Stocks@home

Volunteer Computing and Wall Street

Objective

Create a volunteer computing application for stock analysis

Abstract

Web browsers are some of the most frequently used computer applications today. However, a large portion of their data cycles is wasted during idle time. The purpose of this project is to create a browser page that can take advantage of these wasted cycles, using them to analyze stock price trends. Multiple machines will analyze individual stocks and send their results to a central server, which will then determine the most profitable investment strategies from the compiled data.

Following the naming convention of popular volunteer computing projects such as SETI@home, this program will be called stocks@home. Stocks@home has the potential to have processing power of a supercomputer for a fraction of the cost due to the prevalence of consumer computers and web browser users. The only limit to its computational capabilities is the number of contributing users.

Introduction

- The large demand for processing power is often fulfilled by expensive supercomputers
- Volunteer computing: A central server sends many clients small tasks, receives responses from those clients, and compiles those responses into a useful result
- In this case, each client is a device whose processing power has been donated by a volunteer
- Benefits of volunteer computing:
- Little upkeep cost, volunteers pay for power costs
- Little startup cost, volunteers pay for devices
- Self-sustaining, volunteers maintain their own devices
- Self-adapting, volunteers naturally upgrade their own devices or upgrade the hardware/software in their devices

Background Info

- Quantitative analysis: Prediction of stock price trends solely using historical pricing data
- MACD analysis: method of quantitative analysis of stock price trends
- Node.js is an open-source, cross-platform runtime environment for server-side applications.
- Asynchronous I/O model: makes use of non-blocking calls to keep programs running efficiently.
- Async functions can be started and set aside until they are complete
- In the meantime, the program moves onto other methods
- The program continuously moves forward when using async methods

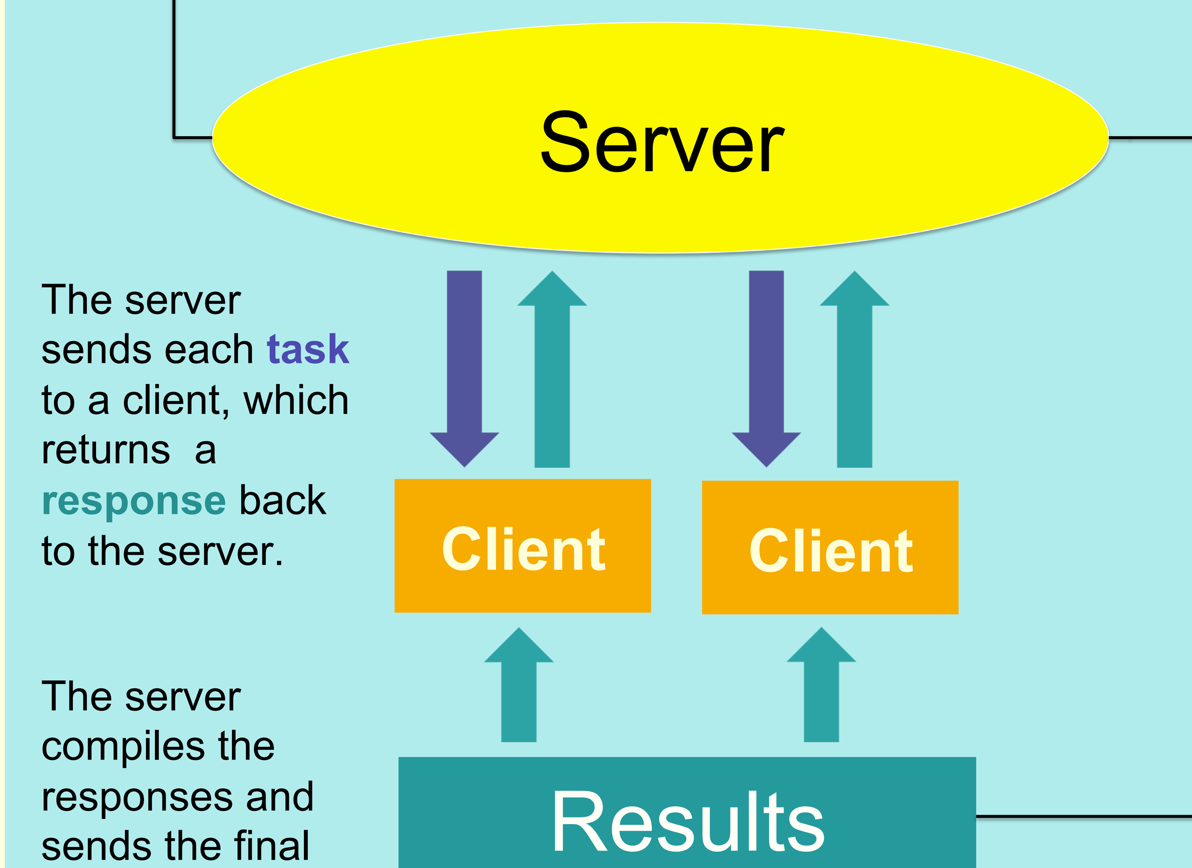
Procedure

1. Install node.js
2. Implement server framework
3. Program function to calculate exponential moving averages
4. Program MACD analysis function
5. Implement requestTask request handler to distribute tasks to the client from the server
6. Implement reportResult request handler to collect responses from the clients for analysis
7. Create code for client html webpage
8. Program an xmlhttp request in the client code to call the requestTask handler in order to obtain the stock ticker for analysis.
9. Implement a callback in this request to perform another xmlhttp request to get and analyze the stock data stored on the Amazon web server using the obtained stock ticker
10. Implement another callback to launch the reportResult handler to post the response to the server once calculations are completed
11. Have the requestTask function call itself recursively until the server sends out an 'all-done' message
12. Display result to clients
13. Compile stock data from finance.yahoo.com and upload it to the Amazon web server
14. Run finished program and record the time taken to analyze all the stocks in the server for different numbers of clients

Design

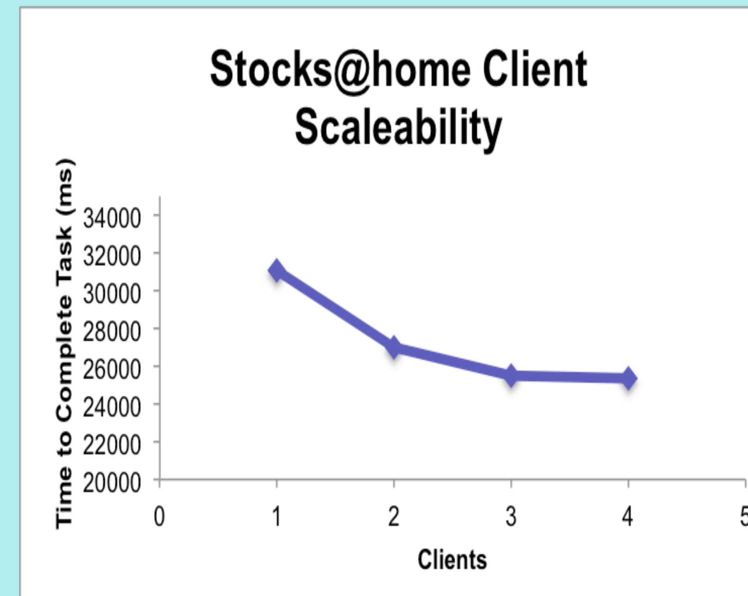
Tasks

Each task is stored in the server, ready to be sent out for completion



*This all occurs asynchronously

Data



Clients	Time Taken to Complete (ms)			
	Trial 1	Trial 2	Trial 3	Mean
1	31118	31734	30463	31105
2	27086	27013	26823	26974
3	26046	24819	25616	25494
4	25718	24924	25361	25335

```
localhost:8888/start
Apps  School  Sports  ACU  FIFA  http://localhost:8888

My Project!

This is my senior project...

Analysis Complete...
AZO: Sell, price decrease incoming.
ACT: Invest, price increase incoming.

Elements  Network  Sources  Timeline  Profiles  Resources  Audits
<top frame>  Preserve log
0.9212184015269193, 1.0345912158095762, 1.1505256734153764, 1.141688
0.9987085079214462, 0.7883385874355597, 0.41435711949225795, 0.08347
-0.15143947725807072, -0.059952487226717865, -0.15477513679194121, -
Sending data to server...
loading Stock URL...
Stock URL retrieved: http://s3.amazonaws.com/victor-sui/EQR.csv%179
count= 179
Performing calculations...
[90.13, 93.03, 91.61, 91.93, 91.08, 93.8, 95.31, 96.49, 96.92, 94.8,
96.19, 95.19, 96.31, 92.52, 89.03, 88.91, 88.1, 93, 92.33, 90.79, 90
86.53, 87.74, 85.06, 84.51, 87.46, 86.88, 83.68, 84.21, 86.68, 92.24
93.82, 94.05, 96.41, 95.06, 95.02, 92.45, 91.14, 85.47, 85.34, 86.37
88.19, 90.75, 93, 91.8, 90.3, 87.24, 86.72, 94.07, 96.74, 99.56, 101
96.35, 98.16, 95.85, 97.56, 98.75, 98.5, 99.55, 98.97, 96.1, 90.28,
91.5, 88.43, 92.1, 93.4, 92.07, 96.03, 92.76, 91.22, 91.27, 88.17, 8
94.56, 93.61, 95.09, 96.57…]

Received POST data chunk '0.0820152138585351l%202'.
About to route a request for /reportResult
Request handler 'reportResult' was called.
Value is 0.0820152138585351l of GLW
Request for /requestTask received
About to route a request for /requestTask
Request handler 'requestTask' was called.
Finished
Request for /reportResult received
Received POST data chunk '0.296232536224483 3%203'.
About to route a request for /reportResult
Request handler 'reportResult' was called.
Value is 0.296232536224483 3 of STZ
```

*Tested on a mid-2010 MacBook Pro on Safari 8.03. Smaller completion times indicate greater processing speeds.

Analysis

The data were collected on a single device running multiple instances of the Stocks@home web client hosted locally to emulate multiple devices collaborating over the internet.

The time taken for the Stocks@home server and clients to complete stock analysis gradually decreased as more clients were added. However, as more clients were added the test device struggled to provide the resources needed to drive each client. A performance plateau was experienced at four clients. The system's CPU ran at 100% during all three four-client trials.

Sample Code

```
236
237 function requestTask(response, postData) {
238   console.log("Request handler 'requestTask' was called.");
239   // This method returns the URL containing the particular stock data
240   // to be distributed for analysis
241   // The client calls this method to receive its task.
242   if(count<tickers.length) {
243     console.log('Count is ' +count);
244     var string = 'http://s3.amazonaws.com/victor-sui/'+tickers[count]+'.csv%'+count;
245     response.writeHead(200, {"Content-Type": "text/plain"});
246     response.end(string);
247     console.log('URL sent: '+string);
248     count = count + 1;
249   }
250   else {
251     console.log('Finished');
252     response.writeHead(200, {"Content-Type": "text/plain"});
253     response.end("FINISHED");
254   }
255 }
256
```

```
80  function getStockUrl(callback){
81    var xmlhttp;
82    var dat;
83
84
85    // This is the function the client calls to request a task, which is in this case
86    // a stock to be analyzed. xmlhttp.open is an example of an asynchronous, or
87    // non-blocking, method.
88    xmlhttp=new XMLHttpRequest();
89
90    xmlhttp.open("GET","http://localhost:8888/requestTask",true);
91
92    xmlhttp.onreadystatechange = function() {
93      if(xmlhttp.readyState==4 && xmlhttp.status==200) {
94        if(xmlhttp.responseText=="FINISHED") {
95          complete=true;
96          console.log("FINISHED");
97          //break;
98        }
99        if(!complete) {
100         console.log('Stock URL retrieved: '+xmlhttp.responseText);
101
102         count++;
103         console.log("count= "+count);
104
105         $("#content").html(xmlhttp.responseText.split('%')[0]);
106         count=parseInt(xmlhttp.responseText.split('%')[1]);
107         dat = getStockData(xmlhttp.responseText.split('%')[0], callback, parseInt(xm
108       //callback(dat);
109         getStockUrl(done);
110       }
111
112     }
113     else
114       console.log('loading Stock URL...');
115
116   }
117
118   xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
119   xmlhttp.send();
120 }
```

Conclusion

Stocks@home takes less time to complete the stock analysis as more clients are added and is a successful implementation of the volunteer computing design. With further testing and design, it could potentially analyze thousands of stocks everyday with little-to-no upkeep costs.

Future Research

- Further optimizing code
- Implementing security measures to prevent malicious/ fraudulent responses
- Testing over multiple devices and device types
- Testing with greater volumes of data
- Using real-time price data as opposed to archived data

References

Anderson, D.P.; Korpela, E.; Walton, R., "High performance task distribution for volunteer computing," *e-Science and Grid Computing, 2005. First International Conference on* , vol., no., pp.8 pp.,203, 1-1 July 2005 doi: 10.1109/E-SCIENCE.2005.51

Stefan Tilkov, Steve Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs", *IEEE Internet Computing*, vol.14, no. 6, pp. 80-83, November/ December 2010, doi:10.1109/MIC.2010.145