2017

Socket Lab

LINUX AND WINDOWS, V1.1

WIKK

Problem statement

The system you shall develop is a real-time multiuser Market Position Tracking system. A stock market position is simply the number of shares owned by an individual. If a person buys 100 shares of IBM on day 1, sells 50 shares on day 2, and then buys another 25 shares on day 3, their position in IBM would be the following on each consecutive day.

Day 0 0 shares

Day 1 +100 shares (bought 100 shares)

Day 2 +50 shares (Day 1 position minus 50 shares sold)

Day 3 +75 shares (Day 2 position plus 25 shares bought)

If the number of shares sold is greater than the number of shares bought, the position is displayed as a negative number. The Stock Market Position Tracking System requires that the user identify themselves (with a simple name) , specify a stock ticker symbol (for example, IBM), an indication of whether the transaction is a buy or sell, and the number of shares bought or sold. The transaction is processed and the resulting position is sent back to the user.

The architecture used for this example is organized as follows (or if you like you exchange the roles). The real-time Stock Market Position Tracking System server resides on a Linux workstation and the client interface system resides on the Windows operating system.

The ideal stock tracking position system has the following features.

- Communication with the position server from multiple clients without affecting position update performance, i.e. try to use multithreaded if possible.
- Reliable transmission of transactions and subsequent transmission of position updates, i.e a TCP based model.
- The server maintains a simple list structure, or equivalent structure, of the user and his position as long as the server is up. This means you do not need to maintain a persistent structure.

When the client executes a transaction, a message is sent by the Windows system using sockets to the server. The server receives the message, casts it back to the original stock transaction structure, updates its internal stock position database, and returns the updated stock position to the client.

The Transaction Class

Both the Windows client and Linux server could use the following structure to transmit receive stock transaction records. To simplify message transmission, the structure for names and characters are comprised of single-byte character arrays when sending and receiving. Structure packing and byte ordering are then the same on Linux as on Windows NT for single-byte character arrays. Therefore, no additional translation needs to occur when delivering or receiving the message of them. But the amount is int (or long int) so you need to account for the differences.

OBSERVE the classes below are just suggestions!

```
#ifndef Transaction_H
#define Transaction_H

#include <iostream>
#include <string>
using namespace std;

class Transaction
{
public:
	// Default constructor
	Transaction();

	// Overloaded constructor.  Takes a user name, ticker symbol, buy/sell indicator,
	// and an integer number of shares to buy or sell.
	Transaction(string sUser, string sTicker, char cBS, long int liAmount);

	//Default destructor
	~Transaction();

	// Sets the position amount
	void SetAmount(long int iAmount);

	// The following methods are used to retrieve the private data members.
	string GetUser();
	string GetTicker();
	char GetBS();

	// The private data member iAmount is returned in its native format
	long int GetAmount();

	// for the console
	friend ostream& operator<<(ostream& out, Transaction& theTransaction);
	friend istream& operator>>(istream& in, Transaction& theTransaction);

private:
	string m_sUser;    // User name
	string m_sTicker;          // Stock ticker symbol
	char m_cBS;                  // Indicator for buy (B) or sell (s)
	long int m_liAmount; // The amount of stock bought or sold

};
#endif
```

The Client

The client is responsible for prompting the user for trade information, packing the trade information into a Transaction instance, and sending it to the Linux Stock Market Position Tracking System server. After a trade message is sent, the client suspends execution and waits for the server to

respond with a Transaction message containing the position update. The client displays the updated position and prompts the user for the input of another trade.

The Server

The Linux real-time Stock Market Position Tracking System server is responsible for keeping track of multiple user ticker positions. When a Transaction object is received, the components of the object are used to update the position server's internal database (in our case a code structure as described earlier). The updated ticker position is placed into the original Transaction object and send back to the client.

Suggestion for some helper clasess

```
#ifndef StockPosition_H
#define StockPosition_H

#include <string>
using namespace std;

// StockPosition is contains the position for a given ticker symbol.
class StockPosition
{
public:
	// The constructor.  Sets the initial position to 0.
	StockPosition();

	// The AddPosition method adds or subtracts a given amount from the
	// position depending on the cBS parameter.
	void AddPosition(long int liAmount, char cBS);

	// Returns the current position
	long int GetPosition();

	// Returns the ticker symbol
	string GetTicker();

	// Sets the ticker symbol
	void StoreTicker(string sTicker);

private:
	string m_sTicker;    // Holds the stock ticker symbol
	long int m_liPosition;    // Holds the stock position

};
#endif


#ifndef UserPosition_H
#define UserPosition_H

#include <string>
using namespace std;
```

```
#include "StockPosition.h"

// The UserPosition class stores the up to 50 different stock ticker symbols for a given user.
class UserPosition
{
public:
	// Default constructor. Sets the number of tickers to 0
	UserPosition();

	// Sets the user for the stock position portfolio
	void SetUser(string sUser);

	// Returns the user
	string GetUser();

	// Finds the stock position for a given ticker
	StockPosition* FindPosition(string sTicker);

	// Adds a new ticker position to the user's portfolio and returns the new ticker position.
	StockPosition* AddNewPosition(string sTicker, int iAmount, char cBS);

private:
	int m_iNumTickers;		// The current number of tickers stored
	string m_sTheUser;		// The user for which ticker positions are stored.
	StockPosition m_StockPortfolio[50];	// An array of 50 ticker positions
};
#endif
```

Spawning or Threading Applications That Use Sockets

One method used to create separate states of execution is spawning processes. Another is to create a new thread per client. After the server accepts a client connection, a new process/thread is created, dedicated to communication with the client. This new process uses the socket file descriptor created by accept for all data transferred with the client. This file descriptor is passed to the new process and the new process exists as long as the connection remains. Once the connection is broken, the process exits. The benefit of spawning processes is that the main process is dedicated to accepting connections. The process spawned performs the actual interplatform communication.

The steps required to create a multiprocess application that accepts multiple client connections are roughly the following.

- Create the server socket
- Bind the socket
- Listen to the socket.
- Accept a client connection
    - A new process/thread is created dedicated to the client communication

Once the client connection is accepted, a new socket file descriptor is created. A new process/thread is spawned/created and the new socket file descriptor is passed to it. The Linux version of a TCP

socket server forks a child process that is a duplicate of the parent. This spawned process can either be a new application that handles client communication or a call to a procedure within the original server process. (Recall that a forked process is actually a duplicate of the original.) If the spawned process is a procedure within the server process and not a new application, the address of the new socket instance containing the new socket file descriptor can be passed directly to it. If the spawned process is a new application, the new socket file descriptor must be passed on the command-line. The socket file descriptor must be passed instead of the socket instance because C/C++ objects cannot be passed application arguments. The Windows operating system does not have the feature of forking processes; instead, new processes can be spawned. Regardless of the operating system, the original TCP socket server loops and waits to accept additional client connections.