# ECE419S: Distributed Systems

*Winter 2010*

Assignment 1

Submission deadline: Tuesday, February 2nd 11:59 P.M.

Cristiana Amza
amza@eecg.toronto.edu

# 1 Introduction

This lab assignment serves as a first exercise towards distributed systems. You will be implementing a simple *Broker* which provides stock quotes to users. The first three labs (Lab 1, Lab 2, Lab 3) will allow you to gain hands-on experience in building distributed systems. In this lab, you will be implementing the lab using sockets. In assignment 3, you will be re-implementing the same functionality using CORBA.

In this assignment, you are required to use the workstations in the GB 243 lab. You need basic knowledge of Java to solve problems in this assignment. To help you, we are providing a `ECHO` server as an example. It is recommended that you understand the `ECHO` example and use it as a template to build your implementation of *Broker*.

The code and starter files are placed with the ECE419 directory, at `/cad2/ece419s/` referred as `${ECE419_HOME}`. The source code for the `ECHO` example is placed on the *ug* machines at `${ECE419_HOME}/labs/lab1/echo/`. Your `Makefile` and sample scripts are already configured to use some Java paths. However, there is nothing specific to this Java version that the code (including our sample code provided to you) requires.

**NOTE: To ease our task, if you plan to run the code on your home machine, you will have to update the path to Java (i.e. `${ECE419_HOME` and `${JAVA_HOME}`).**

## 1.1 Grading

This lab assignment represents 7% of your final grade for the course and is due by Tuesday, February 2nd 11:59 P.M. It consists of three parts. Each part is graded according to the following scheme:

- Full credit for having all components of the part working according to the objectives.
- 80% if your program has a minor functional or logical mistake.
- 40% if your program is partly working but has several or major functional or logical mistakes.
- Otherwise, 20% if the code compiles successfully. To receive this credit, your code has to show "reasonable effort" towards the objective of the lab part and any compilable code will not award you this credit :)

Each lab part consists of one or more client-server sessions. The user issues appropriate "commands" at the client side(s) and then receives a response according to success and failure of the command. You can assume the user enters the commands in the right format. Sample scenarios are provided for each part. Some Makefile and wrapper shell scripts are provided for you for each part which are organized under **broker1**, **broker2** and **broker3** respectively. You must not use any "hardcoded" host name, port number or other configurable parameter within your source code or the shell scripts. Such parameters have to be provided as command line parameters instead. See the content of each shell script to learn about such parameters.

**Important Note:** You are free to develop your code on other platforms but you MUST "compile and test" your work on *ug* facilities of GB 243 lab. Your Makefiles and shell scripts have to be properly setup for each part and unused files removed from the submission directories. The Windows users have to pay extra attention to compilation and test of their work as sometimes their working source codes would not compile on *ug* because of the differences between Unix and Windows text file formats.

# 2 Part 1 - Broker Assisted Stock Quotes

## 2.1 Objectives

You are required to build a *Broker* server, *OnlineBroker*, that serves quotes to clients. The interface is a simple request-reply protocol where the client provides the stock symbol and the *OnlineBroker* responds with the current stock price.

We provide the *BrokerPacket* class as the packet format of the messages exchanged between the clients and brokers. The same class (*BrokerPacket*) will be used for all three parts of the lab and you are not required to change it further. Therefore, there may be certain members of the packet that you don't use for each lab part.

The BrokerPacket class has been heavily commented to aid you in your lab. Use this as a guide.

The client obtains the symbol from user input, queries the *OnlineBroker*, and outputs the result. Here the user command is simply the stock symbol. The client reflects its readiness for accepting a new command by prompting '>'. A typical session is as follows:

```
[amza@ug145 part1]% sh ./client.sh ug55 8000
Enter queries or x for exit:
> MSFT
Quote from broker: 28
> SUNW
Quote from broker: 5
> ORCL
Quote from broker: 16
> ATY
Quote from broker: 0
> x
[amza@ug145 part1]%
```

### 2.2 Implementation Notes

*OnlineBroker* stores the mapping between the symbols and their quotes in a plain text file. The following is an example:

```
MSFT 28
CSCO 13
ORCL 16
SUNW 5
INTC 18
```

When a quote is requested from the client, the *OnlineBroker* does a table lookup from the file (or alternatively from its buffer caching the contents of that file), and returns the current quote for the requested symbol. The requested symbols are not case sensitive. In this part, we ignore all failure handling details. If a symbol cannot be found, *OnlineBroker* simply returns 0.

### 2.3 Grading

The weight of this part in the overall Lab 1 grade is 20%. In the ECE419 directory (`${ECE419_HOME}`), we have provided some relevant files to you. These include a sample Makefile, as well as the shell scripts we use to launch the client or the server (`client.sh` and `server.sh`, respectively) for testing your code. Each of these shell scripts will launch your Java code and pass all the required command line parameters, which are passed to the shell script, down to your Java code. You should name the main class of your client and broker codes according to the content of each script, *i.e.*, `BrokerClient.java` and `OnlineBroker.java`. The provided `nasdaq` file is the table of quotes for *OnlineBroker*.

## 3 Part 2 - Online Brokers and Stock Exchanges

### 3.1 Objectives

Your goal in this part is to extend the basic client-broker relationship implemented in the previous programming assignment, and add a stock exchange, which is referred to as an *exchange* from now on, to the system. The

exchange is responsible for contacting the *OnlineBroker* to add recently IPOed symbols, remove delisted symbols and update the quote for a specific symbol which are respectively done through **add**, **remove**, and **update** user commands. In this part, you also add failure handling according to the errors defined in *BrokerPacket*.

## 3.2 Implementation Notes

Failures aside, it should be straightforward to extend the system since the exchange is just another client to the *OnlineBroker* in the system. There are three types of failures that we are interested in: *InvalidSymbol* is used when the symbol is not found. *QuoteOutofRange* is used when the exchange tries to update with an out-of-range quote (assume the range is [1, 300]). *SymbolExists* is used when the exchange tries to add an existing symbol to the broker. The challenge here is to properly handle the errors in the client and output the errors to the user. Note that the content of symbol/quotes file, *i.e.*, `nasdaq`, has to be updated when the *OnlineBroker* exits.

## 3.3 Grading

The weight of this part in the overall Lab 1 grade is 40%. When grading your submission for this part, a client and an exchange will be started on different machines.

The exchange will try to update, remove and add symbols via simple commands, such as:

```
[amza@ug145 part2]% sh ./exchange.sh ug42 7777
Enter command or quit  for exit:
> add EMC
EMC added.
> update EMC 4
EMC updated to 4.
> add SUNW
SUNW exists.
> remove SUNW
SUNW removed.
> update SUNW 19
SUNW invalid.
> remove SUNW
SUNW invalid.
> update MSFT 709
MSFT out of range.
> x
[amza@ug part2]%
```

The typical session at the client would be:

```
[amza@ug145 part2]% sh ./client.sh ug42 7777
Enter symbol or quit for exit:
> EMC
Quote from broker: 4
> MSFT
Quote from broker: 28
> SUNW
SUNW invalid.
> x
```

A quote for EMC is obtained after the update is issued from the exchange, and SUNW is invalid since SUNW is removed previously by the exchange.

Similar to the previous part, a Makefile, an initial symbol/quotes file, and shell scripts are provided for you. Name your exchange's main class `BrokerExchange` and use `exchange.sh` to execute it.

Please note that the file `nasdaq` should be copied to your working directory so that it is writable by the *OnlineBroker*. Any changes made during a session should then be reflected in the `nasdaq` file.

# 4 Part 3 - Online Stock Quotes System

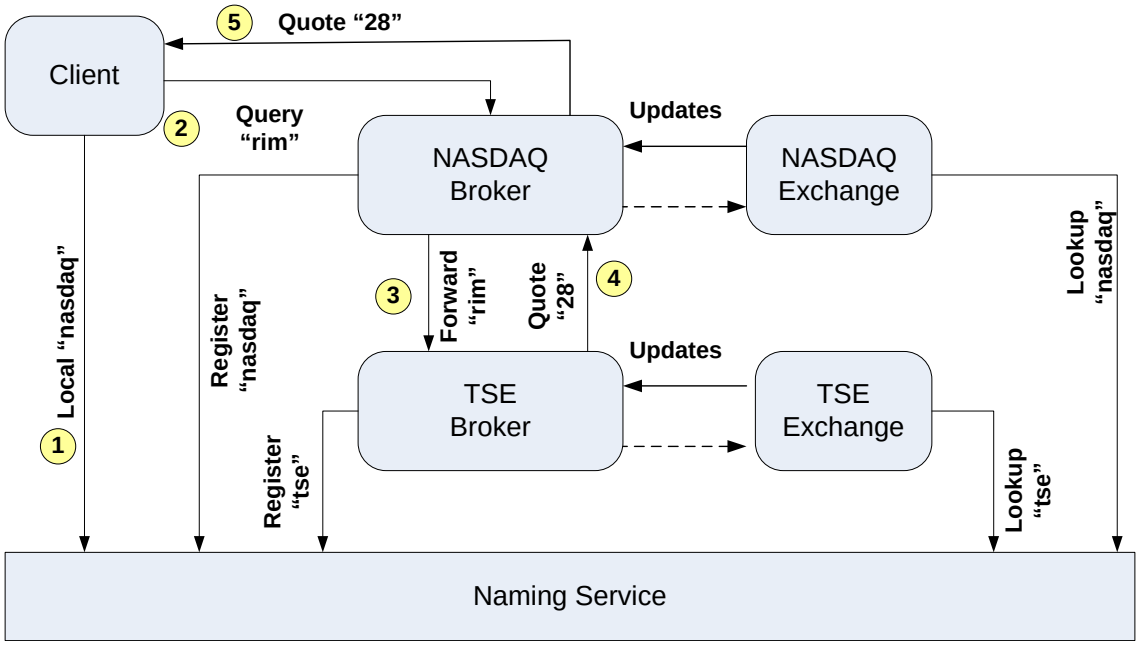


Figure 1: An Online Stock Quotes System

## 4.1 Objectives

In this part, you are challenged to implement a relatively complete online stock quotes system. The enhanced system is illustrated in Figure 1.

The online stock quotes system consists of two stock exchanges, the TSE exchange and the Nasdaq exchange, and two corresponding brokers, the TSE broker and the Nasdaq broker. The client does not know about the locations of the brokers. It uses a naming service, and looks up the broker by their names, i.e. `nasdaq` and `tse`. The client then sends its queries to its local broker (which may be the TSE or Nasdaq broker, since the client may move). The local broker may be changed via a `local` command.

You have implemented much of the system in Part 1 and Part 2. In this part, we are making the system complete with support for query forwarding. Figure 1 shows an example. First, the client connects to a broker by issuing the "local" command. For example, let us assume that the client connects to Nasdaq. When the client issues a query for "rim" to the Nasdaq broker, the broker receives the request, and looks up the symbol in its local list of symbols. If a quote is obtained locally, it is returned to the client. In this case, "rim" is not found since the stock is listed in the TSE exchange. If the symbol is missing (when we previously returned an *InvalidSymbol* error in Part 2), the broker tries to look up the symbol in the other broker (in our example the TSE broker), and sends the result back to the client. In this case, the first broker that the client contacts is both a server (by serving the client) and a

client to the other remote objects. The forwarding complexity is transparent to the client but it eventually receives the result without knowledge of which stock exchange the symbol is listed in.

With respect to the updates, for simplicity we may assume that each broker only receives updates from a single stock exchange of the same geographic location, e.g., the Nasdaq broker receives from the Nasdaq exchange. The exchanges are clients to the brokers, and they also need to resolve the names of broker objects via the naming service. Once the names are resolved, they proceed to update the symbols as in Part 2. Failures are handled in the same way as in Part 2. The packet exchanged between the client/broker/exchange is the same as before. Note that each *OnlineBroker* is exclusively launched for TSE or Nasdaq and is only allowed to access and update its corresponding symbol/quotes file. Similarly, each exchange is dedicated to TSE or Nasdaq exclusively.

### 4.2 Grading

The weight of this part in the overall Lab 1 grade is 40%. An interactive session involving a naming server, two exchanges, two brokers and one client will be used to grade your work. If the semantics of your output conform to the specifications, you get full credits. Otherwise, similar grading policies as in the previous parts will be used.

A new command `local` is introduced to the client. The client session may now look like the following:

```
[amza@ug145 part3]% sh ./client.sh <possible-additional-parameters>
Enter command, symbol or x for exit:
> local tse
tse as local.
> EMC
Quote from broker: 4
> RIM
Quote from broker: 1
> local nasdaq
nasdaq as local.
> MSFT
Quote from broker: 28
> RIM
Quote from broker: 1
> x
[amza@ug145 part3]%
```

Similar to the file `nasdaq` of previous parts, you are provided with two files of `nasdaq` and `tse` as the starting points of your sessions. The example sessions of the exchanges (Nasdaq and TSE) are identical to those in Part 1. Both brokers and exchanges specify the location (`nasdaq` or `tse`) at the command line when started. Note that the provided shell scripts pass down all the required command line parameters down to your Java code. You need to look into the content of each shell script to learn about these parameters. You should name your main classes for client/broker/exchange based on what the related shell script expects as the main class. Since each script passes all the required command line parameters to your Java code, you should not need to change the script. A new script `lookup.sh` is provided for you to launch your naming server which should have its main class called *BrokerLookupServer*.

You should use a single exchange code but launch it in TSE or Nasdaq role by passing a command line parameter. Similarly, you will have a single *OnlineBroker* code stream which will be configured at runtime to be a TSE or Nasdaq broker through a command line parameter.

As a helpful note, use the provided script files as a guide to organize and write your code.

## 5 Instructor Notes and Advice

This lab is designed with two goals in mind: i) to allow you to learn and/or practice basic skills necessary in the development of a distributed system, such as socket-based communication, and ii) to put you into the right mind-set

necessary for building a real-life distributed system.

While other labs you have done have prepared you well to fulfill my first goal (although what it takes to achieve this goal will be, as usual, and by necessity, somewhat tedious), you are perhaps unprepared to tackle the tasks you need to do for my second goal. Building a distributed system normally requires hard work, but also requires quite a bit of creativity. Typical University assignments you have handled in the past fully specify what you have to do to get the mark, and also exactly how to do it, hence hard-wire your problem and your solution too. This is very different from real-life, where even your problem is usually not fully specified, let alone the solution.

In Lab 3, you will have full freedom to define part of the problem and also its full solution. To prepare you for that, consider Lab 1 as a (small) stepping stone, where things are almost fully specified, i.e., you are given a tiny bit of flexibility.

Consider the requirements of this lab similar to the specification you would receive in real-life. The specification, just like an open standard of a distributed system, never gives details on *how* to implement a certain requirement. The implementation is fully up to you and is expected to be substantially different from implementor to implementor (but still conform to the specification). Furthermore, due to the complexity of distributed systems, a new system specification will ever fully indicate what you should do in every single situation. Use the following common sense rule: if something is explicitly asked for, you have to implement it (in your chosen way) in order to get the full mark, otherwise you don't have to do it. You can do additional things only when/if you find the exercise interesting. In particular, there is nothing cast in stone in the example code we give. You should follow the specification of what you need to do provided in this hand-out, not the code excerpt we provide. You can even ignore the sample code entirely and use your own to implement the specification given.

As a concrete example, let's take the requirement "the symbol/quotes file has to be updated when the *Online-Broker* exits."

This only specifies the (required) presence of the updated file at the end of your run. How you provide it, if there is no explicit requirement about the data freshness, and there isn't, is completely up to you e.g., by doing a shut-down at exit and flushing the file upon shut-down, by flushing data to the file periodically, etc. Along the same lines, the example of the working system given in this lab hand-out is a high-level one. There is no precise, detailed explanation of how the system works or should work underneath, only a sketch is given. This is the case for all other details you may be unsure of, such as, errors that you want to report besides the ones mentioned in this hand-out, if any, specific messages that you want to use to report those errors, etc. If nothing has been explicitly asked for, you are free to implement the error types/codes and error messages you want.

**Remember: not specified means not required !**

# 6 Submission

Your submission should be divided into three parts (organized in three separate directories `broker1`, `broker2` and `broker3`). Each part should include an appropriate `Makefile`. You can feel free to use the sample makefiles that are provided to you, or you can write your own makefiles.

Your submission must be in the form of one compressed archive named Firstname.Lastname.tar.gz (for example, John.Smith.tar.gz). The directory structure of the archive should be as follows, assume that your name is John Smith:

```
John.Smith.Lab1/
---README
---+broker1/
------README
------client.sh
------server.sh
------Makefile
------nasdaq
------<Java files>
---+broker2/
```

```
------README
------client.sh
------server.sh
------exchange.sh
------Makefile
------nasdaq
------<Java files>
---+broker3/
------README
------client.sh
------server.sh
------exchange.sh
------lookup.sh
------Makefile
------nasdaq
------tse
------<Java files>
```

For each part, the compressed archive is required to include a `README` file if and only if your code requires a custom setup to run, *i.e.,* other than execution through the provided scripts. The provided scripts should be good enough for the purpose of this lab unless you want to pass extra parameters to your code for some added features. Each part has to have a `Makefile`, along with all source files required to compile for that part. When we check your solutions for each part, we simply enter the corresponding subdirectory, and type `make` at the command line prompt to compile your source code. We will then use `sh client.sh` or similar commands to run the client, the server or the exchange, with additional command-line options that you may specify in the `README`.

Once you have a compressed archive in the .tar.gz format, you may submit your solution by the deadline using the submitece419s command: `submitece419s 1 Firstname.Lastname.tar.gz`

For example: `submitece419s 1 John.Smith.tar.gz`

If you need to provide any additional explanations about your solutions, please do so in the general `README` file in the topmost directory, illustrated previously.