistics, analysis, etc.

4.3 Dynamic Configuration

Dynamic configuration is facilitated by a special meta-component, named *Builder*, which is responsible for initializing components and replacing them on runtime. To achieve this possibility, each component goes through three separate stages of initialization: 1. *Initialize* for registration of name and reference in a shared context. 2. *Connect* for getting references of other components (from the context). 3. *Begin* for starting execution threads. There also 3 corresponding stages of destruction: 1. *Pause* for stopping threads (safely). 2. *Disconnect* for dismissing references of other components. 3. *Die* for deregistration

eventually gets matched in an executed transaction, or canceled (by either the trader or by the floor). Once executed, the order object flows into the information module at the back-end, and from there it gets directed to the trader, along with the status of the executed transaction. Due to the possibly long life-time of these objects and for correctness and robustness of the market, orders are kept persistently with transactional-semantics.

Finally, for flexibility and efficiency reasons, each stock can be traded by different rules. This design allows to optimize the policy to the trading patterns of each specific stock. The implication on the overall design is that it requires a separate exchange framework per stock (commodity) type. Thus, a top container for all these frameworks tunnels the orders and quotes to the proper framework according to the specific stock embedded in it. We now turn to discuss the details of the components and interconnections that comprise the market architecture.

3 Framework Design

Figure 2 gives a detailed view of the market architecture. Each component in this fine-grained decomposition encapsulates a distinct aspect of the market institution, can be customized independently of other components, and is governed by separate rules.

The front-end is comprised of the *Order Acceptor* and the *Authenticator.* Both components belong to the system-layer, and are therefore not concerned with the market-semantics of the incoming order. Specifically, the Acceptor checks whether the server capacity allows to accept another order, (i.e., the server is not overloaded), and the Authenticator checks whether the order's initiator is authorized for the specific stock and specific market (i.e., a broker).

The back-end is comprised of the *Information Board* and the *Order Status Notifier.* Unlike the front-end, this part contains both market-layer and system-layer functionality. The Information Board controls the *information dissemination rules*, i.e., *what* information is actually made available to traders. This "exposure" policy is clearly a market-level consideration. For example, in a sealed auction, only best bids and asks of the previous trading period should be exposed. The Notifier controls the *settlement rules*, i.e., all actions that need to be performed following a successful transaction execution. Its main responsibility is to notify the traders about their order status. In particular, in case an order is executed in a transaction, this component carries out the outcome (e.g., transferring digital signed money and stocks). The actual notification mechanism is a system-level function. To further promote "order-autonomy", each trader can be notified differently according to a mechanism it embeds in the order. For example, if the (automatic) trader is mobile, it may embed in its orders a "relative" address and request the infrastructure to forward the status to the trader's current location. Such design is not far-fetched. By using a mobile object framework such as Voyager [19] or Fargo [20], such functionality is readily available. In fact, GEM already uses Voyager, and a Fargo-based version is under construction.

The *Trading Floor* is the heart of the market, consisting of three main sub-components: 1. The Order Verifier determines *which* orders are accepted; 2. the Market Maker determines *what* orders are executed; and 3. the Scheduler determines *when* orders are executed.

The *Verifier* acts as a filter that controls whether orders are valid in the current institution. Thus, its behavior is governed by *bidding rules* such as limits on the quantity or price of an order, ensuring one seller in a single-sided auction, etc. Notice the distinction between this market-level filter and the system-level filters at the front-end.

The *Market Maker* is responsible for the *clearance mechanism*, i.e., *how* and what orders are matched, and in what price. It can be used to realize artificial specialists (the market maker can enter its own orders to the *Limit Order Book* in case of need), dealers, etc. This component further uses three sub-components that address separate aspects of the matching policy. The *Prioriter* determines priority between orders (e.g. price, time, quantity), the *Pricer* determines the clearance price (e.g. average price, first price, second price, etc.) according to the matched set of orders, and the *Limit Order Book* holds sets of unmatched and potentially matched orders (for more details on the market maker algorithm see [14]).

The *Scheduler* is responsible for the control flow of orders. It controls the continuity degree of the market and is governed by *clearance timing rules.* That is, it controls when orders are entered to the trading floor by invoking the Order Verifier, and it controls when orders are processed there, by instructing the Market Maker to execute the next transaction, thus ensuring proper synchronization between execution and order placement.

Discussion

To illustrate orthogonality of sub-components, consider two important double-sided auctions — double continuous Vs. double sealed auction. We can see that the same market maker mechanism can be used for both, and the only difference is in the Scheduler. In a sealed auction the scheduler determines the clearance interval and controls the flow of the auction's results (market price, volume of traded

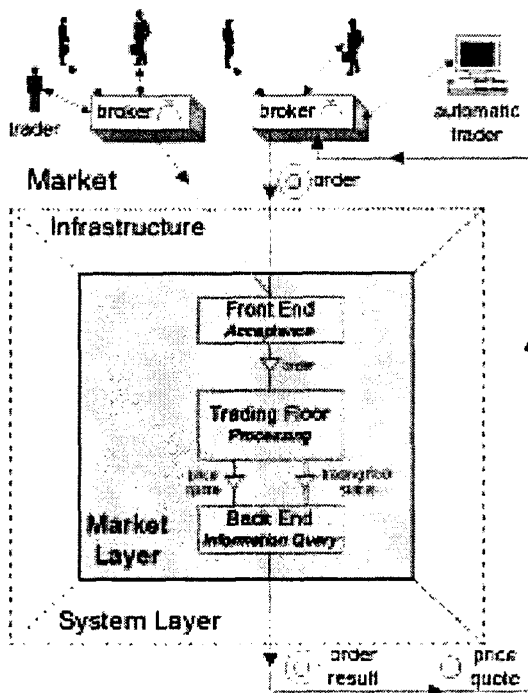

Figure 1: Market Framework Layered Architecture

layers: network services, commerce services and business services. However, Eco uses this separation for inter-application communication, whereas GEM defines layers within the same application.

The basic framework consists of three main elements (see Figure 1), each of which has its own system-layer and market-layer sub-components.

- *Market (server)* — This central element provides basic trading mechanisms, including handling and execution of bids and offers as well as providing information on stock (commodity) prices to enable traders to improve their trading.

- *Brokers* — These elements act as an interface between traders and the market. The broker's main function is to serve as an entity that is trusted by the market, similar to its role in non-electronic markets. Thus, brokers maintain account information and validate traders ability to settle their orders, thereby off-loading this task from the market server and distributing user authentication and account management among multiple (processing) entities.

- *Traders (clients)* — These elements communicate with brokers to get market information and to post bids and offers in order to buy and sell assets. A trader can be automated (i.e., artificial trader that trades according to pre-set configuration), or a human using a front-end graphical Applet. Both kinds access the market through the same interface, however.

Communication between traders and the market is encapsulated in communication objects, of which there are two kinds. *Order* objects are used to direct action requests to the market, and to return the status of the operations back to the trader. *Quotes* objects are used for retrieval of information about the current state of the market. The structure of these objects exemplify both the duality and the separation in our design. The inner circle (see Order object in Figure 1) represents market-properties that are accessed only by the market-layer of the exchange (e.g., bidding/asking price, quantity, or market-specific properties), and the outer circle represents system-level properties that are handled by the system-layer. For example, a typical order may contain the trader's signature that is authenticated by the security sub-system, the trader's address (e.g., its network address and port) that is used by the communication sub-system to deliver notifications, and the trader's logical name that is resolved to a physical address by a naming facility.

The interface to communication objects is simple and includes a small number of generic methods such as order submission or cancellation, getting price and volume quotes, and getting general information on the market (e.g. type of stocks traded, the type of market policy). In addition, the object includes in itself specific commands to be performed upon invocation of generic operations. This design allows to extend the functionality of orders and quotes without changing their interfaces. For example, the Order object is invoked by a single method and includes a command object field that can be interpreted as accept, withdraw, replace or query. By sub-classing the Order object one can add new constraints such as "accept if market price is below some value", by only knowing that the specific market supports this command. Another example is a Confirm Order sub-class of Order, which includes a confirmator address and requires the market to request a confirmation before executing an order.

The Market element consists of three main parts, each containing several components. The *front-end* (acceptance filter) checks whether the incoming order is valid. The *Trading floor* (processing unit) defines specific market mechanism, including components for timing, matching and prioritizing orders according to criteria defined by policy components, and queues for storing orders. And the *back-end* (information query) allows to notify traders on the status of their orders and on general price quotes.

A typical order enters the market through the front-end, gets queued inside the trading-floor, and

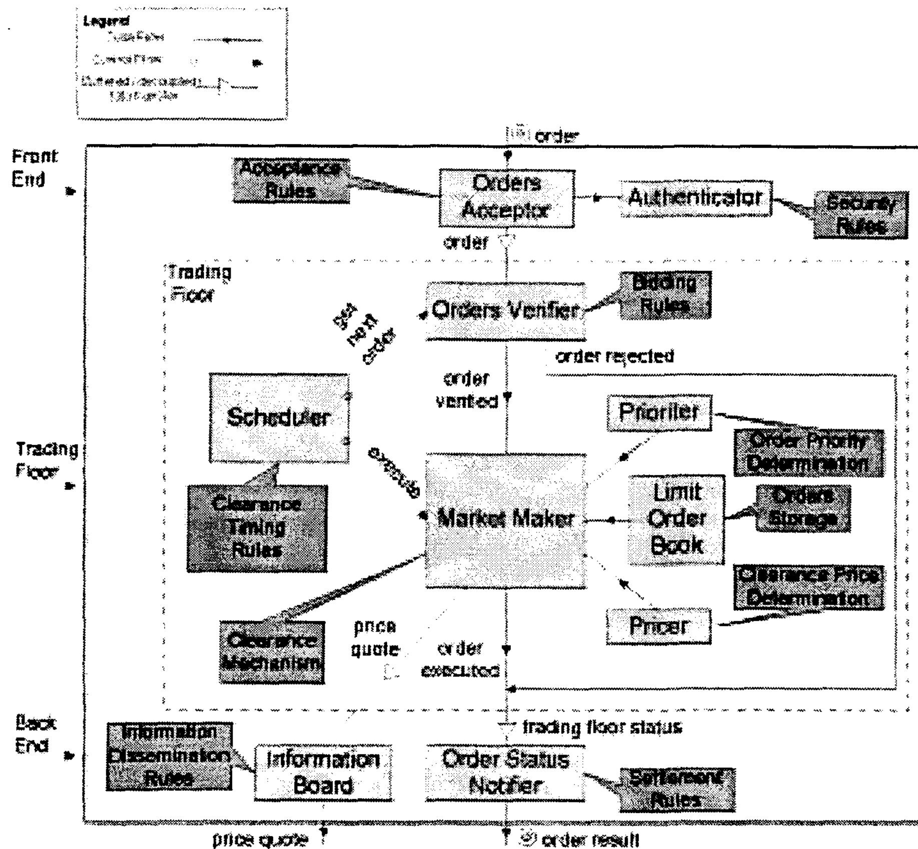

Figure 2: Market Framework Components

stocks, and status of trader's stocks) to the information board, performing an execution only every period of time (and letting the verifier to check orders in parallel). A double continuous auction is represented by a "zero interval", in which case the scheduler degenerates to a simple method that continuously inserts orders, directs the Market Maker to perform an execution if possible, and disseminates the order and/or transaction information.

A sealed auction is not only identified by a periodic scheduler, but also by an information board that does not expose quotes before execution is performed, whereas in continuous auctions the information board exposes all quotes. Thus, although components roles are orthogonal from the system point of view, they are not necessarily fully independent from the market perspective. Clearly, some policies require specific combination of components, and some combinations might not make sense economic-wise. Nevertheless, these components can still be mixed and matched to get new types of auction. For example, a periodic scheduler can be used with a non-sealing information board, to produce a clearinghouse auction that is totally open. This scheme may be good if the periods are too short for clients to get updated, or it can be opened only very close to the end of the clearance period to encourage traders to post new bids at the last moment.

As mentioned earlier, it is desirable to enable dynamic tuning of components according to system and market efficiency parameters. For example, the length of the clearance interval may be increased by the Scheduler to reduce load on the Exchange (system performance) or decreased to improve the rate of convergence to the competitive equilibrium price (economic performance). In fact, the interplay between system performance and economic performance is an interesting research issue that warrants its own investigation in the context of this system.

4 Implementation

4.1 Market Layer

Traders connect to the market using a well-defined trader API that defines basic connection and login facilities, as well as placing orders and quotes. We have implemented several kinds of traders (human user interface and also automatic traders with various trading policies). The trader Applet is shown in figure 3. The left side of the main panel shows the

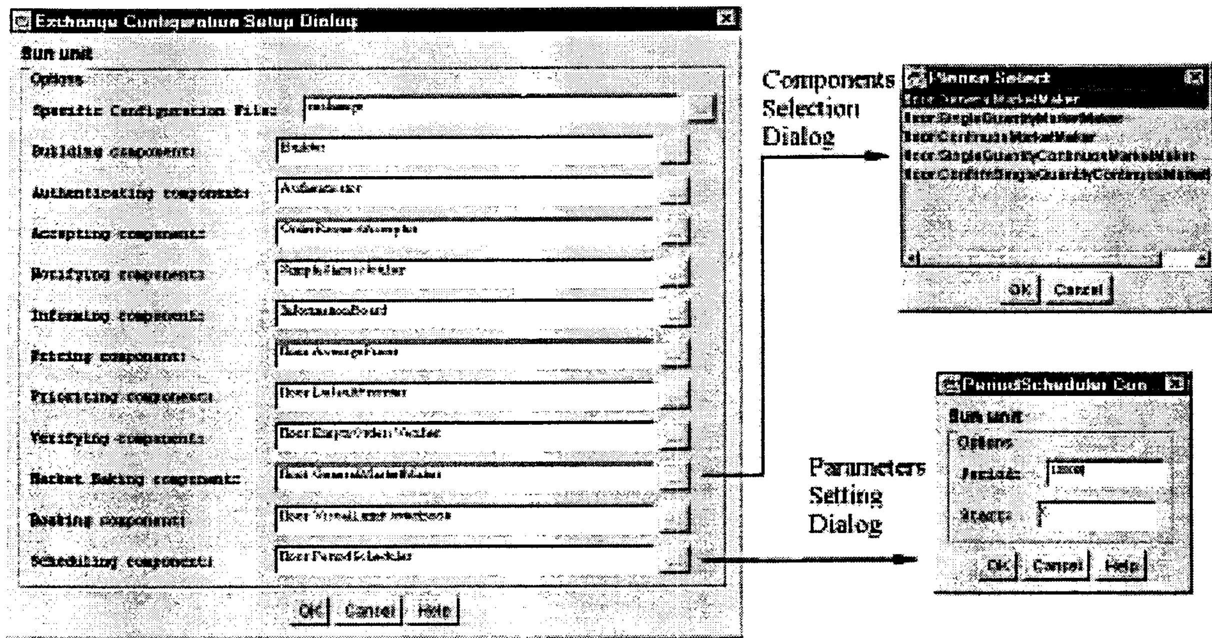

Figure 4: Configuration Utility

and resource deallocation. The market starts its operation only after all initialization stages have been successfully applied sequentially by the Builder to all components. These functions are part of a common interface which is a base for all components interfaces, allowing the builder to initialize components without knowing their specific interfaces.

Pausing the market is achieved by calling `pause` on all components. To resume, a call to `begin` on all components is needed. A component can be replaced at runtime by issuing a `pause` and `disconnect` on all components, `die` on the replaced component, `initialize` on the new component, and `connect` and `begin` on all components. Finally, the *Builder* itself, which decides which type of components to initialize, may be replaced to enable. the whole framework to be changed (in case the structure is not suitable for some unforeseen market mechanism). A Factory design pattern [15] is used by the application to initialize the proper Builder for each stock.

It is worth noting that components that include persistent data like the Limit Order Book are hard to replace at runtime. To solve this problem, there exist in the framework several storage components that can keep data but have no functional role. They are used by the replaceable components. They are themselves unreplaceable at runtime. Examples include queues which act as buffers between the front-end and the back-end from the trading floor, a quotes buffer, which is shared by the Market Maker and the Information Board, a buffer used by the Limit Order Book, and a special recovery buffer that records entrance and exit activity of orders to enable recovery from crashes.

Components are based on the JavaBeans architecture. A component class name can be read from a configuration file and dynamically loaded, replaced at runtime, and then reloaded and initialized as described above. The configuration tool that decides which classes to load, is shown in figure 4. The main panel shows the types of components that can be selected using a dialog box (top right side). After selecting the component, the configuration tool instantiates the class (using dynamic class loading), queries it (using Java Reflection), and opens a dialog box (bottom right side) to set the proper parameters.

4.4 Status

GEM is a "pure" Java system. The full codebase, including support for distributed markets, broker and various traders and at least a single implementation for each component in the framework, but excluding services we use from other vendors, contains about 40,000 lines of code, with a 850KB footprint. The exchange server itself contains about 30,000 lines of code, distributed in the following way: 1. *System Layer* — 15,000 lines. 2. *Framework* — 10,000 lines. 3. *Various Components* — 5,000 lines.

It is worth noticing that most of the code is in the system layer and the framework elements. The components, due to the fact that they can receive most services readily, are relatively small, enabling to conveniently add new components to the system.

More implementation details can be found in [14].

5 Conclusion and Future Work

The main objectives of this research are to investigate common characteristics of electronic markets, fine-grained "vertical" decomposition into components that can be independently tailored or replaced to form specific (and unforeseen) market policies, and a "horizontal" separation between market and system functionality that allows market designers to focus on market-level issues without having to implement the tedious and complex system functionality.

The GEM system addresses these issues by providing an abstract framework with pre-defined but generic component interfaces, a complete system-layer implementation, and implementations of several components from each type and thus several market institutions. In addition, GEM allows dynamic configuration of the market, which can be used by administrators to tune the market according to changes in the environment or in market trends.

We are presently working on implementation of a distributed market. The availability of multiple markets raises new issues and may involve new technologies. For example, we are employing mobile objects to enable (automatic) traders to get closer to the market in which they want to trade. Another important deployment of GEM is to enable researchers to use it as a test bed for experimentation, particularly for electronic markets. As a starting point we are conducting our own experiments and simulations in order to compare various market mechanisms, especially in the context of distributed markets.

Acknowledgments

This work was done with partial support from the Israeli Ministry of Science, grant number 9750. We are grateful to the members of the Artificial Markets Project [5] at MIT, who have helped us tremendously in understanding market mechanisms and their possible implications on electronic market design. We would especially like to thank Tommy Poggio, Andrew Lo, Michael Oren and Nicholas Chan. We would also like to thank other members of the Distributed Software Systems Group at the Technion for their help in the system design.

References

[1] Forrester Research, Inc. Home page at `http://www.forrester.com/`.

[2] R. Forsythe, F. Nelson, G. Neumann, and J. Wright. The IOWA presidential stock market: A field experiment. *Research in experimental economics*, 4:1–43.

[3] A. Chavez, A. Moukas, and P. Maes. Challenger: A multiagent system for distributed resource allocation. In *Proceedings of the International Conference on Autonomous Agents*, Marina Del Ray,California, 1997.

[4] O. Regev and N. Nisan. The Popcorn Market - An online market for computational resources. In *First International Conference On Information and Computation Economies*, Charleston, SC, October 1998. To Appear.

[5] A. Lo and T. Poggio. Artificial Markets Project at M.I.T. Home page at `http://cyber-exchange.mit.edu/`.

[6] A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, April 1996.

[7] P. Wurman, M. Wellman, and W. Walsh. The Michigan Internet AuctionBot: A configurable server for human and software agents. In *Second International Conference on Autonomous Agents*, 1998.

[8] OnSale^{R} auction supersite^{TM}. Home page at `http://www.onsale.com/`.

[9] eBay^{TM} auctions. Home page at `http://www.ebay.com/`.

[10] Preston R. McAfee and J. McMillan. Auctions and bidding. *Journal of Economic Literature*, 25:699–738, 1987.

[11] Paul R. Milgrom and Robert J. Weber. A theory of auctions and competetive bidding. *Econometrica*, 50:1089–1122, 1982.

[12] D. Friedman and J. Rust. *The Double Auction Market: Institutions, Theories, and Evidence.* Addison-Wesley Publishing, Reading, MA, 1993.

[13] W. Vickrey. Counterspeculations, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, March 1961.

[14] B. Reich and I. Ben-Shaul. Design and Implementation of the GEM System. Technical report, Technion — Israel Institute of Technology, September 1998.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Welsley, Reading, MA, 1995.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification. Revision 2.2*, February 1998. Available at: `http://www.omg.org/corba/corbaiiop.htm`.

[17] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI) Specification*, December 1997. Available at: `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`.

[18] J. Tenenbaum, T. Chowdhry, and K. Hughes. Eco System: An Internet Commerce Architecture. *Computer*, 30(5):48–55, May 1997.

[19] ObjectSpace Voyager core package: Technical overview, December 1997. Available at: `http://www.objectspace.com/voyager/whitepapers/VoyagerTechOview.pdf`.

[20] O. Holder and I. Ben-Shaul. Dynamic layout of distributed applications. In *Proceedings of the 3rd International Software Architecture Workshop, in conjunction with ACM SIGSOFT'98*, Orlando, FL, November 1998. To Appear.

[21] P. Wurman, W. Walsh, and M. Wellman. Flexible double auctions for electronic commerce: Theory and implementation. *Revised version to appear in Decision Support Systems*, 1998.

[22] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, Reading MA, 1996.