Homework 2: Programming Component

SCALABLE SERVER DESIGN: USING THREAD POOLS TO MANAGE AND LOAD BALANCE ACTIVE NETWORK CONNECTIONS

VERSION 1.0

DUE DATE: Wednesday March 7th, 2018 @ 5:00 pm

As part of this assignment you will be developing a server to handle network traffic by designing and building your own thread pool. This thread pool will have a configurable number of threads that will be used to perform tasks relating to network communications. Specifically, you will use this thread pool to manage all tasks relating to network communications. This includes:

1. Managing incoming network connections
2. Receiving data over these network connections
3. Sending data over any of these links

Unlike the previous assignment where we had a receiver thread associated with each socket, we will be managing a collection of connections using a fixed thread pool. A typical set up for this assignment involves a server with a thread pool size of 10 and 100 active clients that send data over their connections.

1 Components

There are two components that you will be building as part of this assignment: a server and a client.

1.1 Server Node:

There is exactly one server node in the system. The server node provides the following functions:

A. Accepts incoming network connections from the clients.
B. Accepts incoming traffic from these connections
C. Replies to clients by sending back a hash code for each message received.
D. The server performs functions A, B and C by relying on the thread pool.

1.2 The Clients

Unlike the server node, there are multiple Clients (minimum of 100) in the system. A client provides the following functionalities:

(1) Connect and maintain an active connection to the server.
(2) Regularly send data packets to the server. The payloads for these data packets are 8 KB and the values for these bytes are randomly generated. The rate at which each connection will generate packets is **R** *per-second*; include a Thread.sleep(1000/ **R**) in the client which ensures that you achieve the targeted production rate. The typical value of **R** is between 2-4.
(3) The client should track hashcodes of the data packets that it has sent to the server. A server will acknowledge every packet that it has received by sending the computed hash code back to the client.

2 Interactions between the components

The client is expected to send messages at the rate specified during start-up. The client sends a byte[] to the server. The size of this array is 8 KB and the contents of this array are randomly generated. The client generates a new byte array for every transmission and also tracks the hash codes associated with the data that it transmits. Hashes will be generated with the SHA-1 algorithm. The following code snippet computes the SHA-1 hash of a byte array, and returns its representation as a hex string:

```
public String SHA1FromBytes(byte[] data) {
    MessageDigest digest = MessageDigest.getInstance("SHA1");
    byte[] hash  = digest.digest(data);
    BigInteger hashInt = new BigInteger(1, hash);

    return hashInt.toString(16);
}
```

A client maintains these hash codes in a linked list. For every data packet that is published, the client adds the corresponding hashcode to the linked list. Upon receiving the data, the server will compute the hash code for the data packet and send this back to the client. When an acknowledgement is received from the server, the client checks the hashcode in the acknowledgement by scanning through the linked list. Once the hashcode has been verified, it can be removed from the linked list.

The server relies on the thread pool to perform all tasks. The threads within the thread pool should be created just once. Care must be taken to ensure that you are not inadvertently creating a new thread every time a task needs to be performed. There is a steep deduction (see Section 4) if you are doing so. The thread pool needs methods that allow: (1) a spare worker thread to be retrieved and
(2) a worker thread to return itself to the pool after it has finished it task.

The thread pool manager also maintains a list of the work that it needs to perform. It maintains these work units in a FIFO queue implemented using the linked list data structure. Work units are added to the tail of this work queue and when spare workers are available, they are assigned work from the top of the queue.

Every 20 seconds, the server should print its current throughput (number of messages processed per second during last 20 seconds), the number of active client connections, and mean and standard deviation of per-client throughput to the console. In order to calculate the per-client throughput statistics (mean and standard deviation), you need to maintain the throughputs for individual clients for last 20 seconds (number of messages processed per second sent by a particular client during last 20 seconds) and calculate the mean and the standard deviation of those throughput values. This message should look like the following.

[timestamp] Server Throughput: x messages/s, Active Client Connections: y, Mean Per-client Throughput: p messages/s, Std. Dev. Of Per-client Throughput: q messages/s

You can use these statistics to evaluate the correctness of your program.

If your server is functioning correctly (assuming it is adequately provisioned), the server throughput should remain approximately constant throughout its operation. Note that it will take some time to reach a stable value due to initialization overheads at both server and client ends. Mean per-client throughput multiplied by the number of active connections should be approximately equal to the sever throughput. Furthermore, if every client is sending messages at the same rate and the server's scheduling algorithm is fair, the standard deviation of the per-client throughput should be low.

Similarly, once every 20 seconds after starting up, every client should print the number of messages it has sent and received during the last 20 seconds. This log message should look similar to the following.

```
[timestamp] Total Sent Count: x, Total Received Count: y
```

3 Command line arguments for the two components

Your classes should be organized in a package called **`cs455.scaling`**. The command-line arguments and the order in which they should be specified for the Server and the Client are listed below

```
java cs455.scaling.server.Server portnum thread-pool-size

java cs455.scaling.client.Client server-host server-port message-rate
```

4 Deductions

There will be a **16-point deduction** if any of the restrictions below are violated.

1. You cannot use the Executor interface or any of the thread pool classes that are part of the `java.util.concurrent` package
2. You cannot use third-party implementations of the thread pool. This is something you must implement all by yourself.
3. Threads should be pre-allocated when the server component starts and live the duration of program execution. Do **NOT** create new Threads for each new message.
4. The stop, suspend, and resume Thread methods should not be used. These methods are deprecated and can cause concurrency bugs.
5. The data that you will be sending will be byte[]. None of your classes can implement the `java.io.Serializable` interface
6. No GUIs should be built under any circumstances. These are auxiliary paths and the deduction is in place to ensure that none of you attempt to do this.

5 Grading

Homework 2 accounts for 20 points towards your final course grade. The programming component accounts for 80% of these points with the written element (to be posted later) accounting for the remaining 20%. This programming assignment will be graded for 16 points. The point distribution for this assignment is listed below.

Server Node - 11 points
Client Node - 5 points

Server Node Breakdown:

3 points:	Correct thread pool manager implementation
2 points	Abstracting different types of tasks as work units for worker threads
2 points:	Receiving data from Clients
1 point:	Sending hash codes back to Clients
1 point:	Using the thread pool to manage connections
2 points:	Supporting 100 concurrent client connections

Client Node Breakdown:

1 point	Generation of random data.
2 points:	Maintaining linked list data structure of pending hashes.
1 point:	Sending data to the Server.
1 point:	Receiving and verifying responses (hashes) from the Server.

6 Milestones:

You have 4 weeks to complete this assignment. The weekly milestones below correspond to what you should be able to complete at the end of every week.

Milestone 1: Complete a partial implementation of the Thread Pool Manager. This should allocate a given number of threads, maintain a queue of pending tasks, and assign tasks to be handled by the threads. This component can be created and tested in isolation from the rest of the system.

Milestone 2: Use the **`java.nio.channels.Selector`** class to register and deregister incoming SelectionKeys on the server side. You should also be able to create randomized data packets as byte arrays.

Milestone 3: The server should be able to read incoming data from multiple clients, and clients should be able to send data.

Milestone 4: Your server implementation should use the thread pool to handle clients and be able to manage up to 100 concurrent connections, and send replies back to clients that send data to the server.

7 What to Submit

Use the CS455 checkin program to submit a single .tar file that contains:

- All the Java files related to the assignment (please document your code)
- A Makefile that performs both a `make clean` as well as a `make all`,
- A README.txt file containing a description of each file and any information you feel the GTA needs to grade your program.

The folder set aside for this assignment's submission using checkin is **HW2-PC**

Filename Convention: The class names for your client and server should be as specified in Section 3. You may call your support classes anything you like. All classes should reside in a package called **`cs455.scaling`**. The archive file should be named as <LastName>_<FirstName>_HW<x>-PC.tar. For example, if you are John Doe then the tar file should be named John-Doe-HW2-PC.tar.

8 Version Change History

This section will reflect the change history for the assignment. It will list the version number, the date it was released, and the changes that were made to the preceding version. Changes to the first public release are made to clarify the assignment; the spirit or the crux of the assignment will not change.

Version	Date	Comments
1.0	1/31/2018	First public release of the assignment.