



Technical Specifications

Destinations: AMEX, PHLX, and PCX

Description

The core of any professional securities trading system is the ability to execute, track and clear stock, options and futures trades. To do this, traders often look to service bureaus as an intermediary to the Exchanges and ECNs. Most service bureaus have aging infrastructures and closed APIs. This project sought to create a new service bureau with an Internet-exposed SOAP interface with a fully published API. Such a service bureau enables professional traders to use the applications provided by the new service bureau or create their own custom applications.

The initial release has been targeted toward professional traders, but the goal was to open the service bureau up to all traders. Web-based thin clients as well as compiled thick clients enable any trader to have access to the major Exchanges and ECNs.

AURA's role in this project was two-fold. AURA constructed a Java-based client utilizing the SOAP interface to enter and monitor trades, view live prices and the price books for specific ECNs. AURA also constructed the destination interfaces to the AMEX, PHLX and PCX Exchanges. The destination interface handles the translation between internal order identifiers to the exchanges order identifiers. All orders are tracked until completion, even if the order took months to fill.

Introduction

The initial set of design documents were not intended to be a firm set of designs, but rather a higher set of designs that indicate generally how the component runs. Implementation issues, such as how threads will be managed or will messages from the exchange be pushed or pulled through the system, do not effect the overall design, and as such are defined in the lower-level designs.

Each exchange had a single COM object to handle all functionality. A single instance of the COM object handled a single connection to a single exchange. Everything the COM object does is logged. COM objects for AMEX, PHLX and PCX were created.

Overview

Objective The technical objective of this project was to construct a DLL containing 3 COM objects, each implementing a specific exchange destination for option orders and fills. The three exchanges implemented were AMEX, PHLX and PCX. The DLL is attached to and instantiated by Microsoft's IIS server, and employs the current system's API for communication between the system and the DLL. The DLL contains all of the functionality required to route orders to and receive fills from the exchanges. If the lines of communications are open, the DLL is responsible for proper transmission of messages to and from the exchanges. The DLL is not responsible for resending on errors, but must attempt to resolve ambiguous order states. The DLL also attempts to retrieve missing messages from the exchange, if they can be determined. Appropriate log messages are designed to allow the tracing of all messages and errors between the existing system and the exchanges. The DLL can handle a start/shutdown at any time

without losing open order state.

Document Contents

The project and the object design are divided into common core components and exchange-specific modules. The sections following will describe the functionality of these components.

Core

System Libraries

Logging

Every message is logged in its entirety on receipt, generation or translation. The original and the resulting message are logged in the case of message translation of any sort. Appropriate trace, warning and error messages are logged, along with the entire message that was involved in the trace, warning or error condition.

Communications

The DLL extends the current system's interfaces and implements Initialize, Uninitialize and Send as well as utilize the PassToSubsystems method currently implemented. All of the processes are thread and re-entrant safe. Calls to send() are synchronized with a lock, and result in the instantiation of a Messenger object to manage the processing of the message.

Initialize

The call to initialize() initializes the main objects within the COM object. Initialize sets up the global Configuration object, then initialize in order, the DataAdapter, destination Connection, and incoming message Receiver. Should any component fail to initialize, any initialized objects are uninitialized and deleted. A negative return code signifies an initialization failure. A 0 return code signifies success.

The DataAdapter is responsible for data translation. It contains the private member DataManager, which holds any data needed to perform the functions of the COM object. Initialization of the DataAdapter is dependent on the initialization of the DataManager. All critical data is stored in the DataManager via a Memory File Map (linked memory and file), insuring that all items written to memory are stored on file should a shutdown occur for any reason. On initialization the DataManager re-associates its memory with the memory file map, allowing for the continued processing of open orders. The initialization state of the DLL resumes the state of the last execution.

Uninitialize

Uninitialize will shutdown the COM object as quickly as possible, however, it waits the configured amount of time for all in-process messages to complete.

Uninitialize first starves the system by disabling additional calls to send, then calling Receiver.uninitialize. Receiver.uninitialize() prevents additional calls to Communication.receive(), and kills all receiver threads as they complete processing or become idle. Receiver.uninitialize() returns once all receiver threads have terminated.

A timeout is set for the uninitialize, and all threads are killed immediately once the timeout has been reached. Uninitialize then uninitialized the exchange Communication object and the DataAdapter.

Finally, all object pointers deleted.

Send

This method is thread and re-entrant safe, allowing for multiple simultaneous calls. Each order sent is managed by a single thread (probably the calling thread), which returns success or failure of the send. Success indicates successful transmission (and acknowledgement where applicable) of the order to the exchange.

A single call to the Send method results in the creation of a Messenger object to manage the process of delivering the message to the exchange. The Messenger passes the RML messages through the DataAdapter for translation, sends the CMS message to the exchange through the Connection, and attempts to resolve any ambiguous transmission states by resending POSDUP messages, as configured.

The result of the final transmission attempt is sent back to the calling function as a return code. Success returns a 0; errors are a negative number.

On successful send, the Messenger makes a final call to the DataAdapter, confirming the transmission. This allows for proper data storage or cleanup as needed.

The Messenger is not concerned with business-level acknowledgements, but instead only waits for transmission-level acknowledgements. Success simply means that the message was sent successfully.

PassToSubsystems

MS Queue User Work Item API is used for thread pool management.

Messengers manage inbound messages individually. CMS messages received are translated via a call to the DataAdapter. The resulting RML message is sent to the current system via a PassToSubsystem(RML) call. The messenger will not attempt a resend should the PassToSubsystem call return a failure state. On success, the Messenger confirms the send to the DataAdapter, allowing for appropriate data storage or cleanup.

Data Adapters

The DataAdapters prime function is to translate between data object formats, i.e. from RML to CMS, CMS to RML, and CMS to CMS PosDup. There is a single instance of DataManager, and it is thread and re-entrant safe.

To perform its function of translation, it needs to access the data from previous messages. To accomplish this, the single instance of DataManager is a private member of DataAdapter.

DataManager

The DataManager's sole function is to store relevant information regarding open orders and exchange message state to allow for proper transmission and translation of messages. All such information is maintained in the memory file map to allow for continuity between restarts.

As mentioned above, the single instance of DataManager is held within the DataAdapter. The DataAdapter is thread and reentrant safe by locking critical sections of code. The DataManager maintains state between successive executions by keeping all relevant information in a memory file map. Data is deleted from memory as soon as it is determined that the information is no longer needed for any future transitions.

On initialize, the memory file map is loaded and verified. If the file map is corrupted, initialization fails. In this case, the file map must be fixed, or wiped prior to

initialization.

The DataManager also keeps the sequence numbers for incoming and outgoing messages. Sequence numbers are relevant only for a single day and must be reset each day. The DataManager keeps the last initialization date to determine if the sequence numbers should be reset, or kept.

Memory File Map

The contents of the memory file map are exchange dependant. It contains a copy of incoming RML and CMS messages until the transmission is confirmed. The data stored is minimal, limited only to that needed to insure transmission. Note that it may be easier to keep the entire RML and CMS messages until the order is completed in some way.

Receiver

The receiver object manages the receiving Messenger threads. Receiver calls Connection.getMessage() to retrieve the next message from the Destination connection, then sends a Messenger off to process the message. All messages, even ACK and admin messages, are processed even if it does not result in an RML to the current system. This is required for tracking sequence numbers and connection state.

Messages are retrieved from the Connection rather than spawned from the Connection to allow for messages to queue as needed on the exchange side. This reduces the possibility for losing a message before it is stored in the DataManager.

Registry

The COMs configuration is retrieved from the systems Registry.

