**Logo, company name

Description automatically generated**

Operating Systems Concepts

Assignment 3 – Design Programming Project

CSIS 3810

Ernesto Louis

12/11/2022

**Cache Communication**

**Methodology**

Conceptual Framework / Overview

This project aims to efficiently demonstrate the use of synchronization through various synchronization techniques using semaphores and threads. This project also extends the use of sockets to establish a client-server relationship in which requests will be sent and the tasks made on the cache are sent back informing the client of any updates/changes. This will be accomplished by allowing the communication amongst (10) threads to access a shared resource, that is, a cache with destination and bandwidth pairs.

Goals / Strategies

The goals are to establish a client that creates a connection over a TCP socket. The client will periodically make requests to get a destination and read its bandwidth and then return to its processing area waiting for when to make another request. Conversely, there needs to be a server that runs infinitely listening for client requests (the destinations to query). The server must also be multithreaded as it allows for multiple clients to connect and be able to access the cache. Another class then needs to be created; a client handler class which is a thread that is created when a client connects. This will act as the middleman or rather, the way in which the destination will be queried. It will take in the TCP socket so I can access the client’s input stream to retrieve the client’s request (the destination request sent). The client handler will also have an output stream which will contain the output message to be sent back to the client. It should consist of whether or not the destination was found, and any relevant information pertaining to the actions performed on the cache.

The semaphore needs to properly allow a thread to have access to the cache. This is so that deadlocks do not occur and any concurrent modification exceptions are not raised (as they did due to incorrectly acquiring and releasing the semaphores). Depending on whether a destination is found, the semaphore must be released if it is not found so that the thread can calculate the bandwidth and another thread can use it in the meantime. Once the bandwidth is calculated, the current thread must then acquire it again in order to update the cache. Subsequently, the cache must appropriately be updated depending on whether it is full or not. If is it the case that an address was entered into the cache while the current thread was calculating the bandwidth, only the bandwidth must be updated (and not need to insert the destination-bandwidth pair). Thus, needed are robust algorithms to ensure the efficiency, ease, and synchronization of the cache.

Importantly is the BandwidthCache class which will contain a map of all the current (and possible) entries that will be queried when a client requests a destination. It should leverage a HashMap in order to perform a quick look-up on the entries with a time complexity of O(1). The BandwidthCache class should also have important functions such as an “addToCache(destination, bandwidth)” function to add a destination not currently in the cache, and a “getCache()” function in order to access the cache within the class.

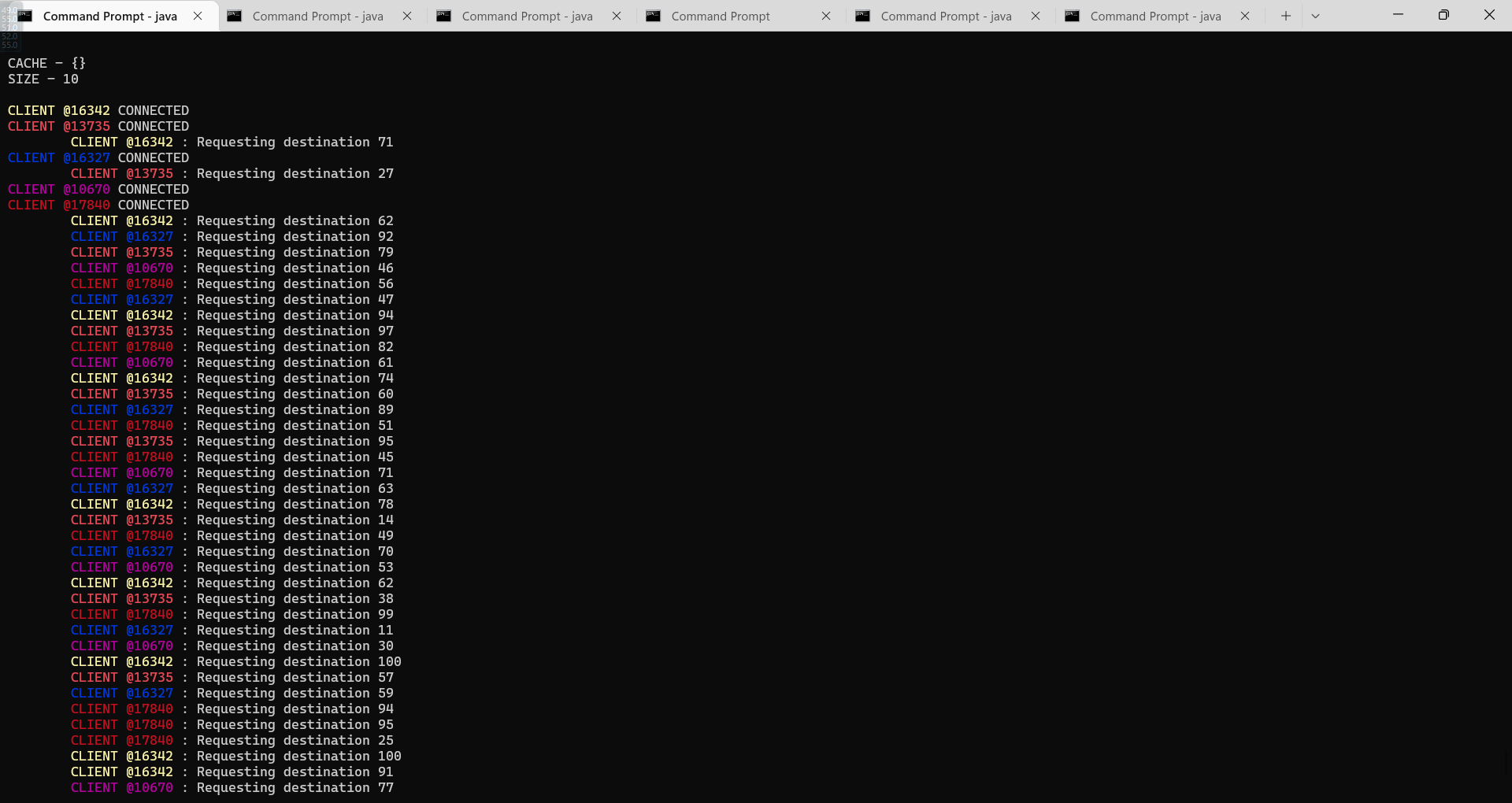
**Design**

Data Design

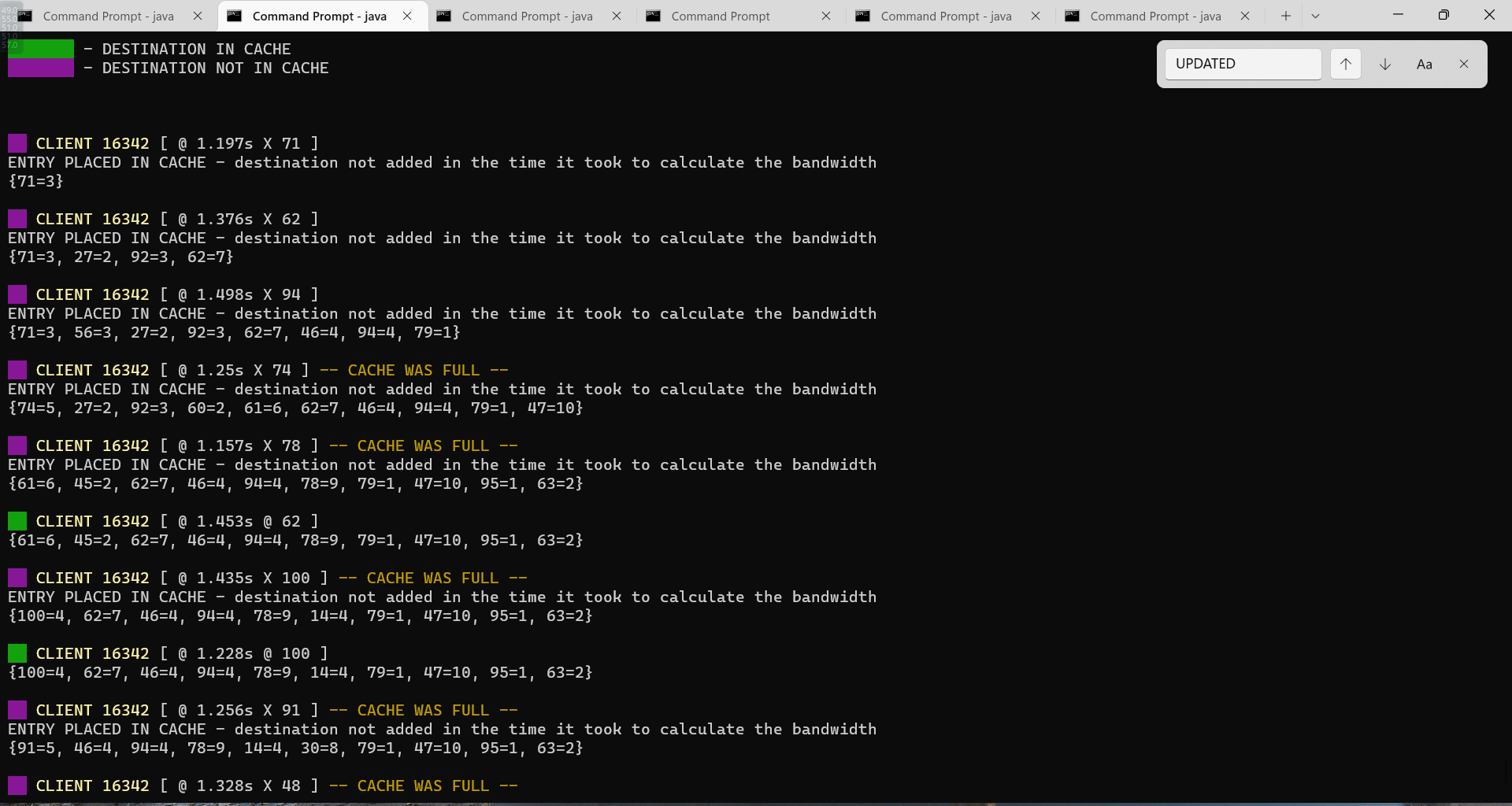
As a client connection is made, the server displays which client is currently connected to it. At this point, the client periodically waits before sending a request. As it gets a request, it sends it over to the server. The server displays which client is making the request, along with the destination it is requesting. The server creates a thread that will handle querying the cache and based on the actions taken on the cache, send over the message to the client so the client can display it. For example, if the destination was not found, the server would respond to the client with a proper message.

Below is a sample output:

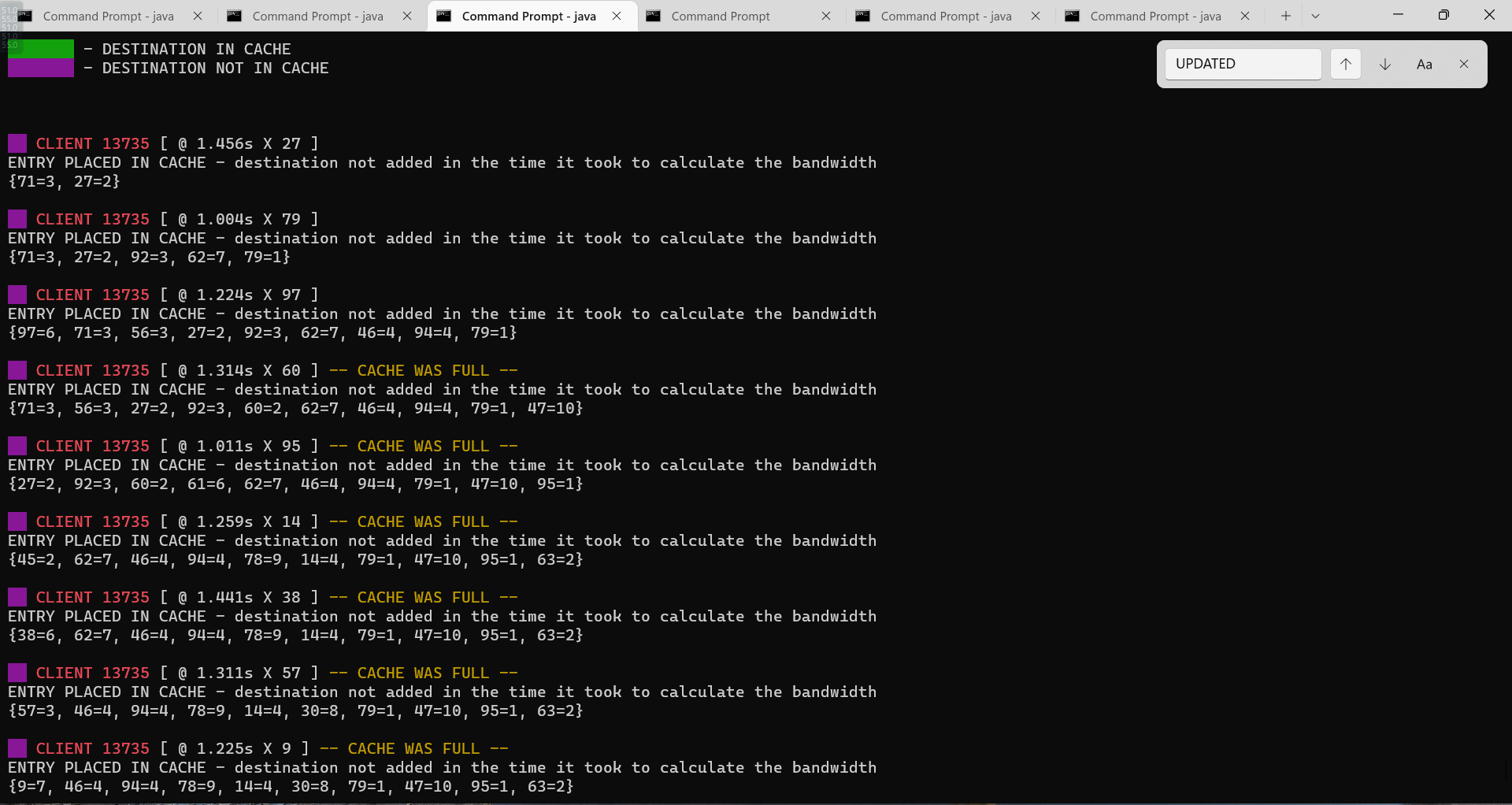
( server – with the 5 clients connected to it and making requests )



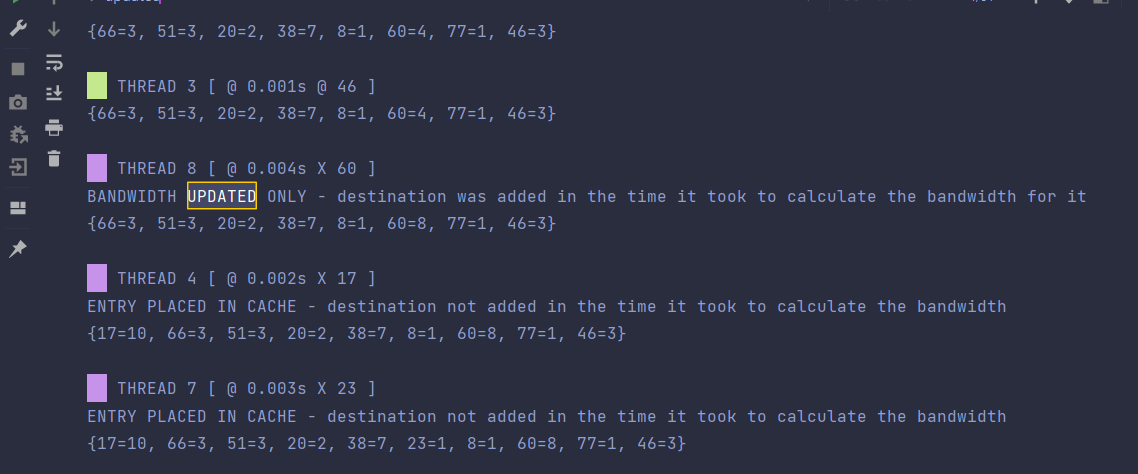
( client 16342 )



( client 13735 )



It is also the case that, once a semaphore is released to update an invalid destination, a thread that is now using it could have had inserted that destination while the previous thread was calculating the bandwidth. In that case, an output would be.



As shown in the output the THREAD 8 (requesting destination 60) displays “BANDWIDTH UPDATED ONLY - destination was added in the time it took to calculate the bandwidth for it”. Notice this is particular true because by THREAD 3, destination 60 with bandwidth 4 was already in the cache. Thus, it only updates the bandwidth to 8. In the case where the cache would be full, it would be more apparent as no replacement would occur.

Because this occurs not so often and the likelihood that would occur is slim, in order to reproduce that, the threads would have to make many requests. To be able to view this output, within the client file, on line 42 change the arguments of the random values for the variable “timeBeforeRequest” to

*int* timeBeforeRequest = random.nextInt(2000-1000) + 1500;

*int* timeBeforeRequest = random.nextInt(0);

This is because each request takes between 1-2.5 seconds, thus, the outputs are not shown as quickly in order to reproduce the output. A time request of 0 will be able to generate that output.

clientMessage.append("\nBANDWIDTH UPDATED ONLY - destination was added in the time it took to calculate the bandwidth for it ");

Data Structures and Algorithms Design

*Classes*

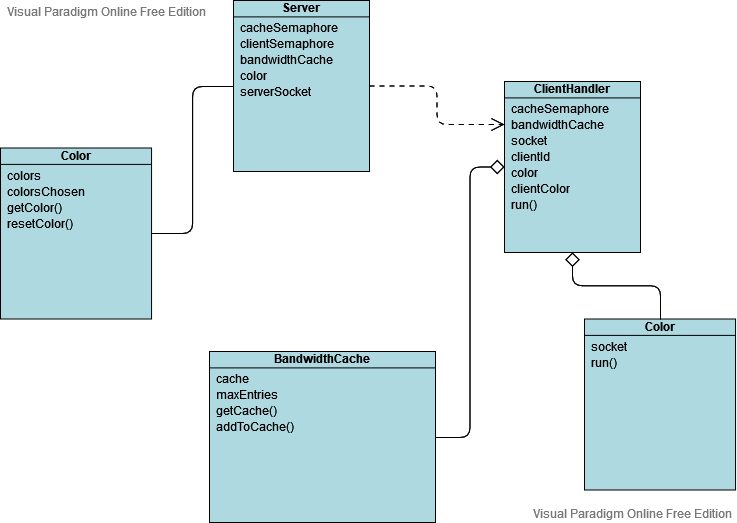
**Client*.*** This class (a thread) takes in a socket that will connect to a port in order to connect to the server. Within the thread-specific function, run(), it runs a loop that will periodically prompt that it is time to get a request and thus sends a request for a specific destination.

**Server.** This class is where the clients will connect; when they connect, the ClientHandler class will be instantiated which will manage the access to the bandwidth and be responsible for sending the message stream back to the client. It holds the attributes that are the semaphores (cacheSemaphore and clientSemaphore), it also holds the bandwidth cache object that will be composed in the ClientHandle class so that each client can access the cache. It also continuously listens to requests made by the clients via its ports continuously.

**Bandwidth.** Contains a HashMap of the cache and important functions such as “updateCache()” and “getCahe()”.

**ClientHandler*.*** This is a core dependent file to the server. With each request the Server class will establish the connection and then creates a thread that will provide access to the bandwidth cache using the shared BandwidthCache (resource) object that is stored in the server. It uses semaphores in order to lock access to the cache and/or releases them if the bandwidth for an invalid destination needs to be calculated and/or updated.

**Color***.* This class allows a random color to be associated with a client to help view the console outputs much easier. It consists of 15 unique colors that return a unique color each time the getColor() method is called.



*Data Structures*

**Hashmap.** A HashMap is leveraged for the cache in order to facilitate, essentially, a cache (a place of storage. It allows for quick access because the operation would generally always be O(1).

*Libraries*

**Semaphore.** From the concurrency library, the use of the semaphores allows us the synchronize the tasks and access the bandwidth by leveraging the acquire (lock the resource) and release() (unlock the resource for the next thread) methods.

**Socket.** Creates a TCP connection using a socket and a server socket in order to establish a client-server connection. Each socket has an input stream and outstream that is being written to and read from in order to exchange messages between the client and the server.

*Algorithms*

**Cache Access Through Semaphore Algorithm*.*** Once a client determines it is time to send a request, the running thread gets a permit and accesses the cache. Once the access is received, the class accesses the bandwidth and checks if the destination sent by the client is in the cache.

If it is, the thread must release the semaphore permit so it can calculate the bandwidth for that destination and simulate the ping. Because the semaphore is released at this point, it can be the case that the destination for which the thread is calculating the bandwidth could be inserted by another thread. If that is the case, only the bandwidth is updated. If not, both eh pairs (destination and bandwidth) are inserted into the cache. Once the bandwidth is calculated and the ping simulated, the thread attempts to access the semaphore again (because it was released to calculate the bandwidth) so that it can either update the bandwidth or place an entry entirely.

Conversely, if it is the case that the destination was in the cache, the thread accesses the bandwidth and then releases the semaphore and returns a message to the client where it waits for another destination request.

Cache Access Through Semaphore Pseudo Algorithm:

*Run(){  
  
 destination request sent by client  
   
 while( true ){  
   
 lock the cache  
   
 if( destination is in the cache ){  
   
 release the lock from the cache   
   
 calculate bandwidth and simulate ping  
   
 lock the cache  
   
 update bandwidth OR add an entry(depending on if a thread*

*already entered the destination while it calculated the bandwidth )   
   
 release the lock from the cache  
   
   
 }else if( destination is in the cache ){  
   
 release the lock from the cache  
   
 }  
   
 }  
  
}*

**Update Cache Algorithm*.*** This algorithm updates the cache with the requested destination and the calculated bandwidth.

* If it is the case that the destination was already updated before it could be added, then it only updates the bandwidth.
* If it is the case that the destination was not already added, first, check if the cache is full and needs to remove an entry, then add the entry value pair.

Update Cache Pseudo Algorithm:

*if( destination added by a previous thread ){  
  
 only update the bandwidth for that destination*

*}else{  
  
 if( cache full ){  
   
 remove the first entry  
 }  
   
 update cache with entry pair of destination and bandwidth  
  
}*

*Problem Analysis*

As the threads run simultaneously, depending on the task that makes the first request, the shared semaphore would call a method called acquire which then “broadcasts” (in a sense) to the other threads attempting to access the cache that it is being occupied. While this occurs the task stalls and keeps checking if the lock on the section has been released.

Through the use of semaphores and running threads simultaneously, I was able to achieve and design a program that would efficiently allow tasks to access resources. Achieving these have also allowed me to demonstrate my understanding of synchronization and optimizing ways to prevent deadlocks and indefinite postponement. Moreover, it allowed me to have created a client-server program simulating multiple clients attempting to access one resource and be able to do so efficiently using various synchronization techniques.

Architectural Design

*Initialization*

At startup, no other input is needed. Only the commands to compile and run the code

*Semaphores / Threads*

Leveraging the semaphore class and thread class, each vehicle class takes in the shared supply semaphore. From the car classes, threads are created and run for their respective tasks. The bandwidth cache semaphore controls any access to the resource that is the bandwidth cache. The bandwidth cache also makes sure that only one car is accessing it at a time. The shared semaphores only carry one permit such that only one task can access a critical section at a time as the threads run independently of each other. Another semaphore included is a counting semaphore which would track how many clients can establish a connection to the server. This semaphore would lock the server if the clients created exceeds the limit and an incoming client wants to connect to the server.

*Termination*

To successfully end the program, the user can press “ctrl + c” to terminate the programs (server or client) running via the console.

**Program Simulation/Output**

To run the program, open at least 2 command prompt windows within the directory of the “src” folder. After, compile both the client and server file by running the commands “javac Server.java” and “javac Client.java”. Once compiled, in separate windows, run the commands “java Server” and click enter and then “java Server” and click enter. It is important that the server is run first as it serves as the connection point for the clients. When compiled correctly and ran, the server and client(s) will be communicating back and forth sending requests and displaying the cached queries. More clients can be connected by simply opening up another terminal window and running the command “java Client”.