<u>Load base Architecture and Design Decisions</u>

Order Service:

From A1, Order Service forwards all the requests(including the purchased endpoint) made to User and Product to the ICIS server. In order to make the process of forwarding requests fast, we implement concurrency using the Executor interface in java. Concurrency is implemented by creating 120 HTTP clients for both User and Product. Then, a round-robin approach is used in forwarding the HttpExchanges to the ICIS server by cyclically going through the 120 HTTP clients and sending the HTTP Requests using the executor. This allows for the requests to be sent concurrently from Order Service to the ICIS server, increasing the efficiency.

Prior to forwarding the request to the ICIS server, some error handling is done in the Order Service for both the User and Product commands so if a command is not possible and causes an error it does not need to be forwarded all the way to User and Product and an immediate response can be made.

The end point for place order was initially computed in the Order Service in our component 1. However, due to the caching system implemented in our Order and Product Servers, we have to forward requests to Order and Product servers to retrieve back data. The caching system will be described later.

ICIS Service:

ICIS Service forwards all requests from the Order Service to either the User or Product Server using the same concurrency and round-robin approach used by the Order Service.

User Service:

After initial profiling and testing, we were able to conclude that the most computationally expensive operations were database operations such as inserting, updating, and deleting elements from the database. Thus, in order to limit the number of database operations, we implemented an in-memory cache with a set max-cache size of 5000. The cache is a Java concurrent hash map which will store the User elements in a list with all the attributes. We use a list instead of an object to save memory. Instead of doing a database operation to insert, we insert to the cache first which greatly reduces the computation time. However, if the cache is full, then we do a bulk “insert or replace” into the database by executing all the queries as a batch. This allows us to clear our cache as we update the database based on our cache. We make sure to disable the auto commit mode on the database so that all the SQL statements can be grouped into one transaction. This makes the bulk insert extremely efficient.

However, prior to inserting a User, we need to check if the User is already in the database or the cache. Upon initializing the User Service, we create a Java HashSet to store all the IDs of the users that are currently in the database. This is not too memory intensive because we are using a HashSet instead of a HashMap. Thus, instead of making a database query to check if User is already in the database, we just check this HashSet. We also update the HashSet on every insert and delete.

While updating a user or deleting a user, we first check if the user is in the cache. If it is in the cache, then we update and delete the user accordingly. If the user is not in the cache, then we check the HashSet to see if the user exists in the database. If the user is in the HashSet, then we do a database query accordingly to update/delete the user. Otherwise, we return the appropriate error response.

Thus using a cache and set for IDs, we are able to significantly reduce database operations and increase our throughput.

Product Service:

The Product Service uses the same cache system and ID HashSet as User Service to reduce database operations and fulfill all the functionality needs.

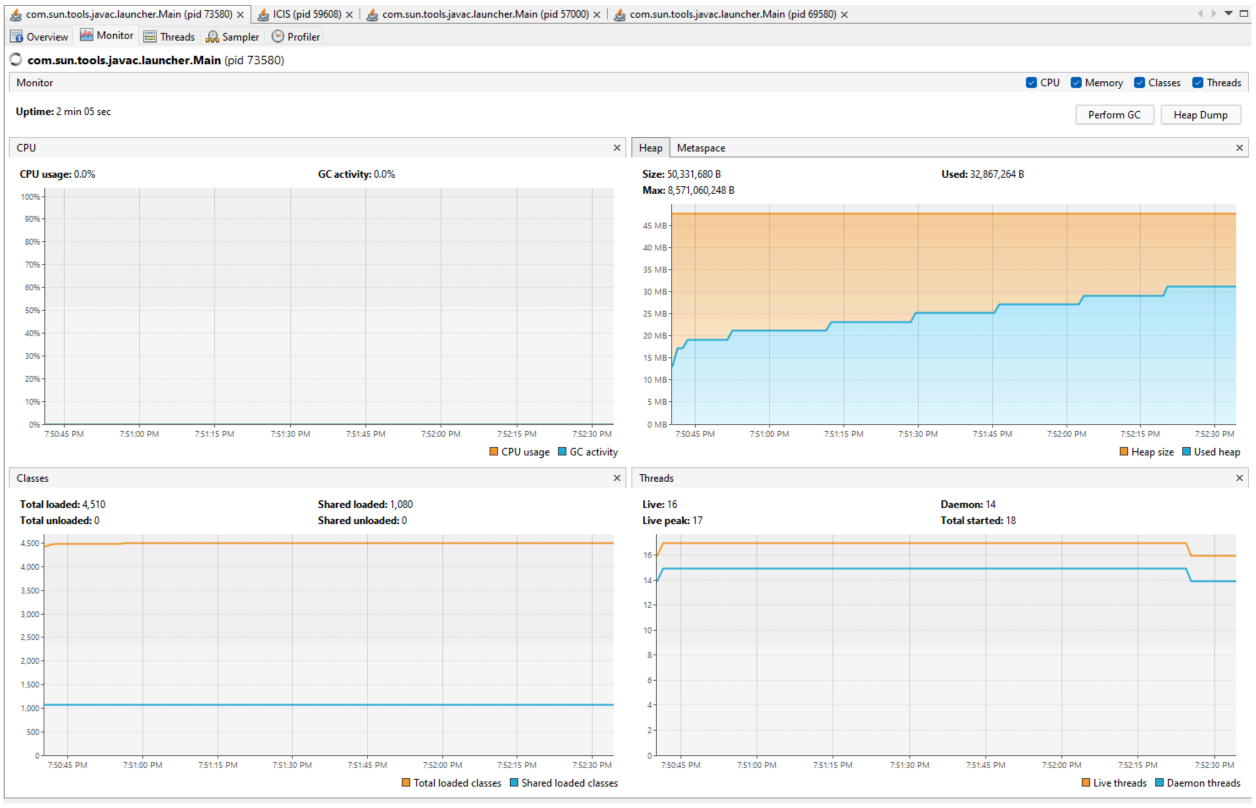
Database System:

We have separate databases for Product and User so that we can keep the services independent. We create our databases with SQLITE and use a SQLITE JDBC driver to connect to the database and do queries accordingly. We considered using postgreSQL as it allows for concurrency in the database operations. However, we had significantly reduced the database operations with our caching system, so we felt it was unnecessary to switch to postgreSQL.

<u>Profiling</u>
For testing we had multiple different testers created. We had a test checking for concurrency running 5000 requests with 100 threads again and again to test the consistency and to test concurrency of the servers. We had a separate test just for creating, getting, and deleting all in one to test how the servers would do after going through many different types of requests. All profiling shown below are profiled on our concurrency tests.

These graphs below for our first implementation before any optimizations and changes, before each profiling will be the changes made that increased its speed.

<u>Original before component 2 modifications</u>
Total Requests: 5000
Elapsed Time: 101.15 seconds
Requests Per Second: 49.43
Order Service



com.sun.tools.javac.launcher.Main (pid 73580)

Uptime: 2 min 05 sec

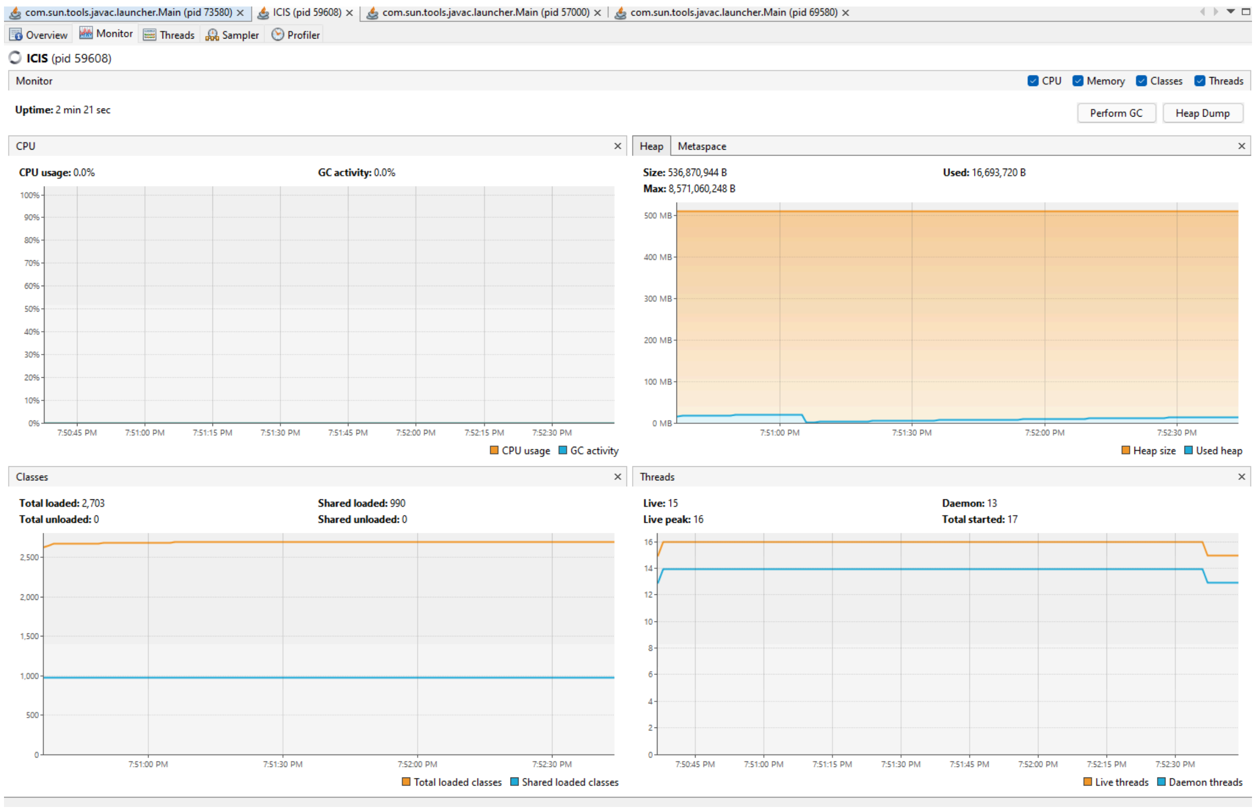
CPU usage: 0.0%
GC activity: 0.0%

Size: 50,331,680 B
Max: 8,571,060,248 B
Used: 32,867,264 B

Total loaded: 4,510
Total unloaded: 0
Shared loaded: 1,080
Shared unloaded: 0

Live: 16
Live peak: 17
Daemon: 14
Total started: 18

ICIS



ICIS (pid 59608)

Monitor

Uptime: 2 min 21 sec

CPU

CPU usage: 0.0% **GC activity:** 0.0%

Heap

Size: 536,870,944 B **Used:** 16,693,720 B
Max: 8,571,060,248 B

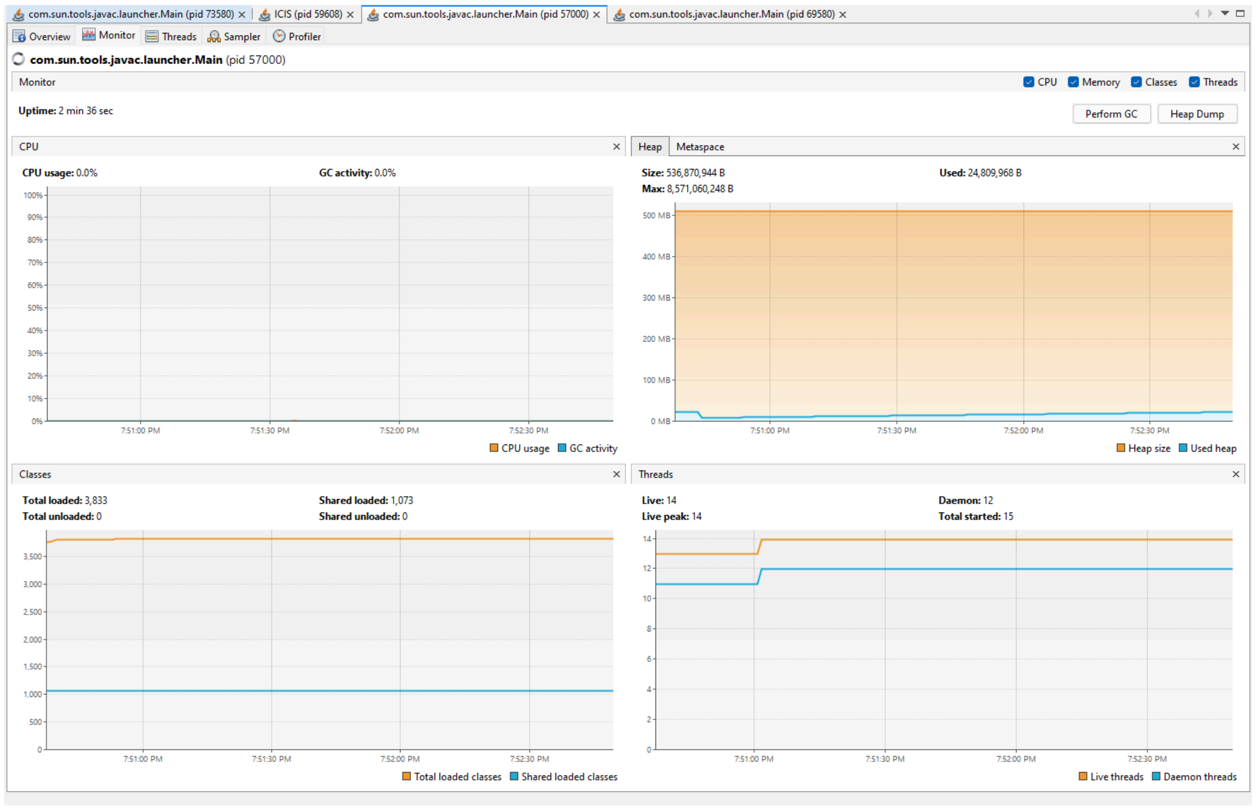
Classes

Total loaded: 2,703 **Shared loaded:** 990
Total unloaded: 0 **Shared unloaded:** 0

Threads

Live: 15 **Daemon:** 13
Live peak: 16 **Total started:** 17

User



com.sun.tools.javac.launcher.Main (pid 57000)

Monitor

Uptime: 2 min 36 sec

CPU

CPU usage: 0.0% **GC activity:** 0.0%

Heap

Size: 536,870,944 B **Used:** 24,809,968 B
Max: 8,571,060,248 B

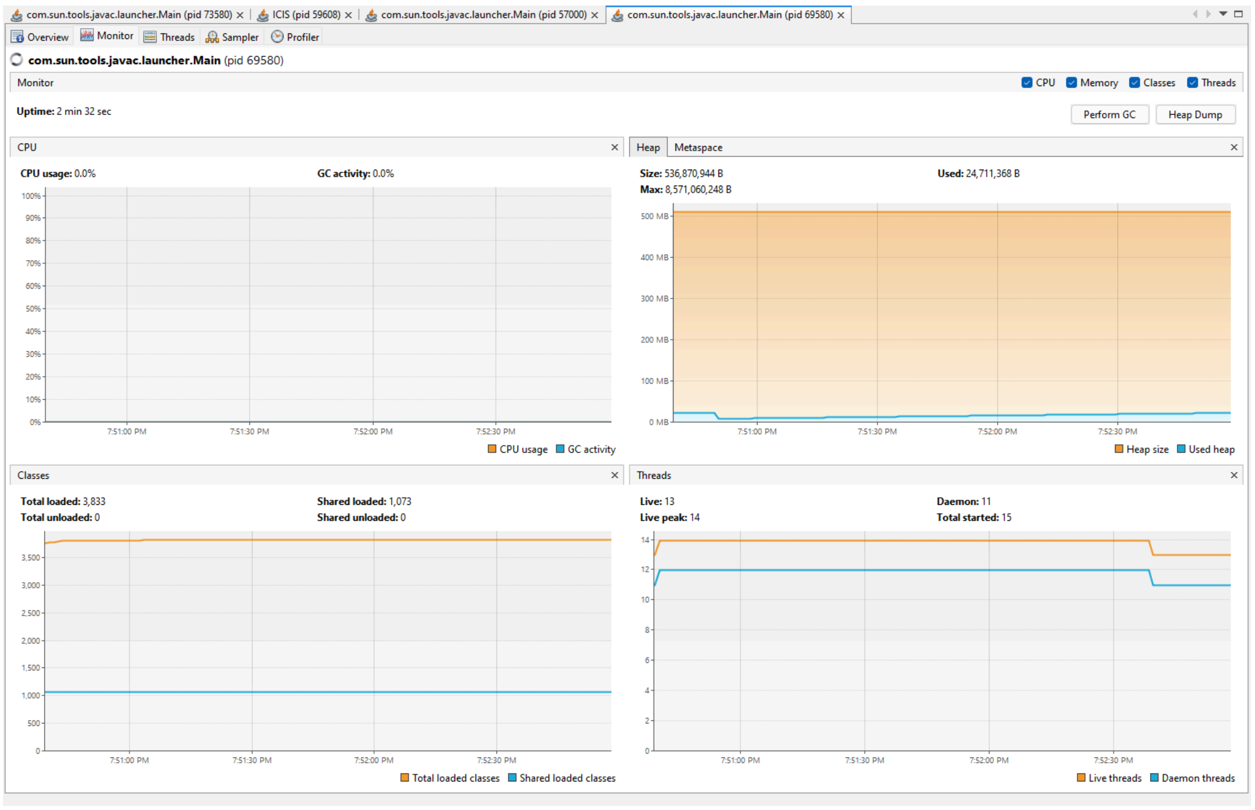
Classes

Total loaded: 3,833 **Shared loaded:** 1,073
Total unloaded: 0 **Shared unloaded:** 0

Threads

Live: 14 **Daemon:** 12
Live peak: 14 **Total started:** 15

Product



com.sun.tools.javac.launcher.Main (pid 69580)

Uptime: 2 min 32 sec

CPU usage: 0.0% **GC activity:** 0.0%

Size: 536,870,944 B **Used:** 24,711,368 B
Max: 8,571,060,248 B

Total loaded: 3,833 **Shared loaded:** 1,073
Total unloaded: 0 **Shared unloaded:** 0

Live: 13 **Daemon:** 11
Live peak: 14 **Total started:** 15

2nd profiling

With the first initial profiling of our code from component 1, we realized that the order service definitely needed to be multithreaded as it got more requests than it could send fast enough to the other services. This can be seen as the graph for order service has way more bytes used compared to the other graphs that get it in a slow and steady stream.

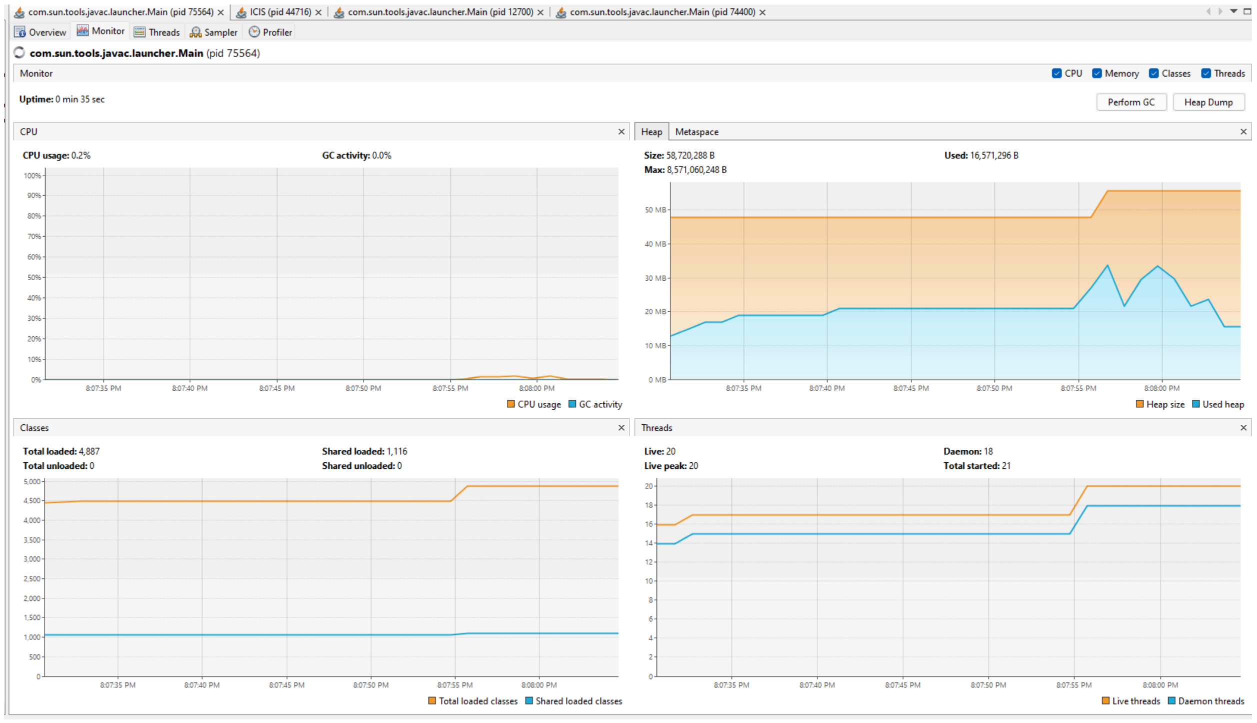
Total Requests: 5000

Elapsed Time: 8.79 seconds

Requests Per Second: 569.00

2nd push

Order



com.sun.tools.javac.launcher.Main (pid 75564)

Uptime: 0 min 35 sec

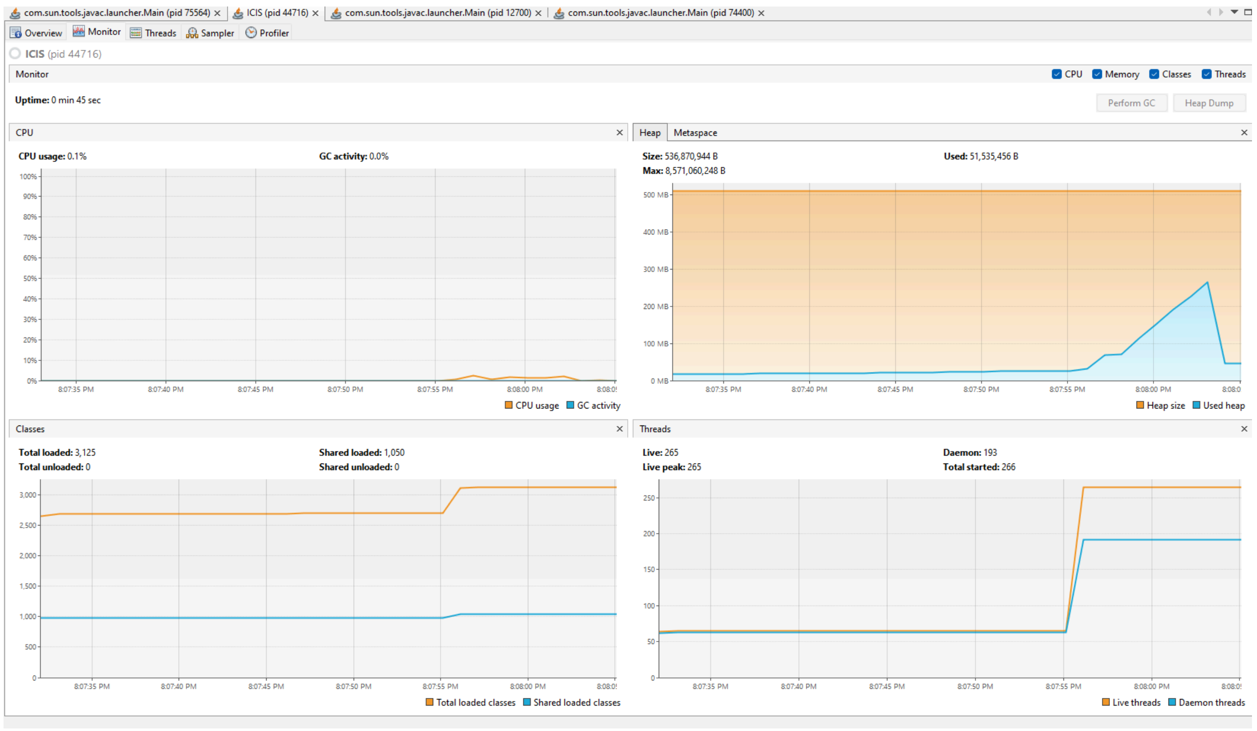
CPU usage: 0.2%
GC activity: 0.0%

Size: 58,720,288 B
Max: 8,571,060,248 B
Used: 16,571,296 B

Total loaded: 4,887
Total unloaded: 0
Shared loaded: 1,116
Shared unloaded: 0

Live: 20
Live peak: 20
Daemon: 18
Total started: 21

ICIS



ICIS (pid 44716)

Uptime: 0 min 45 sec

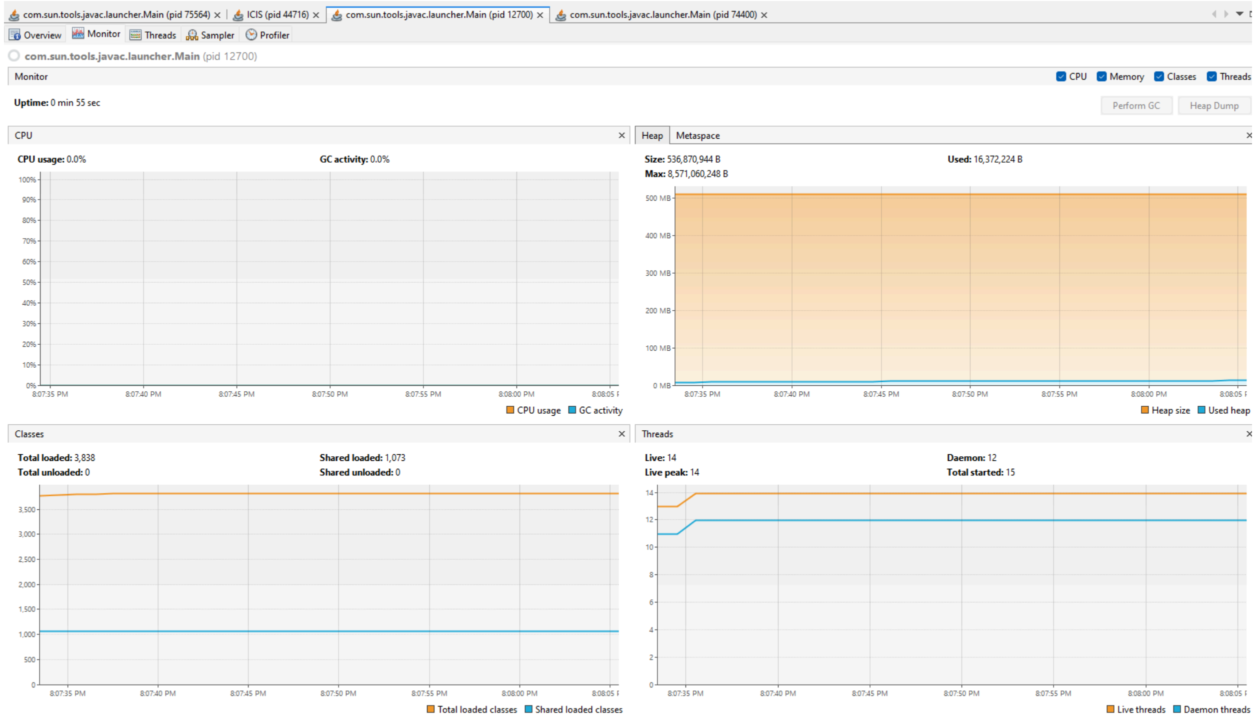
CPU usage: 0.1% GC activity: 0.0%

Size: 536,870,944 B Used: 51,535,456 B
Max: 8,571,060,248 B

Total loaded: 3,125 Shared loaded: 1,050
Total unloaded: 0 Shared unloaded: 0

Live: 265 Daemon: 193
Live peak: 265 Total started: 266

User



com.sun.tools.javac.launcher.Main (pid 12700)

Uptime: 0 min 55 sec

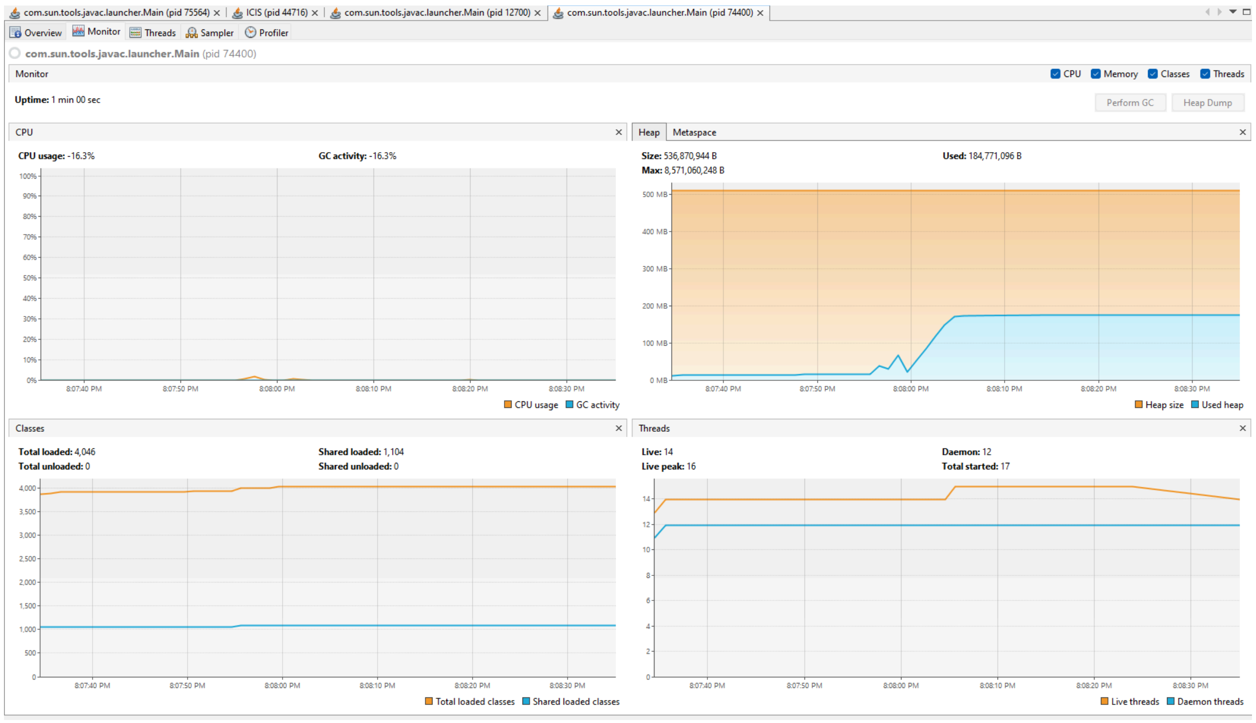
CPU usage: 0.0% GC activity: 0.0%

Size: 536,870,944 B Used: 16,372,224 B
Max: 8,571,060,248 B

Total loaded: 3,838 Shared loaded: 1,073
Total unloaded: 0 Shared unloaded: 0

Live: 14 Daemon: 12
Live peak: 14 Total started: 15

Product



com.sun.tools.javac.launcher.Main (pid 74400)

Uptime: 1 min 00 sec

CPU usage: -16.3%
GC activity: -16.3%

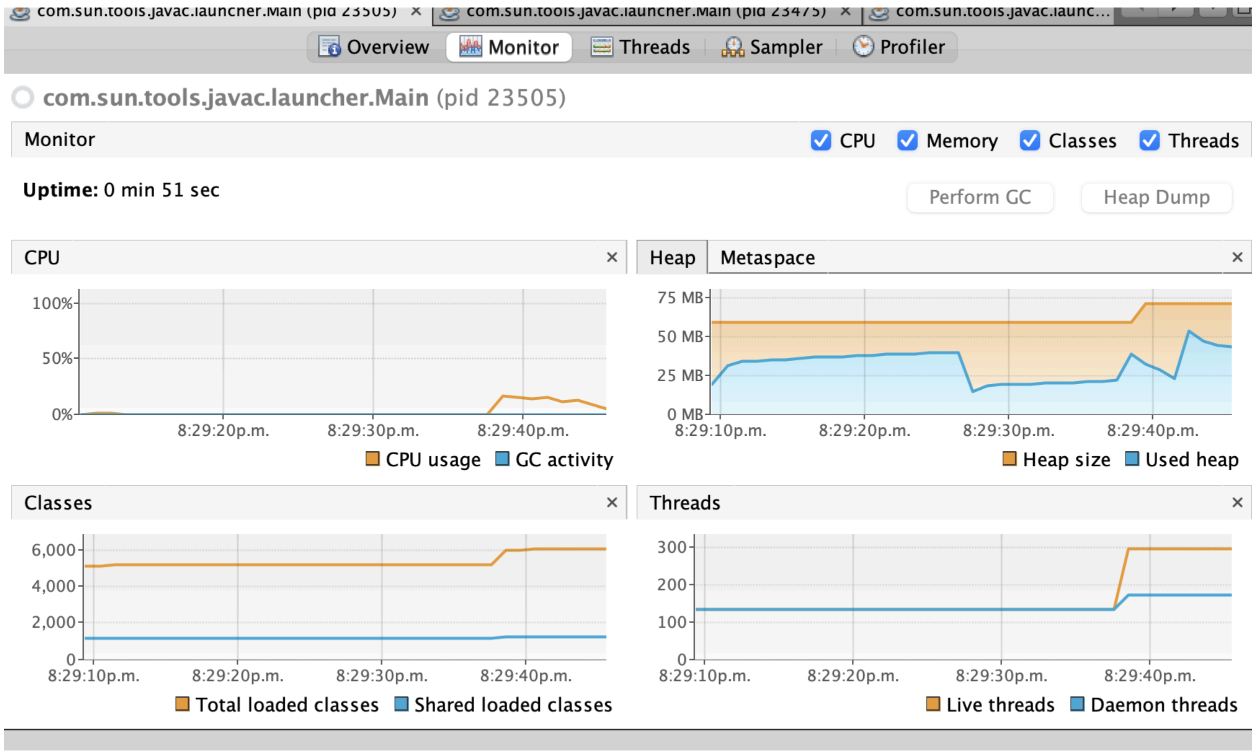
Size: 536,870,944 B
Max: 8,571,060,248 B
Used: 184,771,096 B

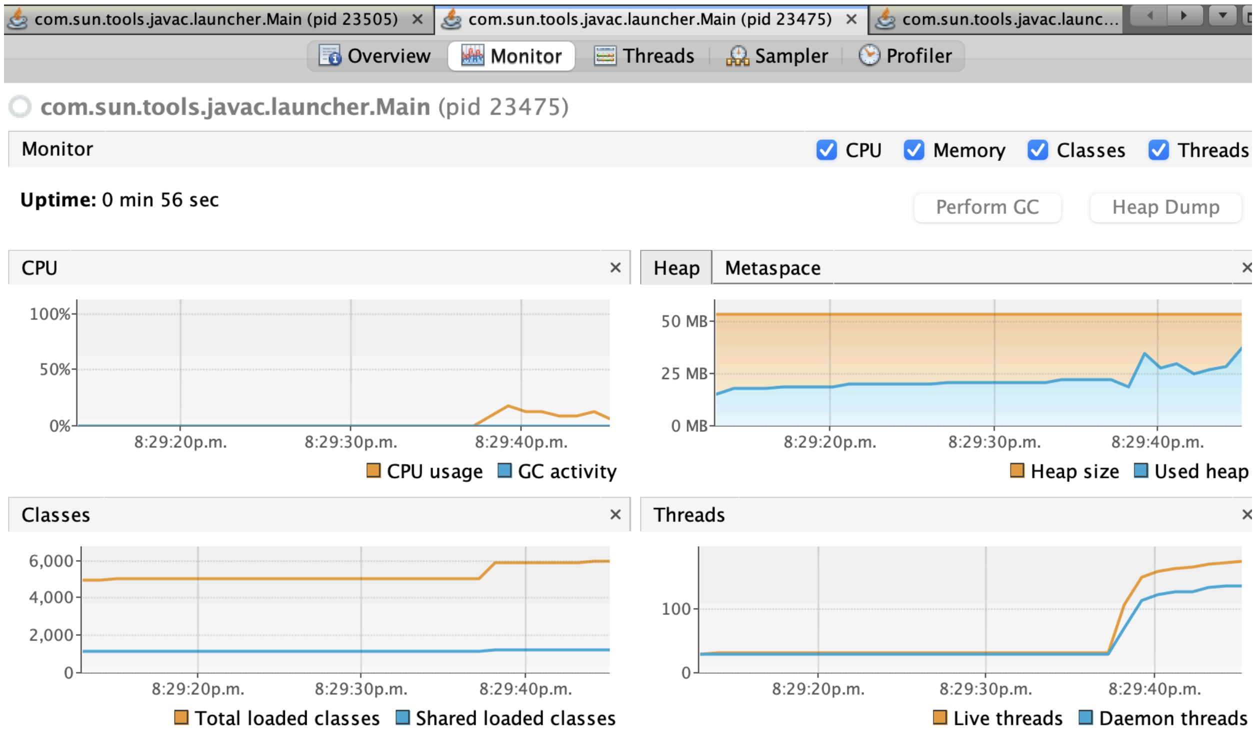
Total loaded: 4,046
Total unloaded: 0
Shared loaded: 1,104
Shared unloaded: 0

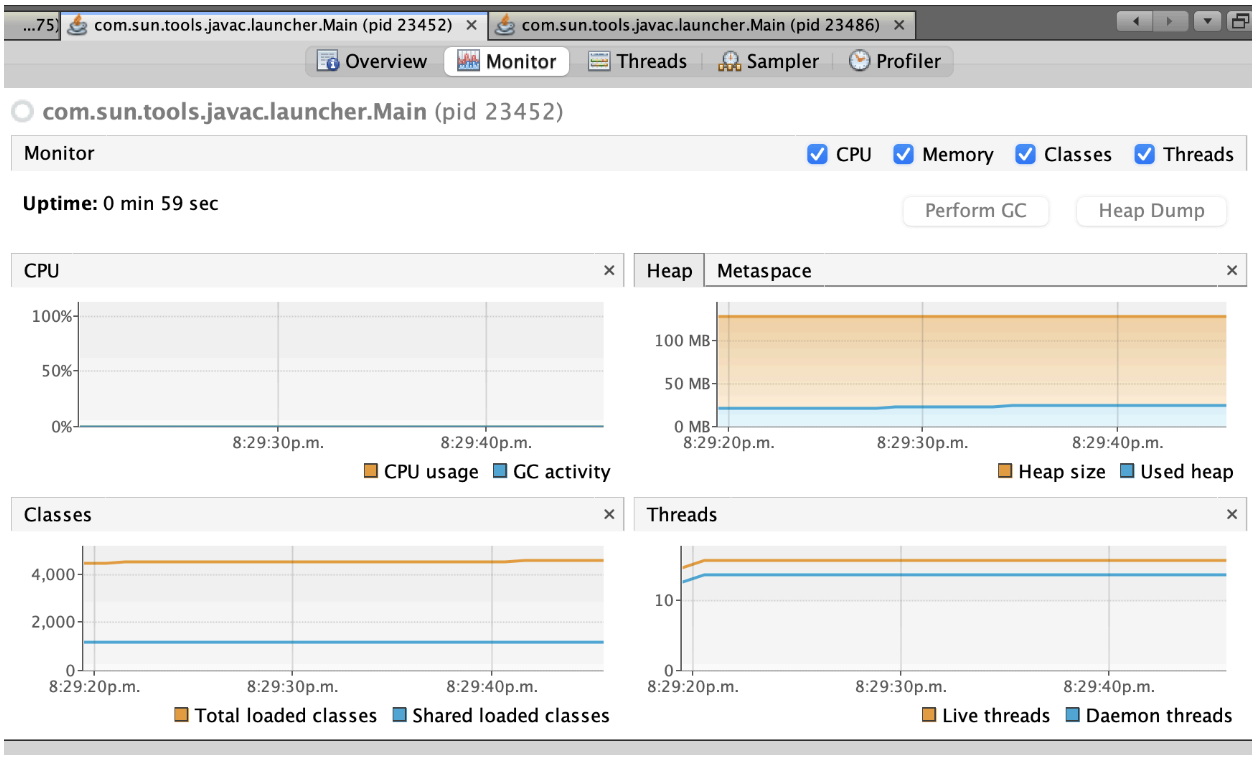
Live: 14
Live peak: 16
Daemon: 12
Total started: 17

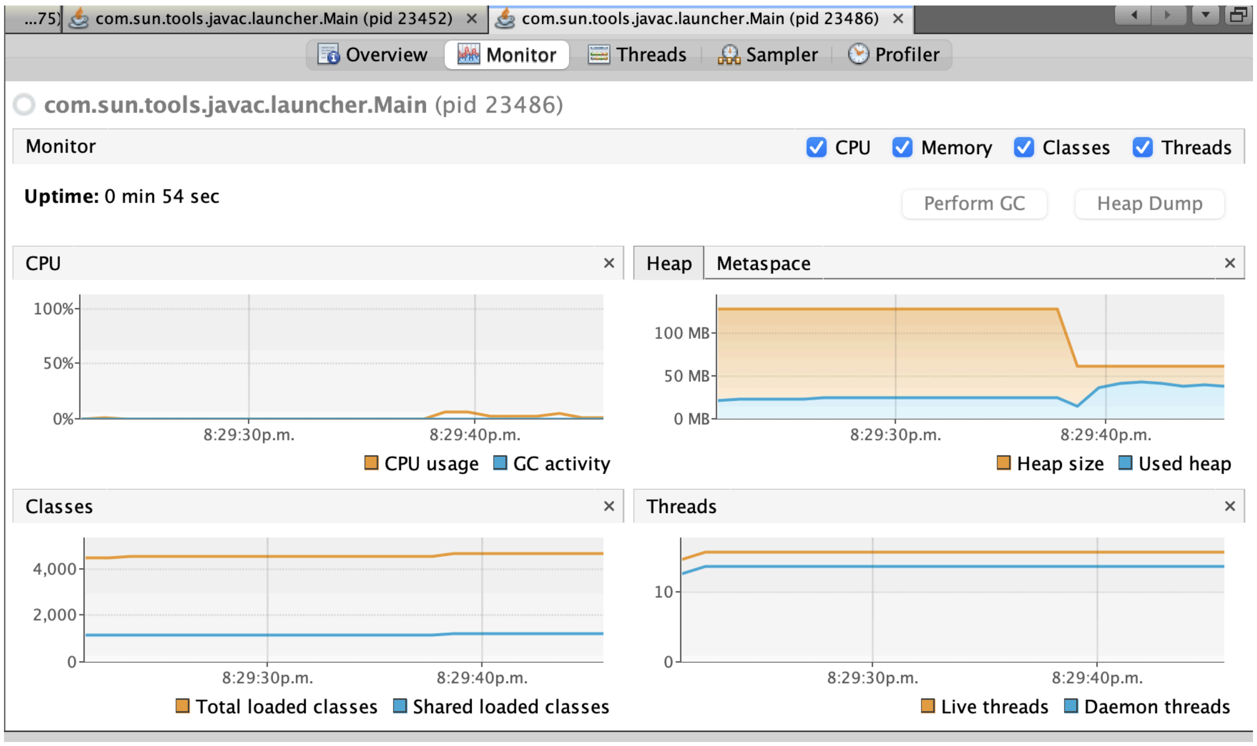
3rd profiling

For the second profiling we start seeing the need for not only multi-threading in the other services but also caching in our product and user service as they now have an increased amount of data. Caching this information would allow for faster retrieval and checking instead of calling our database and waiting to see the response. With all this implemented we see our final request per second which is around 900 for a load of 5000.
Total Requests: 5000
Elapsed Time: 5.51 seconds
Requests Per Second: 1006.96
Final Product




com.sun.tools.javac.launcher.Main (pid 23505)
com.sun.tools.javac.launcher.Main (pid 23475)
com.sun.tools.javac.launc...
Overview
Monitor
Threads
Sampler
Profiler
com.sun.tools.javac.launcher.Main (pid 23475)
Monitor
CPU
Memory
Classes
Threads
Uptime: 0 min 56 sec
Perform GC
Heap Dump
CPU
100%
50%
0%
8:29:20p.m.
8:29:30p.m.
8:29:40p.m.
CPU usage
GC activity
Heap
Metaspace
50 MB
25 MB
0 MB
Heap size
Used heap
Classes
6,000
4,000
2,000
0
Total loaded classes
Shared loaded classes
Threads
100
0
Live threads
Daemon threads


...75)
com.sun.tools.javac.launcher.Main (pid 23452)
com.sun.tools.javac.launcher.Main (pid 23486)
Overview
Monitor
Threads
Sampler
Profiler
com.sun.tools.javac.launcher.Main (pid 23452)
Monitor
CPU
Memory
Classes
Threads
Uptime: 0 min 59 sec
Perform GC
Heap Dump
CPU
100%
50%
0%
8:29:30p.m.
8:29:40p.m.
CPU usage
GC activity
Heap
Metaspace
100 MB
50 MB
0 MB
8:29:20p.m.
Heap size
Used heap
Classes
4,000
2,000
0
Total loaded classes
Shared loaded classes
Threads
10
0
Live threads
Daemon threads


...75)
com.sun.tools.javac.launcher.Main (pid 23452)
com.sun.tools.javac.launcher.Main (pid 23486)
Overview
Monitor
Threads
Sampler
Profiler
com.sun.tools.javac.launcher.Main (pid 23486)
Monitor
CPU
Memory
Classes
Threads
Uptime: 0 min 54 sec
Perform GC
Heap Dump
CPU
100%
50%
0%
8:29:30p.m.
8:29:40p.m.
CPU usage
GC activity
Heap
Metaspace
100 MB
50 MB
0 MB
8:29:30p.m.
8:29:40p.m.
Heap size
Used heap
Classes
4,000
2,000
0
8:29:30p.m.
8:29:40p.m.
Total loaded classes
Shared loaded classes
Threads
10
0
8:29:30p.m.
8:29:40p.m.
Live threads
Daemon threads