Research Feature

A Web-Based Financial Trading System

Rapid advances in IT and growing competition are causing fundamental changes in the world's financial services industry. This article describes an electronic market that traders use to execute bundle orders. Because it is based on distributed objects, it has significant advantages over systems built with CGI scripts.

Ming Fan
Jan Stallaert
Andrew B. Whinston
University of Texas at Austin

A new electronic financial market is fast emerging. Connected by high-speed networks, buyers and sellers are gathering in virtual marketplaces and revolutionizing the way business is conducted. The financial services industry, among the most innovative and aggressive in its use of IT, has created an entirely new online brokerage industry in just a few years. In 1996, there were only 1.5 million online brokerage accounts. That number was estimated to be 5.3 million at the end of 1998, and today, these accounts are responsible for 25 percent of all retail trades.[1] Future technological advances will introduce new trading mechanisms and other new electronic markets.[2]

The principal functions of financial markets are to bring buyers and sellers together and to provide a price discovery mechanism for the assets being traded. In this article, we describe our financial bundle trading system (FBTS), a Web-based continuous electronic market that traders can use to execute *bundle orders*. With a bundle order, a trader can order a combination of stocks or assets. We used a novel bundle trading mechanism developed by the Center for Research in Electronic Commerce at the University of Texas at Austin, described in the sidebar "Automated Matching Mechanism." FBTS provides universal access to traders on the Internet, allows interactive information exchange between traders and market makers, and executes trades using the automated matching mechanism. FBTS is in an experimental stage and is being extensively used for research at the University of Texas at Austin.

We developed FBTS using a distributed object model based on Java RMI (remote method invocation), which supports interactive communication between trading applications and the market. This approach has significant advantages over CGI (Common Gateway Interface) scripts because it improves interactivity by allowing continuous updates. We implemented concurrent trading processes and concurrency controls using Java's multithreading technique. We also implemented asynchronous communication.

Because the financial sector has a sizable presence in the IT market, these innovations and the subsequent institutional changes will also, in turn, strongly influence future IT development in areas such as distributed computing and Web-based application development. We believe financial innovations such as the FBTS and its supporting information technologies will greatly impact the organization of financial markets.

BUNDLE TRADING

Today's financial markets are built around auctions, and there are several auction formats. The specialists at the New York Stock Exchange, for example, are human auctioneers who handle orders and execute trades. Other exchanges are automated, including the Toronto Stock Exchange's CATS (Computer Assisted Trading System) and Paris Bourse CAC (Cotation Assistee en Continu).[3]

Whether automated or not, all these markets execute trading asset by asset; none of them can accommodate bundle orders. A bundle order is a combination of stocks or other financial instruments (commodities, options, bonds, and so on). Large institutional investors, such as mutual funds, often need to trade bundles in order to rebalance their portfolios. For example, an index fund may have to maintain a portfolio that matches the investment performance of the Standard & Poor's 500 Index. The fund manager must rebalance the portfolio as the composition of the S&P 500 Index changes. With current systems, fund managers must do so by trading stocks in separate markets using separate orders. This not only incurs

0018-9162/99/$10.00 © 1999 IEEE

large transaction costs but also increases uncertainty about the overall cost of the portfolio.

Our financial bundle trading system (FBTS) is an automated, continuous auction market that executes bundle orders to buy and sell. Bundle trading lets fund managers pay more attention to overall cost, rather than to the cost of individual stocks. Overall cost is more important to the fund's performance.

For example, suppose a trader wants to buy a portfolio of the so-called "Dogs of the Dow," the highest-yielding stocks included in the Dow Jones Industrial Average. Suppose the trader wants to buy 1,000 shares each of AT&T, DuPont, Exxon, and GM; 600 shares of J.P. Morgan; and 1,500 shares of International Paper. Our trader wants to acquire the whole portfolio for Dogs of the Dow in this proportion (that is, 2.5 times as many shares of International Paper as of J.P. Morgan). So the relative asset weights in this portfolio are 1,000, 1,000, 1,000, 600, and 1,500 (or 1, 1, 1, 0.6, and 1.5).

As Table 1 shows, the trader first specifies the asset weight for each stock and the bundle quantity. She bases her recommended purchase prices on the previous closing prices and her own fundamental analysis. She then places *limit orders*, judging the previous closing prices as fair market prices. A limit order specifies that the stock will not be purchased if the price exceeds the recommended purchase price.

The next day, DuPont, Exxon, and GM are all trading in ranges higher than the recommended purchase price. Therefore the buy orders for these stocks cannot be executed, and the fund manager fails to acquire a balanced Dogs of the Dow portfolio.

Bundle trading could have prevented this. With bundle trading, the fund manager could have specified a limit order price of $361 for the entire bundle. The bundle's limit price is simply the sum of each stock's recommended price and its asset weight. As Table 1 shows, the maximum price for the weighted bundle was $360.35, less than the limit order of $361.

Automated Matching Mechanism

The FBTS employs a real-time order matching and execution system to maximize the trade surplus per bundle match.

The bundle matching mechanism[1] finds one-to-one, one-to-many, or many-to-many matches between offers. This type of match requires intensive computations and is too complicated to handle manually. Our fully automated matching program finds the matches that maximize the market surplus among the open orders subject to the condition that the total amount bought of each asset cannot exceed the total amount sold.

For example, in

$$\max \sum_{j=1}^{n} p_j x_j \qquad (1)$$

Subject to:

$$\sum_{j=1}^{n} b_{ij} x_j \le 0, i = 1, \text{K}, m \qquad (2)$$

$$\sum_{j=1}^{n} x_j \le 1 \qquad (3)$$

$$x_j \ge 0, j = 1, \text{K}, n \qquad (4)$$

vector p is the limit price. A positive limit price signals a willingness to pay for a trade; a negative limit price signals a willingness to sell. Vector x is the proportion of the matched trade. Every element of x should be positive (constraint (4)). The objective function is the goal of the match: to maximize the market surplus. In the matrix B, which contains n vectors $[b_1, b_2 \ldots b_n]$, the element b_{ij} is the bundle weight for asset i of order j.

Each vector represents the composition of a particular bundle. For example, $b_j' = [2, 1, 1]$ means that bundle j contains two shares of asset 1, and one share of both assets 2 and 3. A positive number in the bundle vector means "buy," and a negative number means "sell." A bundle does not have to contain pure buy or sell orders; instead, it can contain mixed buy and sell orders. Constraint (2) means that for a bundle to be matched, each buy order in the bundle must be matched with a sell order of the same asset from another bundle or bundles. For simplicity, we standardized the matched trade to be a number less than or equal to one (constraint (3)).

Reference

1. M. Fan, J. Stallaert, and A.B. Whinston, "The Design and Development of a Financial Cybermarket Based on a Bundle Trading Mechanism," working paper, Center for Research in Electronic Commerce, The University of Texas, Austin, Texas, 1998.

Table 1. Trading the "Dogs of the Dow."

Stock	Asset weight	Recommended purchase price	Next day's trading range	Next day's highest price	Trade executed? (Yes/No)
AT&T	+1	60	54-57	57	Y
DuPont	+1	60	60.5-61	61	N
Exxon	+1	68	68.5-71.25	71.25	N
GM	+1	59	59.25-61	61	N
J.P. Morgan	+0.6	90	88-91	91	Y
Int'l Paper	+1.5	40	33-37	37	Y
Bundle price		361		360.35	
Bundle quantity	1,000				

The bundle order could have been executed easily and a balanced portfolio obtained.

There are three potential types of market participants: market maker, dealer, and public trader. The FBTS currently supports the market maker and public trader. The market maker can access all market information, while a public trader can view only the market price, last traded volume, and current bid and ask prices. Individual traders cannot access information such as the prices and sizes of all open orders in the limit order book, or the identities of other traders. This prevents them from exploiting the private information of other traders and influencing market prices.

Bundle trading has two main advantages:

- *It lets traders submit just one order.* Traders submit a single bundle order with a limit price instead of several separate orders.
- *It ensures a balanced portfolio.* If the entire bundle cannot be executed, no trade occurs. Therefore, the trader always gets a balanced portfolio.

Bundle trading can also improve resource allocation methods in other industries, even in those industries that require centralized decision making or similar inefficient mechanisms. For example, bundles could be applied to the problems of airport runway allocation, railroad access allocation, and natural-gas trading.

SYSTEM ARCHITECTURE

The FBTS consists of two primary applications: the Exchange and the TradeApplet. Figure 1 illustrates the FBTS system components and communication model. FBTS views all applications as objects that are uniquely identified and accessed throughout the network.

The Exchange

The Exchange is the market application. It manages and coordinates the trading activities across different computing platforms. It contains

- *Three servers:* a Web server, a database server, and a naming and directory server. The last provides unique object identifications throughout the entire system.
- *A limit-order table:* This Java hash table stores all open orders. Information about orders that have been filled or canceled is saved in the trade history database before it is deleted from this table.
- *An order routing and notification system:* This system monitors the limit order table and notifies traders of their activities.
- *Automated bundle matching program:* This program matches orders in real time and calculates transaction prices and trading quantities.

This architecture is both scalable and transparent. FBTS is scalable because its design allows for the distribution of different services (represented by objects) among different computers. This not only means that more computers can be added to the system as the number of users increases, but it also means that services themselves can be distributed. The order routing system, the limit-order table, and the matching program are all currently located on an RS/6000 multiprocessor workstation. But the limit order table and

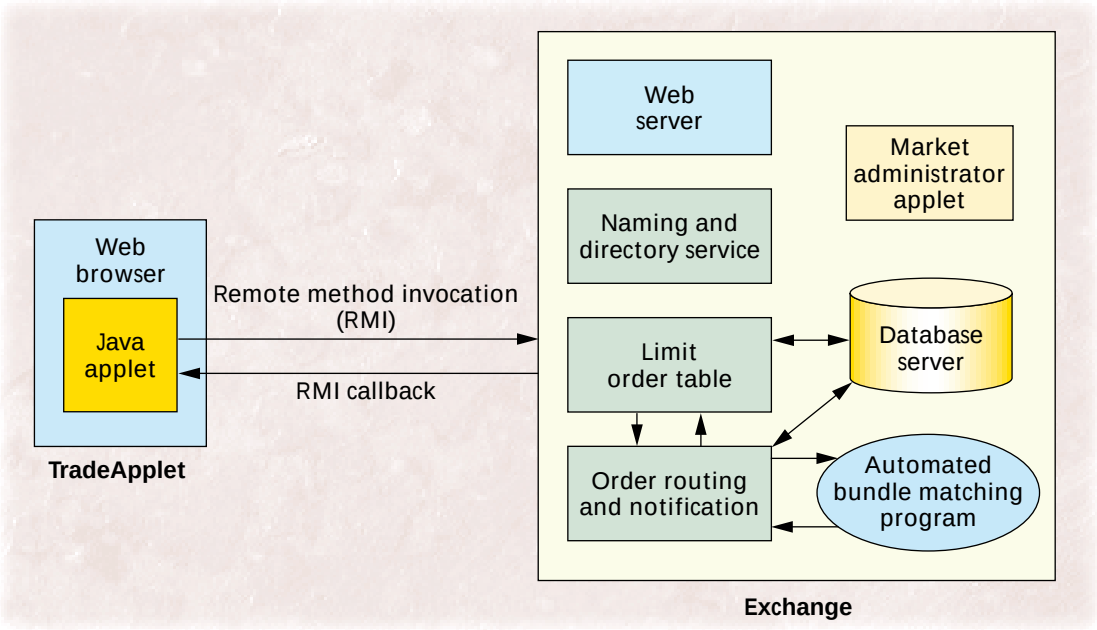

Figure 1. FBTS architecture.

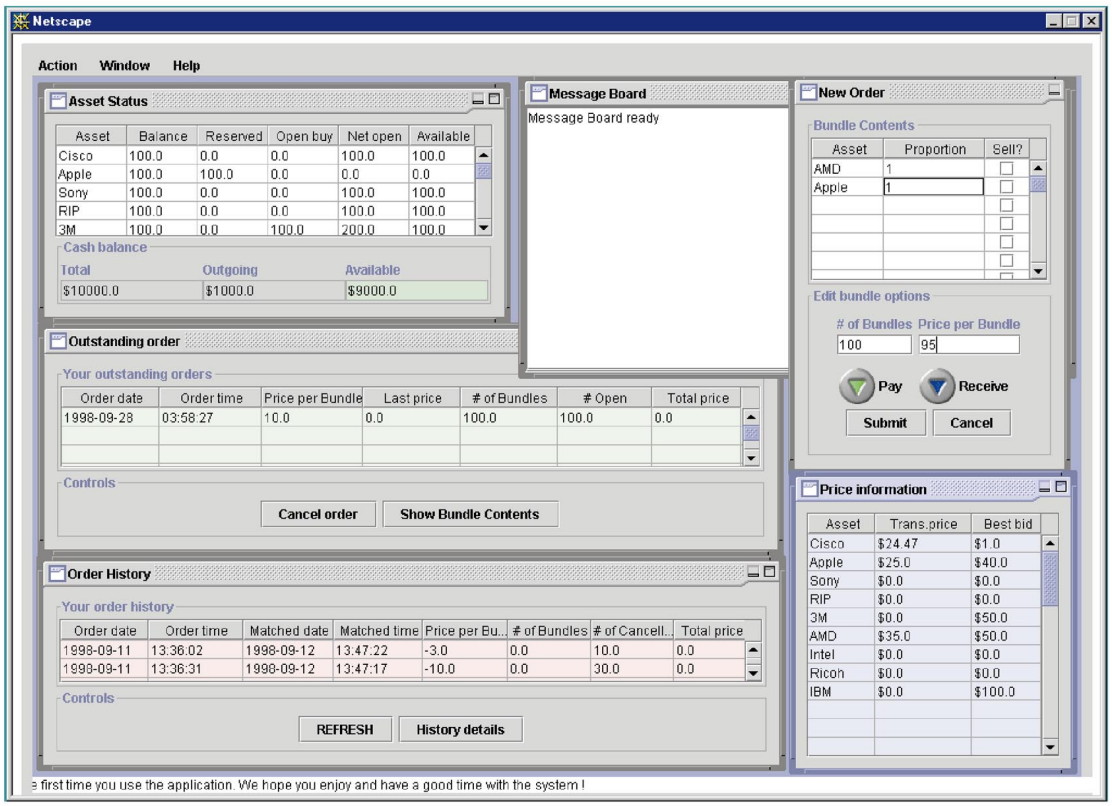

Figure 2. General layout of the TradeApplet, which contains six windows.

the matching program could operate from different workstations. This distributed object architecture ensures that the application need not change as the scale changes, providing location and access transparency throughout the system.

FBTS is transparent because the TradeApplet client, described next, does not need to know exactly which machine holds the limit order table. The naming and directory service presents a coherent bundle market to the traders and hides the internal configuration of the system. Objects located on different computers can call the methods from other objects consistently.

The TradeApplet

The TradeApplet (the client application) is accessed via a Web browser. Traders simply log in and conduct trades. FBTS contains order error-checking functionality, based on its own trading rules, so it catches invalid orders before they reach the Exchange. This helps to reduce server workload and minimize network traffic.

As Figure 2 shows, TradeApplet presents six windows.

- *Asset Status.* The trader's current holdings. "Reserved" lists the number of shares for the trader's sell order that are not yet filled; "Open Buy" shows the number of shares in the open order; "Net Open" is the difference between the two.
- *Message Board.* The text of messages sent from the Exchange are displayed here.
- *New Order.* Where traders submit new orders. Traders are not required to submit bundle orders. To enter a bundle order, the trader enters the proportions (weights) for the bundle. If he wants to sell an asset, he checks the "Sell?" check box. He also enters the bundle quantity and limit price. The Pay and Receive buttons identify whether the valuation of the bundle is net cash outflow or inflow.
- *Outstanding Order.* The current open orders for the trader, including information such as date and time of the order, limit price, last traded price (it shows 0 if the order is not traded), bundle quantity, open quantity, and total transaction amount.
- *Order History.* The orders that have been either entered or canceled. If the trader wants to find more information about a past order, he can select the order and click the History details button.
- *Price Information.* Information such as the last traded price and the best bid and ask prices.

We used Swing components to develop the TradeApplet. (Swing is the visual component kit in the Java Foundation Classes that facilitates rapid GUI development.) Developing with Java and Swing allowed us to present a standardized yet flexible interface. Unlike a typical Web page, traders can open multiple windows and minimize or close windows.

CORBA AND JAVA RMI

Using Java RMI, along with the Common Object Request Broker Architecture (CORBA), also allowed us to develop FBTS as a truly interactive application.

CORBA is a distributed object technology standard[4] that allows objects to communicate independently of the specific platforms and techniques used to implement them. In CORBA, objects are defined

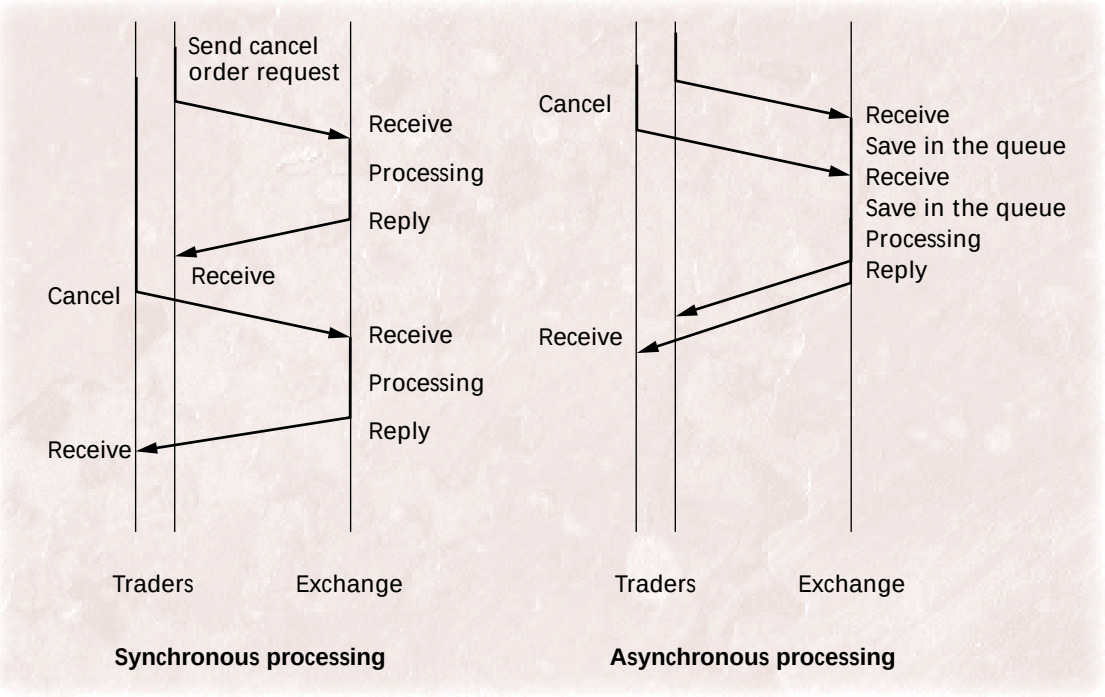

Figure 3. Synchronous versus asynchronous communication.

as a set of interfaces declared in Interface Definition Language (IDL). The implementation of the object is independent of the interface and hidden from other objects. The Object Request Broker (ORB) guarantees portability and interoperability of objects over a network of heterogeneous systems.

RMI is a Java-based framework for distributed object applications that makes it easier to develop distributed objects if (like the FBTS) the applications are developed in a pure Java environment. (In our opinion, a Java development environment is well suited for dynamic and interactive distributed applications.) We implemented the Exchange as an RMI remote object server and the TradeApplet as a Java applet.

The TradeApplet locates the Exchange through the naming service and sends it information. Because we implemented RMI callbacks, the Exchange can pass information back to the TradeApplet. The TradeApplet can pull information from the market and the Exchange can push information to the TradeApplet.

Despite the differences in implementation, Java RMI and CORBA offer similar functionality. Distributed object technologies like Java and CORBA have three important advantages over applications that are based solely on HTTP and CGI.

- *They do not require reloading or resubmitting.* If CGI scripts had been used, traders would have to reload the Web page or resubmit their requests to get the real-time market information. Obviously this kind of delay would dramatically degrade the effectiveness of a financial application.
- *They support efficient information transmission.* The Java/CORBA model passes the values of variables among different applications. In contrast, CGI programs must recreate a Web page and send the entire HTML file back to the browser every time the server responds to a request.
- *They are easier to develop and maintain.* To upgrade FBTS, we can make a change at the individual object level instead of at the system level. As long as the interfaces among the objects remain constant, we can change the implementations of those objects as needed.

CONCURRENT PROCESSING AND SYNCHRONIZATION

FBTS uses both synchronous and asynchronous remote method calls. Figure 3 illustrates the differences between synchronous and asynchronous processing of a cancel order, for example.

Synchronization choices are crucial in the case of systems that support concurrent processing. In FBTS, traders can invoke three processes:

1. *Submit* a new order.
2. *Cancel* an open order.
3. *Query* the details of an executed or partially executed order.

And the Exchange can execute three processes simultaneously:

1. *Route* all open orders in the limit-order table to the matching program.
2. *Match* orders and calculate market clearing prices and trade volume.
3. *Notify* traders of order execution and market information.

So therefore traders can submit new orders while the Exchange routes open orders to the matching program, which conducts matches in real time. If a match is found, the order execution system updates

the limit-order table and notifies corresponding traders.

These multiple processes use Java threads to execute concurrently. A new thread is invoked every time a trader sends a request to the Exchange. Threads require fewer system resources than computations. In a multiprocessor workstation, multiple threads can operate simultaneously to take advantage of different processors. In a single processor machine, multiple threads can run in an interleaved manner so that different tasks run simultaneously. Thus the Exchange can concurrently perform intensive computations for order matching and at the same time support interactive access.

However, multiple threads are not protected; more than one thread can access the same data item. The most common way to implement concurrency control is to use exclusive locks. By locking the data, the application is in effect serializing access to the data. For example, when a trader submits an order, a new thread is launched at the Exchange. The thread operation has three parts:

1. Assign the current OrderID to the new order.
2. Increment the OrderID.
3. Add the order to the limit-order table.

Now suppose two traders submit orders and the resulting threads interrupt each other:

1. Trader A starts to submit an order. Thread A executes part 1 of the submit order.
2. Trader B starts to submit an order. Thread B interrupts Thread A. Thread B executes part 1 of the submit order.
3. Thread A interrupts Thread B. Thread A executes parts 2 and 3 of the submit order.
4. Thread B finishes parts 2 and 3 of the submit order.

This scenario causes the two orders sent by traders A and B to have the same OrderID.

We can solve this problem by adding the synchronized keyword to the SubmitOrder() method. This keyword serves as a mutually exclusive lock for the method, allowing only one thread to call the method. Upon completion of the method, the thread automatically releases the lock. Locks are useful if the portion of the data that must be serialized remains as small as possible. If unnecessary locks are applied, program performance becomes less efficient.

For example, if a trader cancels an order right after it was routed to the matching program, it is immediately deleted from the limit-order table. The matching program may then find a match and make a trade, only to find that the order has been deleted. To solve this problem, we could lock the limit-order table while matching is conducted, but this approach would freeze the limit-order table constantly. Instead, we used asynchronous processing.

```
// The routing & notification thread
{
   Infinite Loop
   {
      while (the cancel queue is not empty)
      {
         select the front cancel order from the cancel queue;
         delete the order from the limit order table;
         remove the cancel order from the cancel queue;
         notify the trader;
      }

      route the orders in the limit order table to the matching
         program;

      if (matches are found)
      {
         if (matched order IDs are in cancel queue)
         // It means that the trader(s) sent cancel orders after the orders
         // have been routed for matching. Cancel order comes too
         // late!
         {
            delete these order IDs in the cancel queue;
            notify the traders that cancel comes too late;
         }
         update price & quantity information & notify the traders;
      }
   }
}
```

Figure 4. Pseudocode for asynchronous communications for order cancellation.

In a synchronous remote call, object A sends a message to object B and waits for feedback. Thus, the sending and receiving processes synchronize with every message. To continue, object A has to wait for feedback from object B. Object B has to respond instantaneously to object A's request as well as to other remote calls. Otherwise, object A and other objects will be delayed while waiting for replies.

With asynchronous communication, the server can schedule its operations more efficiently because it does not have to reply to each order immediately. Meanwhile, the client application does not have to wait for an immediate reply in order to conduct the next task. We use asynchronous communication for cancel orders because we do not want the trade process to wait for the replies of the cancel process. Normally, an order cannot be canceled immediately if matching is processing. Using asynchronous communication, the cancel requests are stored in a queue at the Exchange side. After submitting the cancel requests, the client application can proceed without waiting for the replies. The Exchange side empties the cancel queue each time before it restarts the matching program. Figure 4 shows the algorithm of the cancel process.

As the field of computerized market mechanisms is more recognized as part of electronic commerce, the traditional issues of authentication, secure and efficient communication, as well as the rigorous implementation of secure order entry from unauthorized users will have to be seriously addressed. Further research will show how multiple markets trading overlapping assets will be integrated, and

how the arbitrage opportunities can be eliminated. The emerging stream will lead to profound changes in the financial industry where traditional exchanges such as the New York Stock Exchange will eventually evolve into a computerized trading system that incorporates these richer ways of trading assets. ❖

Acknowledgment

The authors gratefully acknowledge the support of the IBM Institute for Advanced Commerce.

References

1. P. Dwyer, "The 21st Century Stock Market," *Business Week*, Aug. 10, 1998.
2. S. Choi, D. Stahl, and A.B. Whinston, *The Economics of Electronic Commerce*, Macmillan Technical Publishing, Indianapolis, Ind., 1997.
3. I. Domowitz, "The Mechanics of Automated Trade Execution Systems," *J. Financial Intermediation*, Vol. 1, 1990, pp. 167-194.
4. J. Siegel, *CORBA: Fundamentals and Programming*, Object Management Group, John Wiley & Sons, New York, 1996.

Ming Fan is a PhD candidate in the Department of Management Science and Information Systems at the Graduate School of Business, University of Texas at Austin. His research interests include technologies for financial markets, supply chain management, and large-scale distributed system development based on economic principles. Contact him at mfan@uts.cc.utexas.edu.

Jan Stallaert is an assistant professor in the Graduate School of Business, University of Texas at Austin. His research interests are large-scale system optimization, financial engineering, and supply chain management. He received a PhD in management from the Anderson School of Management at UCLA. Contact him at stallaert@mail.utexas.edu.

Andrew B. Whinston is the Hugh Cullen Chair Professor in information systems, computer science, and economics at the University of Texas at Austin and is is the director of the Center for Research in Electronic Commerce. His research spans various realms of electronic commerce, its emerging technologies, and its impact on business protocols and processes. Contact him at abw@uts.cc.utexas.edu.