Distributed, Fault-Tolerant, Serverless Computing for Java Programs

Semester Project for Introduction to Distributed Systems (COM 3800), Fall 2024

Document version: 1.2.1 – added updated PeerServerImpl constructor signature on page 12
version: 1.2 – added all due dates.
version 1.0.1 – added 4.a on page 5

Table of Contents

Overview of the Project

Serverless Computing is a model of cloud computing in which the user of the service does not manage the computer / virtual machine on which their code runs. Instead, the user simply uploads his code to the cloud service, which takes care of all the

setup and management necessary to run the code. Commercial examples of serverless include AWS Lambda, Microsoft Azure Functions, Google Cloud Functions and App Engine, and Cloudflare Workers.

In this project you will build, mostly from scratch, a service which provides serverless execution of Java code. You will start out with building the simplest form of distributed system – a single client talking to a single server – and work all the way up to a scalable, fault-tolerant cluster. It will involve both applying concepts we learn in class as well as independent learning (including trial and error) regarding how to do specific things in Java.

Working on a Multithreaded Program

Note that this project makes heavy use of threads. You will learn about threads in your Parallel Programming course. For information about debugging multithreaded programs, see the section on that topic later in this document.

Some Rules for This Project

1. **Collaboration on ideas only:** You are allowed to discuss concepts, approaches, algorithms, and the use of JDK classes/libraries with classmates, but you may not share any code or allow a classmate to look at your code.
2. **No Third-Party Code or Libraries:** You may not use any thirds party Java classes, libraries, or code. The only things you may use are the JDK, JUnit for testing, Maven for building, and whatever code I give you.
3. **JDK 17:** We are using JDK 17. If you have a different JDK version installed on your computer, please change it to 17.
4. **GitHub Repo Directory:** all stages must be submitted on the private GitHub repo we created for you, which you have been using since Intro to CS.
    a. Create a directory called "com3800" (all lower case!) at the root of your GitHub repo.
    b. Each stage should be checked in under that directory as "stage1", "stage2", (all lower case!) etc.
    c. The directory structure under the directory of each stage MUST be that of a maven project – e.g.
    `com3800/stage1/pom.xml`
    `com3800/stage1/src/main/java/…`
    `com3800/stage1/src/test/java/…`
    d. Every file needed for a given stage must be in that stage's directory. **In other words, each stage's directory must have a complete maven project in it that is 100% independent of anything in any other stage's directory.** For example, a student named Mickey Mouse would have the following files in his GitHub repo at the end of the semester:
        i. `...\MickeyMouse\com3800\stage1\src\main\java\edu\yu\cs\com3800\JavaRunner.java`
        ii. `...\MickeyMouse\com3800\stage2\src\main\java\edu\yu\cs\com3800\JavaRunner.java`
        iii. `...\MickeyMouse\com3800\stage3\src\main\java\edu\yu\cs\com3800\JavaRunner.java`
        iv. `...\MickeyMouse\com3800\stage4\src\main\java\edu\yu\cs\com3800\JavaRunner.java`
        v. `...\MickeyMouse\com3800\stage5\src\main\java\edu\yu\cs\com3800\JavaRunner.java`
5. **Capitalization:** The capitalization (lowercase/uppercase) of all java packages, class names, directories, etc., must be exactly what I write in this document. You are too far along in the major to get package or class names wrong!
6. **JUnit Testing:** you must include **JUnit 5** tests, the class of which is named after the project stage, in a package named after the stage, for every stage of the project. This unit test must demonstrate all aspects of that stage working. **You must write all tests yourself – you may not share tests with another student.**
    a. For example, in stage #1 you must commit a class called edu.yu.cs.com3800.stage1.Stage1Test which is saved in your repo as
    `com3800/stage1/src/test/java/edu/yu/cs/com3800/stage1/Stage1Test.java`.
    b. It must be a JUnit test, **not** a "plain old" Java class with a main method.
    c. Your JUnit test class should not depend on any other classes other than those that comprise your project code for that stage, and the **Junit 5** library.

7. **Maven build and test:** Your project must have a valid and complete pom.xml, such that I can build and run your unit tests by simply typing `mvn test` at the command line. **Make sure you specify JDK 17 in your pom.xml!**

Further Information About Java & Tools

Oracle, the company that maintains Java, has some wonderful documentation for all the various aspects of Java – the language, the JVM, etc. Here are some you should be familiar with:

1. The Java 17 Javadocs. The part relevant to you most of the time is java.base, although other modules to come into play in this project (e.g. jdk.httpserver)
2. JDK 17 Documentation – lots of different things available there, some better than others
3. The Java tutorial – written for Java version 8, but still very useful for many things
4. If you want to devote more time: Professional-grade tools

As a side point, there are four different ways of talking about Java that you should be familiar with (if you aren't already):

1. Java Virtual Machine (JVM) – the runtime environment that takes java bytecodes (.class files), translates them into machine code on the current machine, and executes them. There are many languages that can run on the JVM in addition to Java.
2. Java Runtime Environment (JRE) – the program and libraries needed to run a Java program on the JVM. Does not include javac (the Java compiler) or other command line development tools.
3. Java Development Kit (JDK) – includes the JRE as well as the command line development tools needed to write, compile, and package Java programs.
4. Java Language and Virtual Machine Specifications – for each version of Java, these documents give you rigorous details regarding the syntax and semantics of the Java language, as well as the precise required behavior of any JVM implementation.

Lastly, you might want to check out VisualVM – the JVM monitoring tool that, as far as I have seen, is the easiest to jump into quickly.

Following Instructions, Academic Integrity

Each stage has specific instructions below. Failure to follow any instructions below, or rules above, can result in a zero on that stage, so read carefully and ask any/all questions on Piazza!

I will be checking all submissions for academic integrity issues using Stanford MOSS (and possibly some other ways as well.) If two students are found to have the same code, it is irrelevant who gave code to who – they are both in violation.

Stage 1: A First Experience with Clients and Servers. Due: Sun., Sept. 15, 2024, 11:59pm

Goals

1) Write your first client and your first server. Client and server are the basic building blocks of any distributed system
2) Familiarize yourself with HTTP, which is the protocol used for most communication over the web
3) Create one of the building blocks for subsequent stages of the project

Overview

You have been given a class called JavaRunner, which takes a String of Java code (must be a complete / valid class) and compiles and runs it. You will write a client which submits (over an HTTP connection) such Strings to a server that you will write, and then

waits for the output to be sent back from the server. The server you write accepts requests from clients, compiles and runs the Java code sent by a client, and returns the output (or compilation errors) back to the client.

To understand how JavaRunner works, see the following:

- To understand how to call the Java compiler within your code, see the Javadocs for javax.tools.JavaCompiler, as well as code examples at programcreek.com.
- A nice post about Java Classloaders (Java classes must be loaded into the JVM before they can be run)
- In order to execute the compiled class, we use the Reflection features available in java.lang.Class

To learn more about writing HTTP clients and servers using the JDK, see the following:

- For the server side: httpserver package, HttpServer class, HttpContext class,
- For the client side: either HttpURLConnection or the java.net.http package. Code samples for both can be found here.

Server Requirements

How Servers Share Information About Their Health / Functioning

It is important to keep in mind that servers are generally not designed to run on your laptop with you watching; servers generally run on a server machine in a rack in a data center, with no humans watching, or interacting with, individual machines. Therefore, servers should **use logging, not messages printed to System.out**, to output any messages that may be of interest to the system administrator or developers. A professionally run system will be monitored (likely in the aggregate with other servers/machines.) Monitoring can be any combination of the server proactively sending system health information to a monitoring service, reactively replying to requests that ask for health information, and/or logging to a file or another output stream. For this semester, you should use logging as your mechanism for outputting any messages or health information about your server. To get started with logging in Java, see the following resources:

1) The Java Logging Overview and
2) The package overview in the Javadocs for the java.util.logging package
3) This StackOverflow post regarding how to log to a text file

Logging is something you should use in all servers you write for now on, not just on this project!

SimpleServerImpl

1) **Class name and interface to implement:** You will write a class called `edu.yu.cs.com3800.stage1.SimpleServerImpl` which implements `edu.yu.cs.com3800.SimpleServer` and uses the HTTP server package that is built in to the JDK.
2) **Main Method:** your class must include the main method copied below.
3) **Constructor:** `SimpleServerImpl` must have the constructor whose **signature** is commented out in the interface code
4) **Handler:** `SimpleServerImpl` must have one `HttpHandler` which responds to POST requests. This handler handles requests to the path "/compileandrun" and uses JavaRunner to compile and run Java source code. The handler must:
    a. Make sure the request method is POST, and if not reply with a response code of 405.
    b. make sure that the content type in the client request is "text/x-java-source" and return a response code of `400` if it is not.
5) **Requirements on Java classes to compile & run:** The Java code submitted by clients to the server must conform to the following rules:
    a. It must not rely on / import any class other than those built into the JRE
    b. It must have a constructor that takes no arguments
    c. It must have a public method whose name is `run`, takes no arguments, and whose return type is `java.lang.String`. In other words, it must have a method whose signature is `public String run()`
6) **You may not modify JavaRunner at all.**

7) **Server Response to Client:** after attempting to compile and run the Java code, the server should respond to the client as follows:
    a. If compile and run succeed, the response code must and be `200` and the response body must be whatever string was returned by the `run` method of the class that was executed
    b. If an exception is thrown when trying to compile or run the code, the response code must be `400`, and the response body must be comprised of the return value of `Exception.getMessage()` followed by a newline character ('\n') followed by the String contents of `PrintStream s` after calling `Exception.printStackTrace(PrintStream s)`.

Main Method for SimpleServerImpl:

```
public static void main(String[] args)
{
    int port = 9000;
    if(args.length >0)
    {
        port = Integer.parseInt(args[0]);
    }
    SimpleServer myserver = null;
    try
    {
        myserver = new SimpleServerImpl(port);
        myserver.start();
    }
    catch(Exception e)
    {
        System.err.println(e.getMessage());
        myserver.stop();
    }
}
```

Client Requirements

You will write a class called `edu.yu.cs.com3800.stage1.ClientImpl` which implements the **`edu.yu.cs.com3800.stage1.Client`** interface. For more info on writing HTTP clients in Java, see the overview.

Requirements for the client:

1. Your client must have the constructor whose signature is commented out in the interface code. The constructor arguments indicate the host and port of the server the client must submit its requests to.
2. When sendCompileAndRunRequest is called, the Client must send the Java source code string to the server. Calling getResponse() must then return its response as an instance of Client.Response
3. When your client sends a request to the server, it must set the content type of the HTTP request to "text/x-java-source"
4. Your stage1 Junit 5 test must print out both the expected response as well as the actual response exactly as shown below (including newlines), must test if they match, and the test must fail if they don't match.

```
Expected response:
[print expected response here]
Actual response:
[print actual response here]
```

Don't forget to write and include JUnit tests and pom.xml!

Stage 2: Distributed Leader Election Algorithm. Due: Tuesday, October 15, 11:59pm.

Goals

1) Implement your first distributed algorithm, and thereby gain experience and comfort in designing, reasoning about, and writing distributed code
2) Learn to use threads and queues, both of which are critical mechanisms for a server regardless of whether it's a standalone node or part of a horizontally scaled system
3) Build more depth of understanding of the standard internet protocols, and their appropriate use, by using UDP to pass messages between servers
4) Create one of the building blocks for subsequent stages of the project

Overview

As you know from class, leader election is one example of the need for consensus & coordination among nodes in a distributed system. In this stage of the project you will focus on implementing the leader election algorithm used by ZooKeeper in a cluster of servers that you will write. Each sever will have 3 threads running:

1) the main sever thread (via the PeerServerImpl class) which is focused on taking part in the election
2) a thread for sending messages (via the UDPMessageSender class)
3) a thread for receiving messages (via the UDPMessageReceiver class)

Requirements

1) **What to write:** you will write a class called `edu.yu.cs.com3800.stage2.PeerServerImpl` which implements **`edu.yu.cs.com3800.PeerServer`**. A number of instances of this class running simultaneously will form a cluster. Your "only" task in this stage is to implement the leader election algorithm, such that if a number of these servers are started, they will communicate with each other and elect a leader according to the leader election algorithm. You will see as you look at the classes provided to you that the server's code will be made up of a number of classes. See Code Being Given to You below for more details.
   Secondly, you must complete the LeaderElection class, which is where you implement the actual leader election algorithm.
2) **Threads:** as described above in the overview, each server should have 3 threads running. The two threads for UDP communication must be daemon threads.
3) **Algorithm tweaks:** We are modifying/simplifying the election algorithm in one important way: we are only considering the id of the servers (represented as a long), not the id of the latest transaction they have seen. Since we are not building a data store, we have no significant transactions or state to be shared across the servers. In addition, we are not using the "OBSERVING" server state yet. **You may not modify the algorithm in any other way.**
4) **UDP protocol:** the servers will communicate with each other using the UDP protocol. UDP is a "datagram" service, with no guaranteed delivery. What that means for us practically is that its messages that are sent over the network are both fewer in number and smaller in size than TCP (which, in turn, is much more efficient than HTTP.) Since any two servers in our system will only send each other a single message without any expectation of a reply, and if a couple of messages get lost here and there it will not impact our election in any significant way, there is no reason to pay the price of using TCP.
   To understand this point further, see the ZooKeeper source code, or, for a lighter perspective, see this presentation about Microservices at Google which discusses why not all servers/services should be microservices talking to each other using less efficient protocols etc.

Helpful Information

Threads and Network Code in Java

This stage involves writing multithreaded, networked code in Java. Useful resources:

1. Java Tutorial on Concurrency in general, and specifically on Threads:
   https://docs.oracle.com/javase/tutorial/essential/concurrency/

2. Fantastic book on Java Concurrency: http://jcip.net/
3. http://cs.baylor.edu/~donahoo/practical/JavaSockets2/textcode.html - great example source code of networked clients and servers in Java.

Queues

As you can see in the provided code, queues are you best friend when communicating between threads and/or processes and/or across the network – they allow you to get/put information without having to synchronously wait around for "the other guy" to be ready to put/get. (You may recall that we touched on this tangentially in Data Structures.) You should note that we are using instances of `java.util.concurrent.LinkedBlockingQueue`. Since these queues are being used by multiple threads, we must use thread-safe queues.

Code Being Given to You

You have been given code that specifies the classes, interfaces, and logic involved. Even though these classes are being provided to you, you must understand every line in them, both for the final exam as well as to understand the system well enough to succeed in coming project stages. If you try to "just get by" at this stage, you are setting yourself up for a lot of pain and trouble later.

Very Important Note: in the two UDP classes, as well as the "Peer Server Hints", you will see a pattern which is very typical in server code: the server runs in an infinite while loop which will only exit when some other thread has interrupted it. Make sure you understand how this pattern works, as you will need it in later stages.

1. LeaderElection.java – pseudocode for the election algorithm
2. PeerServer.java – the interface your server must implement
    a. Example methods – provide you some inkling of what certain parts of your server may look like. (Do not check this class in to your git!)
    b. Peer Server Implementation Hints – Some additional hints regarding what your peer server implementation will look like
3. ElectionNotification.java – describes the data that must be present in the message content of any message that is sent between servers as part of the leader election
4. Vote.java – represents a vote inside a single server that will not be sent to other servers
5. UDPMessageSender.java and UDPMessageReceiver.java – the classes you must use for sending/receiving messages via UDP between servers.
6. Message.java – messages sent between server must be sent as byte[] captured as the network payload produced by this class.
7. QuorumPeerServerDemo.java – a simplistic demo of what it looks like to create and start a number of peer servers. (Do not check this class in to your git!)

Don't forget to write and include JUnit tests and pom.xml!

Stage 3: Master-Worker. Due November 10, 11:59pm

Goals

1) Extend your knowledge into writing and debugging a distributed cluster which actually provides a service to clients
2) Further shift your software engineering thinking away from a single program/process to instead thinking about many processes which could be running anywhere on the network, and how they collaborate to get work done
3) Gain experience implementing a common distributed architecture: master-worker

Overview

Once you've made it this far, congratulations! You are now experienced in writing networked servers and distributed algorithms. In this stage you are going to combine stages 1 and 2, with some extensions, to create a cluster of PeerServers that provide a service which allows clients to submit Java source code for "serverless" execution. The cluster will elect a leader, and the elected leader will coordinate cluster activity in order to fulfill client requests.

Warning: Change in How We Think About Program Execution Ahead!

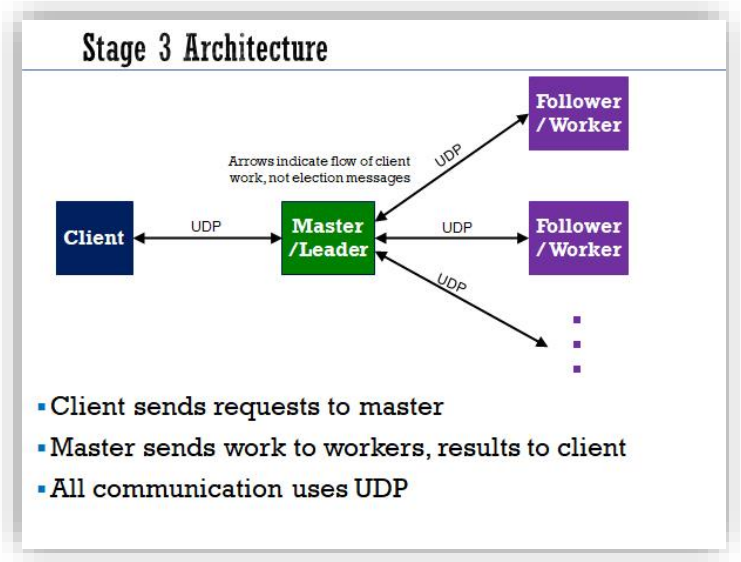
This stage is your first experience with writing and running an actual service that is comprised of multiple nodes/servers. Specifically, your cluster of nodes will use the Master-Worker architecture that we discussed in class. The server that wins the election to be the leader will function as the master, and the other servers will function as workers. The leader/master is the only server that:

- accepts requests from clients
- sends replies to clients
- assigns client requests to worker nodes that can do the work necessary to fulfill the requests.

While each cluster can have only one leader, it can have as many workers as the hardware on the machine(s) on which the cluster is running can handle. In other words, the code you write in this stage could be executed with one master and one or more workers that are running…

1. …all in the same JVM on the same machine as separate threads, or
2. …each in a different JVM on the same machine, or
3. …each on their own machine, or
4. …in any combination of the previous 3 choices

Keep in mind as you write your code that each server must communicate with other servers over network connections (not calling each other's Java methods directly), since you can't assume they will be running in the same JVM, or even on the same machine!



Requirements

Classes to Write

This stage revolves around three classes:

1. `edu.yu.cs.com3800.stage3.PeerServerImpl`, which picks up where your peer server implementation left off in stage 2 but now adds logic for the peer server being in the FOLLOWING (worker) and LEADING (master/leader) states.
    a. NOTE: the logic being added for FOLLOWING and LEADING at this point only address the master-worker distribution of work based on the results of the first election; it does not yet deal with triggering a new election when the leader dies, which we will deal with in stage 5.
2. `edu.yu.cs.com3800.stage3.JavaRunnerFollower`, which is run on a worker node. When the leader assigns this node some work to do, this class uses an instance of JavaRunner to do the work, and returns the results back to the leader.
3. `edu.yu.cs.com3800.stage3.RoundRobinLeader`, which is run on the leader node. It assigns work to followers on a round-robin basis, gets the results back from the followers, and sends the responses to the one who requested the work, i.e. the client.

While you are not required to write/submit a specific client class, some of your JUnit tests must, by definition, play the role of a client.

Threads Running on Each Server/Node

Each peer server is going to have a number of threads running at any given time:

1. The PeerServer thread, which focuses on the peer's state (LOOKING, FOLLOWING, LEADING) and creates and shuts down the other threads
2. A UDP send thread
3. A UDP receive thread
4. Either a JavaRunnerFollower OR a RoundRobinLeader OR neither, depending on if this peer server is in the FOLLOWING, LEADING, or LOOKING state.
    1. Note: your server **MUST NOT** run threads that aren't relevant to it. If it is NOT the leader, it MUST NOT run a RoundRobnLeader thread; if it is NOT a follower, it MUST NOT run a JavaRunnerFollower thread. If its role changes, threads no longer relevant to this server's role must be shut down.

We continue to use queues to communicate between the other threads and the UDP send/receive threads.

Round Robin Task Assignment, Asynchronous Results

The leader/master assigns tasks to the workers on a round-robin basis. What we mean by round-robin is simple: when a client request comes in, the master will simply assign the client request to the next worker in the list of workers. When the master gets to the end of the list of workers, it goes back to the beginning of the list and proceeds the same exact way. You should think of the list of workers as a circularly linked list which the master is looping through forever – the master assigns a request to the current worker and then advances to the next worker in the list.

The master thread MUST NOT synchronously wait (a.k.a. block) for a worker to complete a task; all work in the cluster is done asynchronously (otherwise, we would gain nothing by having multiple servers.) The master sends work to a worker via a UDP message and receives results via a UDP message; there is no synchronous connection or blocking.

Implication of Moving Work Around the Cluster: Request IDs

Each client request must be assigned a request ID by the master so that as requests / results are passed between master and worker we have a way of knowing what request a given message is relevant to.

Sample Code

Since at this point you are all already experts in writing multithreaded servers I will not insult you by giving you extensive sample/skeleton code, but here is a simple sample program that starts up a bunch of stage 3 PeerServerImpl servers, prints out who the elected leader is (this election is "rigged", since we use port numbers as server IDs we know exactly which server will

win, so we therefore know which port to send requests to as the client in this demo), sends the master some work as a client, and prints out the results received back from the master.

Don't forget to write and include JUnit tests and pom.xml!

Stage 4: Architectural Completeness. Due November 26, 11:59pm

Goals

The first 3 stages of the project were simplistic along multiple dimensions, compared to a real-world system. In this stage and the next we will add sophistication to your thinking and implementation skills (a.k.a. "get more real") in terms of architectural completeness and fault tolerance.

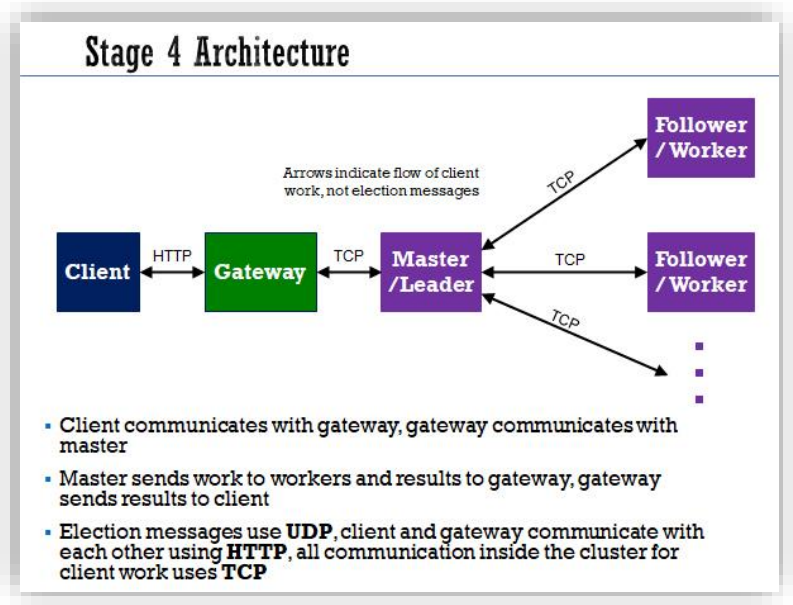
In this stage you will practice building a more realistic system by:

1) Adding an HTTP-accessible endpoint and API, thus realistically differentiating between the public face of a service and its internal implementation.
2) Caching replies at the HTTP-accessible endpoint and replying therefrom when there is a cache hit, as is done in internet-scale systems to avoid unnecessary traffic to the backend.
3) Differentiating between internal system messages and client requests by using the internet protocols that are most appropriate (and conventional) for each type of communication.
4) Recording behavior of various parts of the system in log files, which is critical for debugging as well as system administration (e.g. post-mortem analysis of system failures, traffic analysis for capacity planning, etc.)

Overview

Add a Gateway to Complete the Cluster Architecture

If you didn't have a home (or a dorm room), and at any given moment you could be anywhere, how could Amazon, UPS, or FexEx ever deliver a package to you? There must be some unchanging location (i.e. an address) where packages can be sent to you. The same is true of a service available over the internet – regardless of what architecture is used "under the hood" when building it, there must be some fixed single point to which clients can always send requests to access the service. That single point can't be whatever machine is currently elected as leader, since that machine may fail and a new leader will be elected, at which point all clients would have the "wrong address". To solve this problem you will add a new node called a Gateway (see, for example, Amazon API Gateway), which provides the public endpoint and API of our cluster and serves as the middleman between all clients and whichever node is currently the leader. (The details of how to architect a reliable public gateway is beyond the scope of this course, but if you're interested you can start learning about it by reading up on load balancing.)



Requirements

Required Classes

All the classes you are updating (`PeerServerImpl`, `RoundRobinLeader`, `JavaRunnerFollower`) and newly creating (`GatewayPeerServerImpl`, `GatewayServer`, and any others you see fit to create for this stage) must be in the `edu.yu.cs.com3800.stage4` package.

Gateway

The Gateway must be implemented in a class called **`GatewayServer`** which…

- …creates an HTTPServer on whatever port number is passed to it in its constructor
- …keeps track of what node is currently the leader and sends it all client requests over a TCP connection
- …is a peer in the cluster, but as an OBSERVER only – it does not get a vote in leader elections, rather it merely observes and watches for a winner so it knows who the leader is to which it should send client requests.
    - An OBSERVER must never change its state to any server state other than OBSERVER
    - Other nodes must not count an OBSERVER when determining how many votes are needed for a quorum, and must not count any votes sent by an OBSERVER when determining if there is a quorum voting for a given server
    - Think though carefully what changes this will require to `LeaderElection`!
- …will have a number of threads running:
    - an `HttpServer` to accept client requests
    - a **`GatewayPeerServerImpl`**, which is a subclass of `PeerServerImpl` which can only be an OBSERVER
    - for every client connection, the `HttpServer` creates and runs an `HttpHandler` (in a thread in a thread pool) which will, in turn, **synchronously** communicate with the master/leader over TCP to submit the client request and get a response that it will then return to the client.
        - Be careful to not have any instance variables in your `HttpHandler` – its methods must be thread safe! Only use local variables in its methods.

Because the peer servers now must know how many of their peers are observers, and as such don't count towards the quorum, and it also must know the gateway ID, the constructor signature of `PeerServerImpl` **must** be as follows for stages 4 and 5:

```
public PeerServerImpl(int udpPort, long peerEpoch, Long serverID, Map<Long, InetSocketAddress>
peerIDtoAddress, Long gatewayID, int numberOfObservers) throws IOException
```

Caching at the Gateway

It is standard practice in internet-scale systems to cache past replies as close to the client as possible, so that if the same exact request is made the reply can be sent immediately from the cache (to avoid burdening the backend with unnecessary work which would just make the entire system more loaded). Next semester, we will look in depth at an important example of this from Facebook's infrastructure.

To achieve this caching behavior, your `GatewayServer` must:

- Maintain a map of request body hash codes to responses. The map must be accessible by all your instances of `HttpHandler` and thread safe.
- Whenever a new request comes in, check if its hash code is already a key in the map.
    - If so (i.e. you have a cache "hit"), send cached response from the map, and don't bother sending the request to the leader.
    - If the hash code is not a key in the map (i.e. you have a cache "miss"), this exact request hasn't been seen before, and therefore the request must be sent on to the leader. Once you have the response from the leader, add a new entry to the map/cache so that the next time the same request is sent, it is found in the cache.
- In **every** response sent by the `GatewayServer` from now on, the http response must include an additional http header whose name is `Cached-Response` and whose value is `true` if the response was found in the cache/map in the `GatewayServer`, and `false` if it wasn't.

Protocols: HTTP, UDP, TCP

Until now we've been exclusively using the UDP protocol to communicate between nodes. This is entirely appropriate for leader election, since losing a message here and there is unlikely to affect the election outcome in any important way. However, it would be very bad if we would simply lose a message that has to do with a client request, so UDP is not appropriate for such communication. In addition, when it comes to communication between clients/users and servers/services over the internet, HTTP is the most commonly used (and the most appropriate) protocol. Therefore:

- **UDP** must continue to be used for all messages between servers that have to do with leader election
- **TCP** must be used for all messages between servers that have to do with client requests
    - keep in mind that unlike UDP, in which you send an asynchronous message and don't establish any sort of two-sided channel or connection, TCP involves a client Socket connecting to a ServerSocket, and synchronously waiting for a response over that connection. This doesn't make things much more complicated for you in your GatewayServer since the HttpServer uses a thread for each client request anyway, as mentioned above, but this will significantly change how your RoundRobinLeader works
- **HTTP** must be used for all communication between clients and the gateway

In each peer server, the port passed to the constructor will continue to be used for UDP, and you will calculate the TCP port of any server by adding 2 to the UDP port.

It is critically important that you understand that there are three entirely separate sets of communication going on in stages 4 and 5 – UDP for cluster management, TCP within the cluster for fulfilling client requests, and HTTP between clients and the gateway; each of those three are totally disjoint from each other.

You have experience writing an HTTP client and server from stage 1, and you have used UDP in stages 2 and 3. For examples of how to write TCP clients and servers, I recommend you see the Java Tutorial on Sockets and/or TCPEchoClient.java and TCPEchoServer.java here.

Threads in the Master

One advantage of asynchronous communication via UDP is that one thread (e.g. UDPMessageSender) can be responsible for sending out many messages to many servers for multiple purposes, as we did in stage 3. However, because in this stage the

Gateway will connect to the master via TCP, and the RoundRobinLeader now must connect to worker nodes over TCP, your master node will have a number of threads running:

- The thread in which PeerServerImpl itself is running
- The two (send, receive) UDP threads for leader elections.
- The RoundRobinLeader
- A thread that uses ServerSocket.accept() to accept TCP connections from the GatewayServer, reads a Message from the connection's InputStream, and hands that message to the RoundRobinLeader to deal with. You could do this within the RoundRobinLeader itself, or you could create a separate Thread whose only job is to do this. (I personally took the later approach, creating a class called TCPServer and having it give the Messages to the RoundRobinLeader via a queue, but you are not required to do it this way.)
- Within the RoundRobinLeader, each client request must be fulfilled by connecting, synchronously over TCP, to a worker node, sending the worker the request, and waiting for its reply. Because it is totally unacceptable for our cluster to only handle one request at a time (no point in having a cluster if you do one request at a time!), therefore for each client request the RoundRobinLeader must use a separate thread that will communicate synchronously with the worker and then send the worker's response back to the gateway. I strongly recommend that you use a thread pool to run and manage these threads, with the size of the pool being some very small multiple of the number of cores you have in your machine. Otherwise, you could overwhelm your machine with too many threads and it will grind to a snail's pace. (This is also useful in avoiding one type/aspect of a DOS attack by a malicious client.)

Logging

You must create a separate Logger instance for every significant thread in your cluster, and it must log to a file via a FileHandler. Note that I did NOT say a Logger for every class, rather a logger for every thread; if you have 5 instances of JavaRunnerFollower running, each one should have its own Logger that logs to its own file. The file name should clearly identify what this log file is a log from, e.g. "`edu.yu.cs.com3800.stage4.JavaRunnerFollower-on-`*`insertServerIDHere`*`-on-tcpPort-8022-Log.txt`". **You must have separate log files for:**

- The GatewayServer
- Every PeerServerImpl
- Every UDPMessageSender
- Every UDPMessageReceiver
- Every JavaRunnerFollower
- The RoundRobinLeader
- Every TCPServer, if you choose to take that approach to the RoundRobinLeader

Your logs should be informative but not saturated with information that is not useful after the servers have shut down. So, for example, it would be a good idea to log any key lifecycle events (such as the thread starting up and shutting down, making a connection to another server, etc.) as well as the unique/descriptive parts of requests the thread receives and the responses it sends. It would be a very bad idea to log every step in your logic.

Do not make me read through every line of all your different log files to know that your system works. Configure you logging / output such that I can read the basic output and see that it is generally working, and then I can dive into the log files if I want to see more details. Look into log levels and consider using multiple loggers, or some similar approach, to make this happen.

Important Miscellaneous Notes

Below is a list of miscellaneous important points that should prove helpful to you in this stage

- I've given you a class called `Util` which has methods for reading all the bytes from an input stream. I recommend you use it to save yourself certain headaches
- For our purposes, a TCP connection should only be closed by the client, never by the server. This ensures that the response from the server is fully read by the client before the connection is closed.
- Every thread you create other than the GatewayServer and instances of PeerServerImpl should be daemon threads, so if the servers shut down, all the threads shutdown
- You must have an easy and safe way to shut down your servers. To achieve this, you should call (either directly or inside a method you create on your servers) interrupt on each server, and every server's run method should be structured as below. Alternatively, you could use your own AtomicBoolean that you set in a shutdown method and check in your run method; this can be simpler to use in cases where setting interrupted will fail due to the thread being blocked on I/O.

```
@Override
public void run() {
    while (!this.isInterrupted()) {
       //server logic goes here
}
```

- Any data structures (maps, lists, queues, etc.) that will be accessed by multiple threads in a server should be instances of the Concurrent Collections found in java.util.concurrent
- Any individual values (e.g. a long) that will be accessed by multiple threads should be instances of the thread-safe classes in java.util.concurrent.atomic
- You will notice that the Message class that I gave you includes the sender and receiver host and port. Be sure to make use of these, since the port numbers in an actual connection may be ephemeral ports, which can be confusing.

Debugging Multithreaded Programs

Debugging multithreaded programs is notoriously difficult. Some hints about debugging:

- The most important thing to help you in this task is making wise use of the log files that I required you to write. I also strongly recommend that you make use of an IDE that has a debugger that handles multiple threads well, such as IntelliJ (which is free for students.)
- I strongly recommend you use VisualVM to see what threads are actually running in your system. It has integrations with popular IDEs.
- Name your threads (use Thread.setName) so that in any tools you are using, it is clear what each thread actually is and you don't see thread names that are meaningless to you
- while debugging / iterating / fixing your code, work with the smallest number of threads. For example, the gateway + the leader + 2 workers. When that works, add more workers and see if everything still works. If so, go up to a larger number of workers.
- Make sure you are logging useful messages to your log files in all your threads. You should use both the debugger and the log files when figuring out what's going on in your multithreaded programs.
- **Make wise use of logger levels:**
    - you should configure your logging in such a way that there is one place in your code where you set Logger.setLevel which will then set the level for ALL your loggers. This can be hard-coded or a command line parameter you pass in. Either way, you will want to let different levels of log messages through, depending on what you are working on / debugging
    - use different log levels for different levels of detail! For examples, messages that give information about whether your system is fundamentally working or not should be either WARNING or SEVERE.

Information that is much more narrow/detailed, e.g. the contents of every message you send over UDP, shouldn't be any greater than FINE.

- Be conscious of the following question: when your code hits a breakpoint in your debugger, do you want all the threads to be suspended, or just that one? Make sure to set up your debugger accordingly.
- Make sure to either delete your log files in between debug sessions, or have some systematic way to either move them or name the log files or directory differently each time you run/debug
- If you have multiple threads reading off the same queue for different purposes (e.g. you are running an election and gossiping (see stage 5) at the same time), make sure that each thread doesn't permanently remove from the queue a message that is not meant for it. So, for example, if you election thread gets a gossip message from the queue, it should put it back onto the queue so the gossiper can read it from the queue
- If there is a thread that you no longer need, make sure to null out all references it has to any shared data structures, in addition to interrupting/stopping it
- Multithreaded programs making heavy use of the resources on one system can be somewhat unpredictable without expert tuning. For example, in my final stage 5 implementation, sometimes (not always) one follower would fail to rejoin the cluster after the leader was killed and there was a new election. Real world production systems would be highly controlled and tuned by experts to reach a high level of predictability/reliability. If you get such an intermittent "strange behavior" you should definitely try to track down the cause, as you don't want it to work on your machine but not mine, but if after many attempts and many runs the system works albeit with some intermittent problem, that *might* be ok.
- After a couple days of debugging some stubborn bugs in my stage 5 implementation, I decided that I would (*bli neder*) give $54 to *tzedaka* as a "*korban todah*" if it worked; it did work. I am not delusional enough to think that I know what is happening in *shomayim* and whether the *zechus* of that *tzedaka* had anything to do with it working, but giving more *tzedaka* is never a bad thing!

Don't forget to write and include JUnit tests and pom.xml!

Stage 5: Fault Tolerance. Due December 29 11:59pm

Goal

Take your first concrete step into building **reliable** distributed systems by:

1) using Gossip-Style Heartbeats to detect node failures
2) triggering a new leader election if the leader node fails
3) taking steps to avoid losing a client request if a node fails

Overview

In the real world, machines break, and the larger your system is the more likely it is that something somewhere in your system will break. We will therefore implement **Gossip-Style Heartbeats to detect node failures**. If we detect that the leader failed, we will run a new election to elect a new leader. If either the leader or a worker node die, we will take steps to ensure that we don't lose any client requests.

Requirements

Heartbeats

Gossip-Style heartbeats must be used between all servers – gateway, leader, and followers. Details of this method of failure detection will be covered in class.

Once a node is marked as failed, no other node should ever accept any messages from it, regardless of the type of message (election, vote, work, etc.), or the communication protocol (TCP, UDP.) In other words, we are assuming **crash-stop** failures – a dead node remains dead for as long as the system is running. In addition to that, details below describe specific behavior by specific node types in our system upon the failure of a node.

It will take some trial and error to see what timings work on your machine for your gossip logic. Here are the values I worked with, in milliseconds:

```
static final int GOSSIP = 3000;
static final int FAIL = GOSSIP * 10;
static final int CLEANUP = FAIL * 2;
```

If a follower is found to have failed

1) Other followers must put that node in a list of failed machines and ignore all subsequent messages/votes they receive from it.
2) The Leader will:
    a. no longer assign work to the failed worker
    b. reassign any client request work it had given the dead node to a different node

If the leader is found to have failed

1) The **Gateway** will locally queue all new client requests until a new leader is elected. (Reminder: the Gateway only observes – it does not get a vote.) If the Gateway had already sent some work to the leader before the leader was marked as failed and the (now failed) leader sends a response to the gateway after it was marked failed, the gateway will NOT accept that response; it will ignore that response from the leader and queue up the request to send to the new leader for a response. It's dangerous to accept messages/replies from a node that was marked as failed.
2) **Followers**
    a. queue the results of any client work they have completed, until a new leader is elected
    b. change their sever state to LOOKING, increase their epoch by one, and run an election
    c. ignore all election notifications they receive that have an older epoch number
3) **Once a new leader is elected:**
    a. **The elected leader**:
        i. Gathers any work that it, or other nodes, completed as workers after the previous leader died. Eventually the gateway will send a request with the same request ID to the leader, and the leader will simply reply with the already-completed work instead of having a follower redo the work from scratch.
        ii. Goes into leading mode
    b. **Followers** go back to FOLLOWING mode. If they had any queued completed work, they null it out once they send it to the new leader upon request.
    c. The **Gateway** sends any queued client work (be it new requests, be it requests that the response came from the old leader after it was marked failed) to the new leader

Each Node/Server Decides Autonomously What the State is

1) There is no centralized detection of any node going down – each node comes to that realization on its own via the heartbeats.
2) A FOLLOWING or LEADING node will only switch to the LOOKING state and take part in an election if it sees that the leader has gone down
3) If a node is in the FOLLOWING or LEADING state (i.e. not currently taking part in an election) and receives a LOOKING ElectionNotification message, it must reply with an ElectionNotification indicating which node it thinks is the current leader

Logging Requirements

Do not make me read through all your different log files to know that your system works. Configure you logging / output such that I can read the basic output and see that it is generally working, and then I can dive into the log files if I want to see more details.

Failure Detection & Election Behavior: Logging

- **Summary log file:**
  - Whenever a node discovers that another node has failed via missing heartbeats, **log AND System.out.println "[insert this node's ID here]: no heartbeat from server [insert dead server ID here] - SERVER FAILED"**
  - Whenever a node **learns something new via a gossip message**, it should log **only exactly** what it learned and from whom it learned it. Example: **"[insert this node's ID here]: updated [serverID]'s heartbeat sequence to [sequence number] based on message from [source server ID] at node time [this node's clock time]"**
  - Whenever a node switches its state (e.g. due to leader failure), it should **log & System.out.println "[insert this node's ID here]: switching from [old state] to [new state]"**
- **Verbose log file:**
  - A node must have **a separate/second logger to which it logs every gossip message** it receives. Each log entry should include the machine it received the message from, the message contents, as well as the time it was received.
- **Http endpoints:**
  - Each node should provide two separate http endpoints, one to retrieve the summary log file, and one to retrieve the verbose log file.
- **Nodes should not System.out.println anything pertaining to failure detection other than what is specified here**

Other Important Points

Since in this stage the leader may die, it must be the Gateway that assigns request IDs to client requests

FYI, my final stage 5 version of code was 2,097 lines of Java code (not counting comments or tests.)

Don't forget to write and include JUnit tests and pom.xml! Your unit tests must demonstrate that all the requirements listed above have been met!

Stage 5 Demo Requirements

Demo Script Requirements

You must write a demo script showing off your stage 5 impl.

General points about the demo script:

- **There must be a bash script called demo5.sh in the root of your project.** Whether you accomplish everything listed below using bash commands, or whether your script uses some combination of maven, JUnit, and Java, doesn't matter to me. Just make sure I can run a script with that name in Ubuntu and it makes everything happen. I suggest you run your demo script on a Ubuntu virtual machine to make sure it doesn't work ONLY on your laptop with your directory structure, etc.
- All console output from your demo must go both to the screen as well as to a file called "output.log", which must be found in the same directory as the demo script itself.

- You may collaborate with **one** partner on the required JUnit tests and demo script for stage 5. You may NOT share the actual script or JUnit with anyone but your partner. If you do develop them with a partner, write the name of your partner in a comment at the top of the file.

What the script must do:

1. Build your code using mvn test, thus running your Junit tests
2. Create a cluster of 7 peer servers and one gateway, starting each in their own JVM
3. Wait until the election has completed before sending any requests to the Gateway. In order to do this, you must add another http based service to the Gateway which can be called to ask if it has a leader or not. If the Gateway has a leader, it should respond with the full list of nodes and their roles (follower vs leader). The script should print out the list of server IDs and their roles.
4. Once the gateway has a leader, send 9 client requests. The script should print out both the request and the response from the cluster. In other words, you wait to get all the responses and print them out. You can either write a client in java or use cURL.
5. kill -9 a follower JVM, printing out which one you are killing. Wait heartbeat interval * 10 time, and then retrieve and display the list of nodes from the Gateway. The dead node should not be on the list
6. kill -9 the leader JVM and then pause 1000 milliseconds. Send/display 9 more client requests to the gateway, in the background
7. Wait for the Gateway to have a new leader, and then print out the node ID of the leader. Print out the responses the client receives from the Gateway for the 9 requests sent in step 6. Do not proceed to step 8 until all 9 requests have received responses.
8. Send/display 1 more client request (in the foreground), print the response
9. List the paths to files containing the Gossip messages received by each node.
10. Shut down all the nodes