Assignment 2:

* Go to https://ipintelligence.neustar.biz/portal/
* Sign up for a free API key
* Write a java program that uses the geolocation API to output details
about the IP address (city, state, country, latitude, longitude, etc.) for the computer from which the program is running.
* Zip the project and send it to us (or put it on dropbox). We should be able to easily build and run it.
* OR: (Bonus points if) you host it in the cloud (Google App Engine/Micro Amazon instance) and we can run it online.

Implementation Notes

- App hosted on EC2 at http://54.201.102.202:8080/IpDetails/
- "… Output details about the IP address for the computer from which the program is running": I hosted the app on Amazon EC2, so I'm assuming this meant displaying the ip address of the client connecting to the app. Otherwise, the same ip info for the Amazon EC2 instance would be displayed all the time.

Caching

Since the ip info doesn't change very frequently, caching is appropriate here to reduce the Restful service calls. I've only implemented a simple FIFO cache as a prototype because the FIFO strategy would naturally reduce the cache expiration issue. Caching can be enable/disabled in the properties file. Here are some of the notable points:

1. From my machine, I tested using Jmeter to send 1000 requests within 10 secs. The benefit of caching in this scenario is exaggerated because the cache hit rate is 100% (all requests come from the same ip). Having some sort of cloud testing where the requests are sent from different ips will give a more accurate simulation of real load & benefits.
2. Caching improves the performance (as shown in the Jmeter graphs below). For the scenario above, the avg response time with caching is 523ms, and 795ms without caching. Considering point #1, this is not as significant as I expected. There are 2 reasons I can think of:
    - When using cache, adding and retrieving cached items are synchronized, so the execution is behaving as single-threaded in this cache-checking stage. When many requests are sent concurrently, this bottleneck reduces the benefits of caching.
    - The service call is somewhat quick (under 1sec). If the call was longer, the performance gained from caching would be higher.

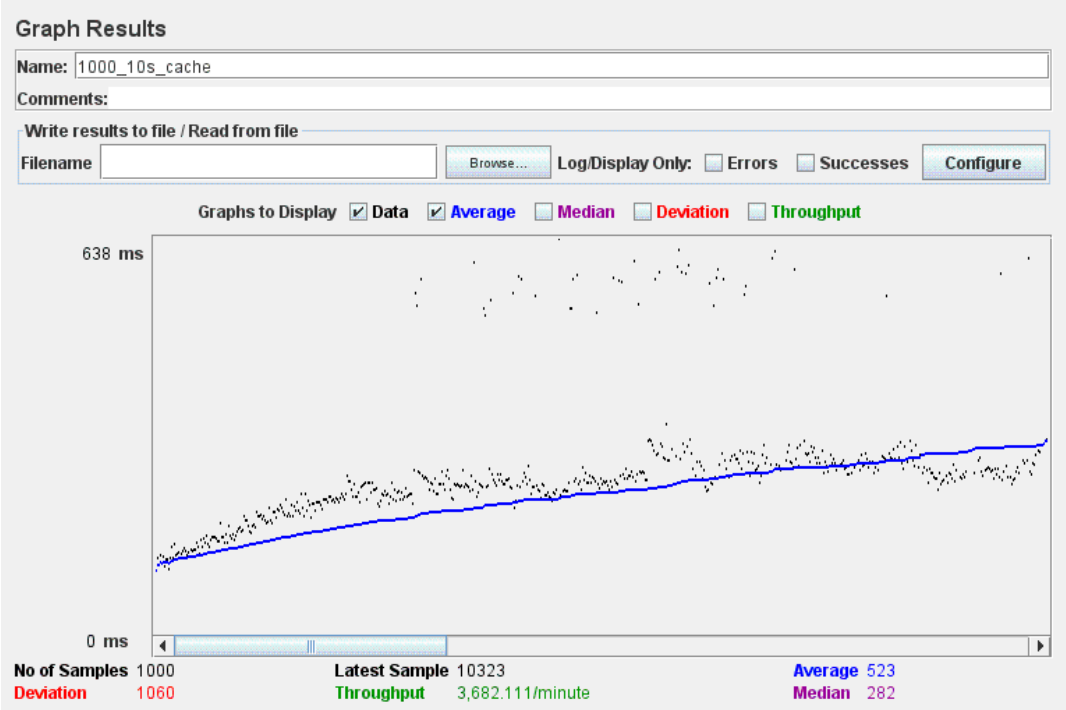

Figure 1: Caching enabled: 1000 requests in 10 sec

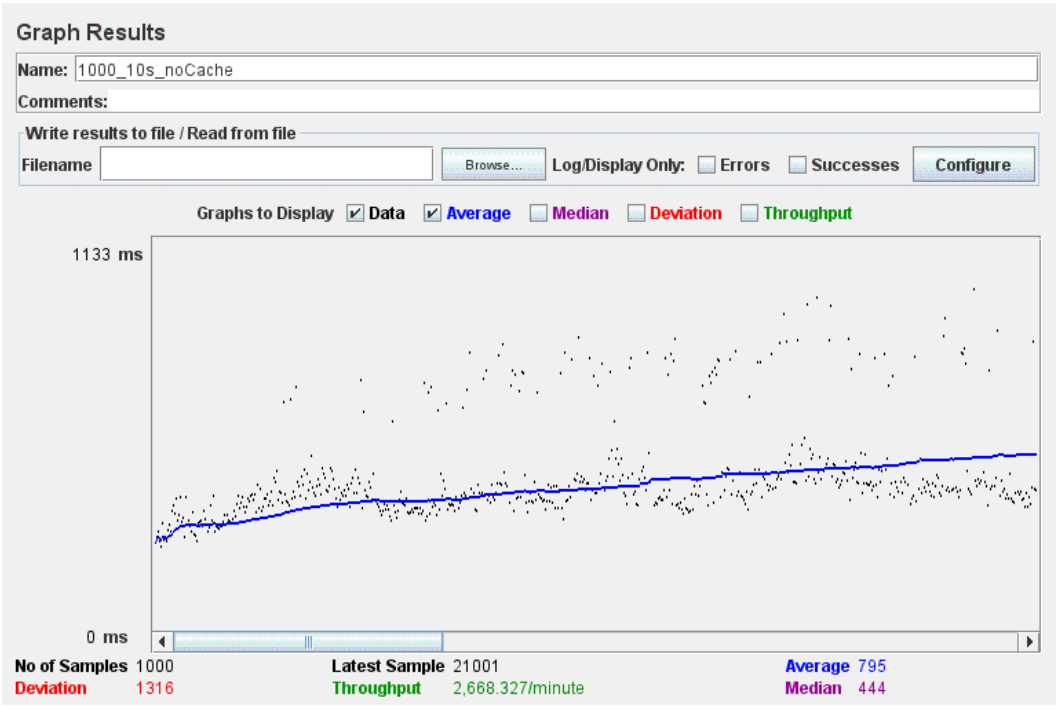

Figure 2: No caching: 1000 requests in 10 secs

Alternate Design Consideration

The current implementation just uses the default Tomcat threadpool size of 200 to handle concurrent requests. Instead of using this thread-per-request model, another approach would be to use the Async Servlet and Executor service to set up a linked blocking queue with a fixed

threadpool. Under this approach, each incoming request will be put on to the queue by the worker thread, to be processed by an async thread in the threadpool associated with the queue.

This approach is advantageous when there is a backend limit (low JDBC connection pool or low throughput web service). For example, if there's a limit of only 25 concurrent connections to the database, then we can use this approach and configure the Executor's threadpool to 25 and have the worker thread queue up the incoming requests. This way, the worker thread will be released early to process the next requests. If we were to use the default Tomcat configuration, a lot of worker threads will be used but they will idly wait for the available database connections, leading to waste of resources.

In this specific assignment, since there is only one backend task and there is no throughput limit, the implementation using Async Servlet would not provide any additional benefits.