BANKING TECHNOLOGIES ASIA MSC
FX, FINANCIAL FUTURES TRADING APPLICATION

IDEALBROKER

IdealBroker Technical Overview

1. The IdealBroker system

The IdealBroker is a system that allows users to monitor different symbol quotations, view them in a very user-friendly manner (charts), and buy/ sell symbols online. Buying/ selling is further divided into Orders (where you buy/sell at the current price), and Pending Orders (that order will be placed only when a certain limit is reached).

The IdealBroker system is divided into two parts:

- The IB server side. It resides on a server, not accessible (directly) to the users.
- The IB client side. It consists of the programs that users can use, in order to place trades, monitor symbols, etc.

The IdealBroker server consists of an underlying (Microsoft SQL) database, and the program itself. It runs on a Win2000 server machine. The database contains the latest symbol quotations, orders, pending orders, users, news, mails, etc. For analyzing purposes, we also keep a **history** of symbol quotations. This consists of two tables: one, which is updated so that it stores 1-minute quotations (last 3 months), and one that stores daily quotations (last 3 years). The **charts** that are visible on the IB client programs take their data from here.

The IB client sides consists of the IB Client (that allows users to view charts/trade), and the IB Manager (for management) and can run on Win98, Me, 2000, XP.

2. Connections

The client applications connect to the server using **TCP connections**. The server manages up-to-1000 users at a time (however, this is configurable at compile time).

The server has one-thread that monitors for connections from **new** users, and 10 threads that monitor for requests from **existing users**. Each such thread can monitor up-to 100 users at a time. Once a new user has successfully connected, it's assigned to one of the above 10 threads. In case all threads are full, we generate a "Too many users connected" error, in case a new user tries to connect.

All connections are encrypted using private/public keys. We follow this technique: first, a user issues a logon command in plain-text, presenting its login name. The server then sends back an (encrypted) answer that can be decrypted only with the users' private key. If the user can decrypt this, communication begins, and he's assigned to one of the above 10 threads. From now on, each message is encrypted twice: with user's public key + server private key (user will read it), or user's private key + server public key (server will read it).

2.1. Internet traffic

The connection protocol was designed to **minimize** Internet-traffic. When a request is answered, and the answer is too big, it will **automatically be zipped**. The client application cache data they receive, so that each answer will be minimal.

3. Data

On the server, all data related to symbols/trading (symbol quotations, orders, pending orders, etc) is kept in a database. All data (except for history data), is kept in memory as well. Updating follows this pattern: first, the DB is updated, then, if that succeeds, the in-memory copy is updated as well. This way, the in-memory copy is always synchronized with the DB. But most important, in case the system crashes due to reasons beyond our control (power failure, etc.), we can simply re-boot, restart the server, and it will pick up where it was last time – **nothing** is lost.

The important data client applications can inquire about consists of: history of symbol quotations, symbols, orders (normal orders/ pending orders/ closed orders), and price requests.

4. Commands

The commands that are issued by the clients are: GET, ADD, UPDATE, DELETE.

Except for history data, the logic for handling commands is explained below. For a:

- **GET**: the data is returned from memory (we have a duplicate of the database). It was developed with allow-for-**cache** in mind. Each GET request contains a time-entry. Based on that time-entry, we only return the objects that have the timestamp bigger than that time. Client applications have a cache, and at each start request only newest data.
- **ADD**, **UPDATE**: the database is updated. If this succeeds, we update our in-memory copy, and reply to the users, letting them know the operation succeeded.
- **DELETE**: the database is updated. If this succeeds, we update our in-memory copy, by re-reading the table from the database. However, note that the system was designed so that there will be **very few** delete commands. Indeed, the only need at this time for a DELETE command is when a user is deleted.

For history data, first of all, we don't allow ADD, UPDATE, DELETE commands (we handle all of this internally). In case a we have a GET request, there is a lot of data to retrieve. Thus, we **zip** the content, making the response up-to-20 times smaller. However, even this is not enough, when dealing with slow internet connections (dial-up). Therefore, we follow this convention:

- Once a day, the server creates small packs of data, zips them, and writes them as files. For 1-minute history quotes, each pack contains history data from a given day. For daily history quotes, one pack is enough (for 3 years, this means around 1000 entries, which zipped usually don't take more than 15Kb).
- Clients have their own cache (which first time is empty). Once a client connects, it checks its cache for the time of the latest entry. It sends a GET request, specifying that entry.
- The server analyzes the req-time, and: if the req-time is not from today (it specifies a time in the past), it returns the pack that contains that req-time (*read from its corresponding file*). Otherwise (it's today), reads today's data from the database (bigger than req-time), and returns it.
- The client, reads the answer, unzips it, processes it and places it in its cache. If it does not contain any data from today (the time of the latest entry read is in the past), makes

another request using the time of the latest entry read. This continues until the client has the most up-to-date history data.
- Using the technique underlined above, the client gradually fills its cache, and it does not take too much Internet bandwidth.
- Since there is a lot of data to retrieve, the cache is symbol-dependent. Each GET request, besides a time-entry, specifies a symbol for which to retrieve the data for (example: “AUDUSD”).

5. Forwarding of answers

On the server side, once a command is received, the server answers it, and based on the data this command refers to (order, symbol, etc.), it will forward the answer to **all** the users that are interested in it. For example, once a user adds an order, the broker will be notified as well.

This is a critical part of IdealBroker system, making sure that when something changes, all the interested parties will find out about it as soon as possible.

6. Symbols

Latest prices for symbols are loaded from an external tick-database source, and are inserted into our own database. At the same time, history data is updated, if necessary.

The server, once per second, sends to **all** connected users, all updated symbol information – bid/ ask/ open price/ close price/ high/ low (push technology). This insures that all clients have most up-to-date information at all times.

7. Automating of trading

A very important part of IdealBroker is the ability to trade. Based on how the market progresses (and your balance account) you can place orders on existing symbols.

There are two types of orders:
- normal orders: where you buy or sell, at the current price
- pending orders: where you buy or sell, when the price reaches a certain limit.

The IdealBroker system is designed with **automation** of opening/ closing of orders in mind. This is a critical feature, allowing for brokers to focus on important orders, while the small orders are handled automatically, and they just cash in.

Every time a user opens an order, it has to issue a price request from the broker. Same goes for closing of orders.

Automation means the ability of IdealBroker to answer to price requests **automatically**, without bothering the broker (but letting it know in case an order/ closing order took place).
On the server side, a separate module in the server takes care of automation: once a user issues a price request,
- if the user’s broker enabled automation, and the number of lots allows for automation, reply automatically (after a given delay)
- otherwise, ask the broker

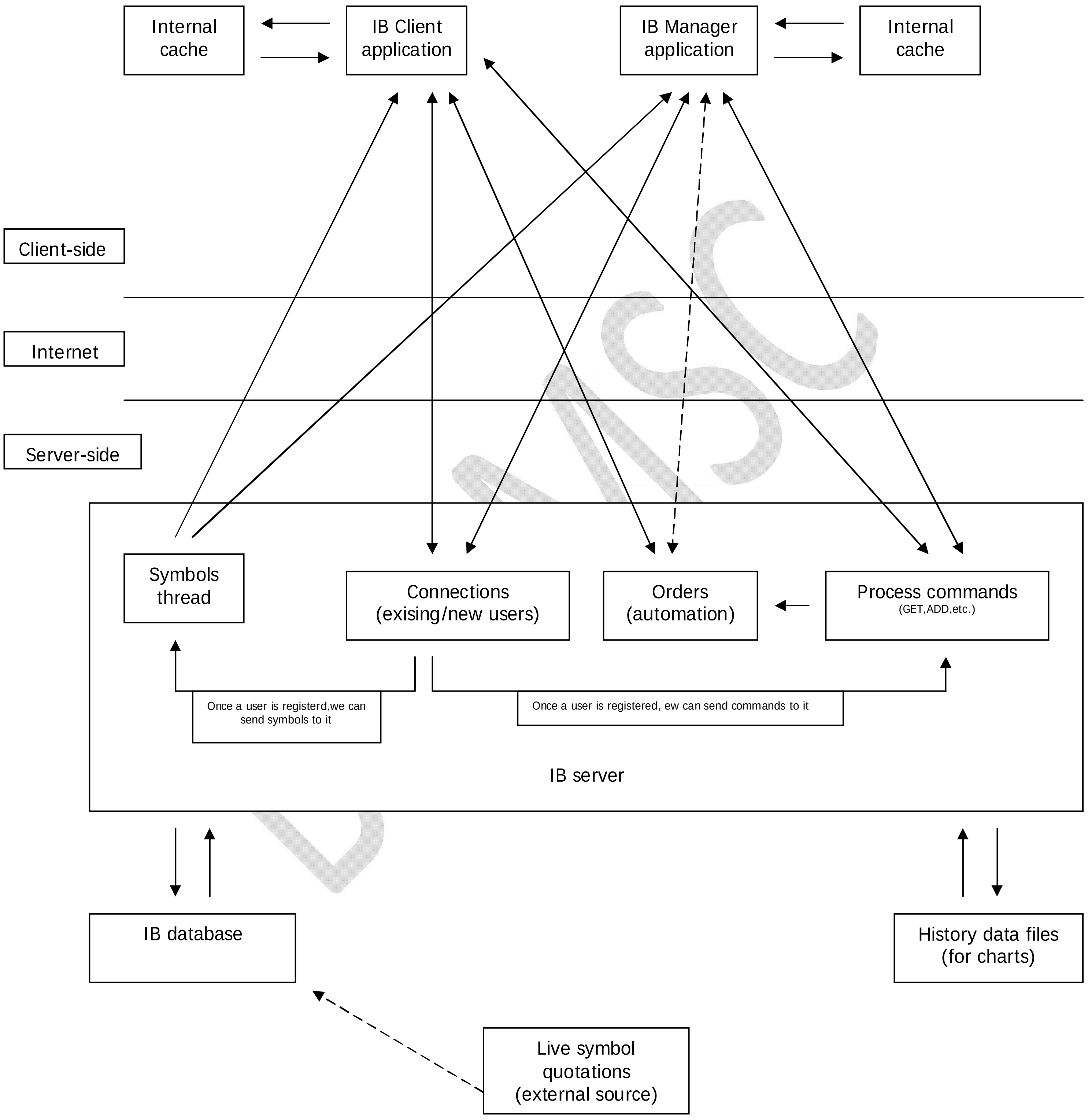
We have a separate thread that monitors all opened orders: once an order can be closed automatically,
- if the broker enabled automation and the number of lots allows for automation, close the order at the current symbol’s price, and notify the user and the broker

- otherwise, ask the broker for a closing price for this order. Wait for this closing price, and when it arrives, close the order, and notify the user and the broker.

8. Conclusion

The IdealBroker system has been designed from ground-up to minimize access to resources/ CPU, and minimize Internet traffic. The server is very scalable (up to 1000 concurrent users), using the latest Microsoft SQL server. The server makes extensive use of logs, so that any possible problems are caught as early as possible. It combines push/pull technologies, in order for users to have most up-to-date content.

The database can be upgraded to oracle, postgrel, mysql and various others and front end technology can also be upgraded to take advantage of current efficient technologies in areas of data streaming and dissemination. Various other potentials are possible, depending on the business and risk management strategies of the organization.


Internal cache
IB Client application
IB Manager application
Internal cache
Client-side
Internet
Server-side
Symbols thread
Connections (exising/new users)
Orders (automation)
Process commands (GET,ADD,etc.)
Once a user is registerd,we can send symbols to it
Once a user is registered, ew can send commands to it
IB server
IB database
History data files (for charts)
Live symbol quotations (external source)