# JAVA AGGREGATOR LIBRARY PROJECT REPORT

## COMS 6156:
## TOPICS IN SOFTWARE ENGINEERING

**TEAM : APPLE**

RAKESH YARLAGADDA - ry2294

SHARAN SURYANARAYANAN - ss4951

## DELIVERABLES

We developed a Java Aggregator Library which helps people to parallelize their code and improve their performance without having to worry about creation and handling of threads. Clients can import the jar file of our Aggregator Library into their project to make use of it. You can find the jar file of our library at :
https://github.com/ry2294/JavaAggregatorLibrary/blob/master/AggregatorLibrary/AggregatorLibrary.jar

The source code of the Aggregator Library, a sample Java REST service using Aggregator Library and other documentations are available on github at:
https://github.com/ry2294/JavaAggregatorLibrary

## EXTERNAL SOFTWARE USED

We did not make use of any other libraries or software to build our Aggregator library. To build our sample REST Services we made use of:

- Jersey
- Reddit API
- AngularJS
- NodeJS
- PlotlyJS
- AWS Technologies

## DESCRIPTION

We have built a Java Aggregator Library which takes a set of tasks with their dependencies as input, builds a dependency graph for these tasks and executes these tasks as concurrently as possible. In the world of web services, it is commonplace for a web service to communicate with downstream services to process an API request from its clients. Web service developers try to decouple these actions into as many individual tasks as possible. However, some tasks end up as dependencies for other tasks to execute. Typically, web service developers execute these tasks in a sequential order one after the other. However, in this implementation, the throughput and latency of the web service will take a hit because each task is blocked on the previous task even though there is no dependency between them.

On the other hand, another way of implementation is to execute independent tasks in parallel through creation of threads, and use their result to process the dependent tasks. In this implementation even though it improves the web service's latency and throughput, the developers will have an overhead of thread creation and thread management. A better implementation for the above usecase is to decouple the tasks from their execution and hand it over to an executorservice which takes care of running tasks while honoring their dependencies on other tasks, which is our project idea. We want to build a task execution library, Java Aggregator Library, which allows web service developers to exploit parallelism with minimal overhead of handling thread creation and thread management at the same time honoring task dependencies.

## Example Use-case

Let us take an example to illustrate the problem our library tries to solve. In Fig 1, we can see four tasks **A, B, C** and **D** with their dependencies. Task A is independent and can execute first, whereas Tasks B and C depend on A since they use Task A's output. Finally Task D is dependent on outputs of Tasks B and C and hence it has to execute last. So typically, the common schedule of execution that developers prefer or use is sequential which is shown below Fig 2. In the below execution, the main problem is that for web services which are latency critical, will affect their request handling time since developers are not utilizing the independency of Tasks B and C. Fig 3 shows an implementation of these tasks in Java for their sequential execution.

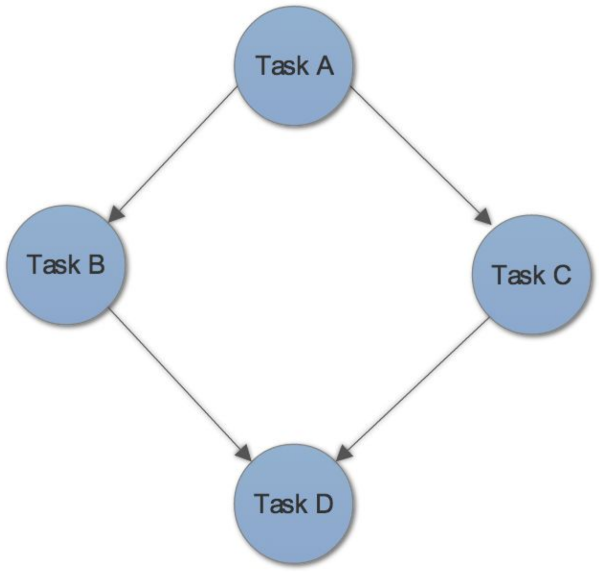


Fig 1 Tasks Dependency Graph

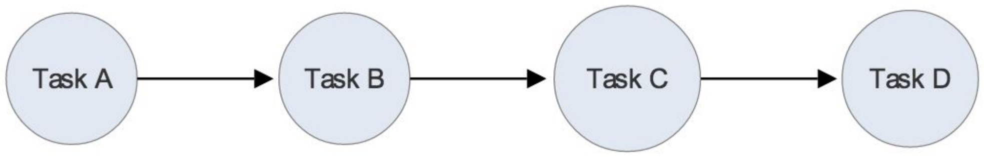
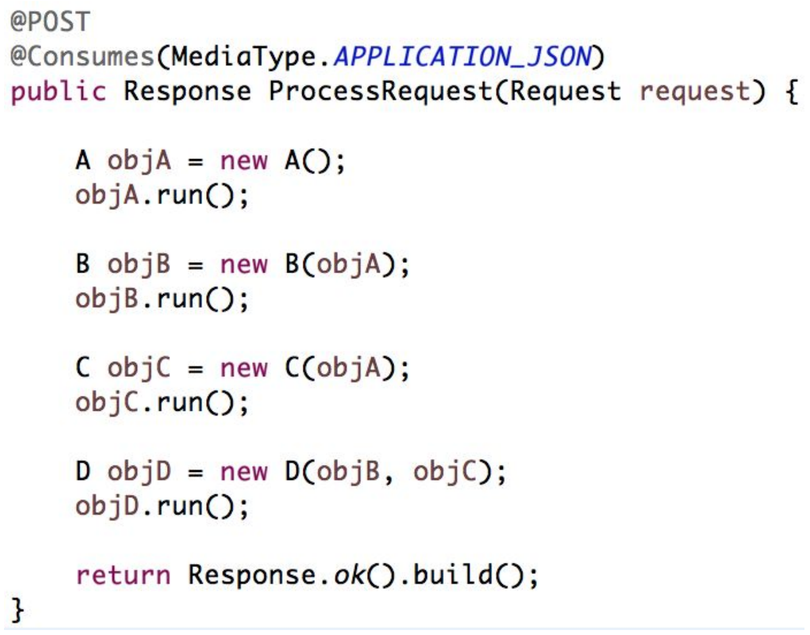


Fig 2 Tasks Execution Schedule



```java
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response ProcessRequest(Request request) {

    A objA = new A();
    objA.run();

    B objB = new B(objA);
    objB.run();

    C objC = new C(objA);
    objC.run();

    D objD = new D(objB, objC);
    objD.run();

    return Response.ok().build();
}
```

Fig 3 Java Code for Sequential Execution of Tasks

The ideal way of execution would be to implement Task B and C in parallel by creating two parallel threads say, Thread B and Thread C which execute Task B and Task C respectively. Finally the main thread will wait for the child threads to complete and finally execute Task D. Below is their invoking strategy for parallel execution.

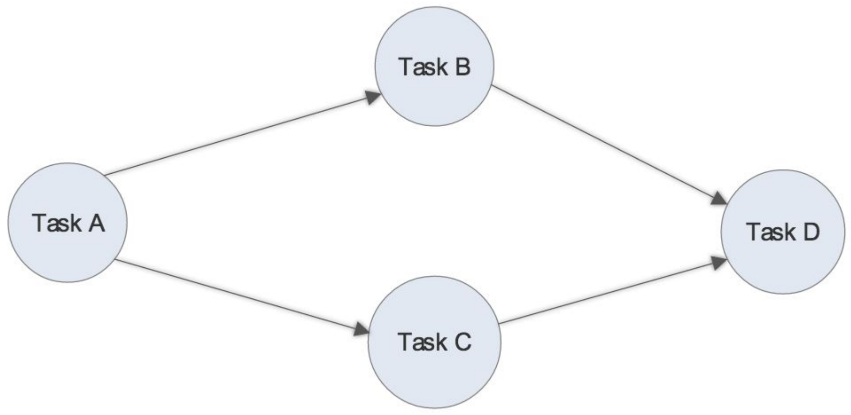
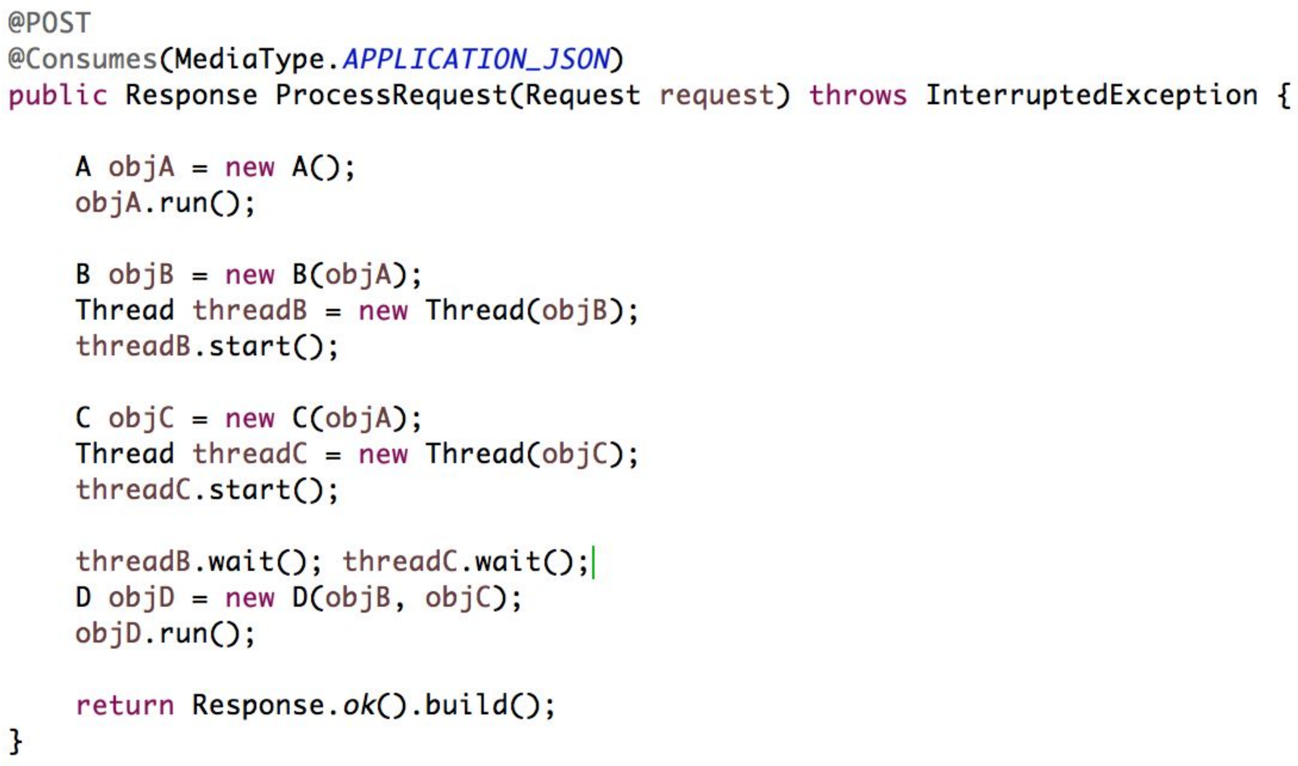


Fig 4

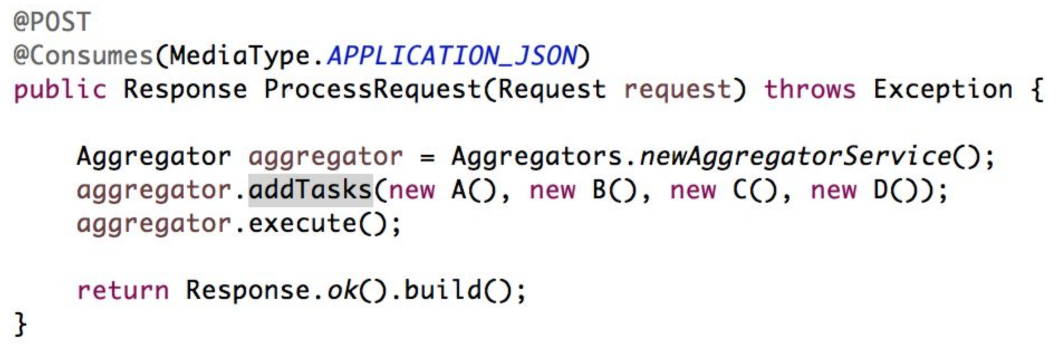
To achieve the above parallel execution of independent tasks, Fig5 describes the Java code which creates threads and executes these tasks.



```java
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response ProcessRequest(Request request) throws InterruptedException {

    A objA = new A();
    objA.run();

    B objB = new B(objA);
    Thread threadB = new Thread(objB);
    threadB.start();

    C objC = new C(objA);
    Thread threadC = new Thread(objC);
    threadC.start();

    threadB.wait(); threadC.wait();
    D objD = new D(objB, objC);
    objD.run();

    return Response.ok().build();
}
```

Fig 5

The main problem in the above code is that, the developers have to deal with thread creation, management, and dependency injection of the output results to dependent tasks. This is the problem, that our aggregator library tries to address and mitigates it, by taking away the burden of thread creation, management and dependency injection from developers. Below is the rewritten code using our aggregator library.

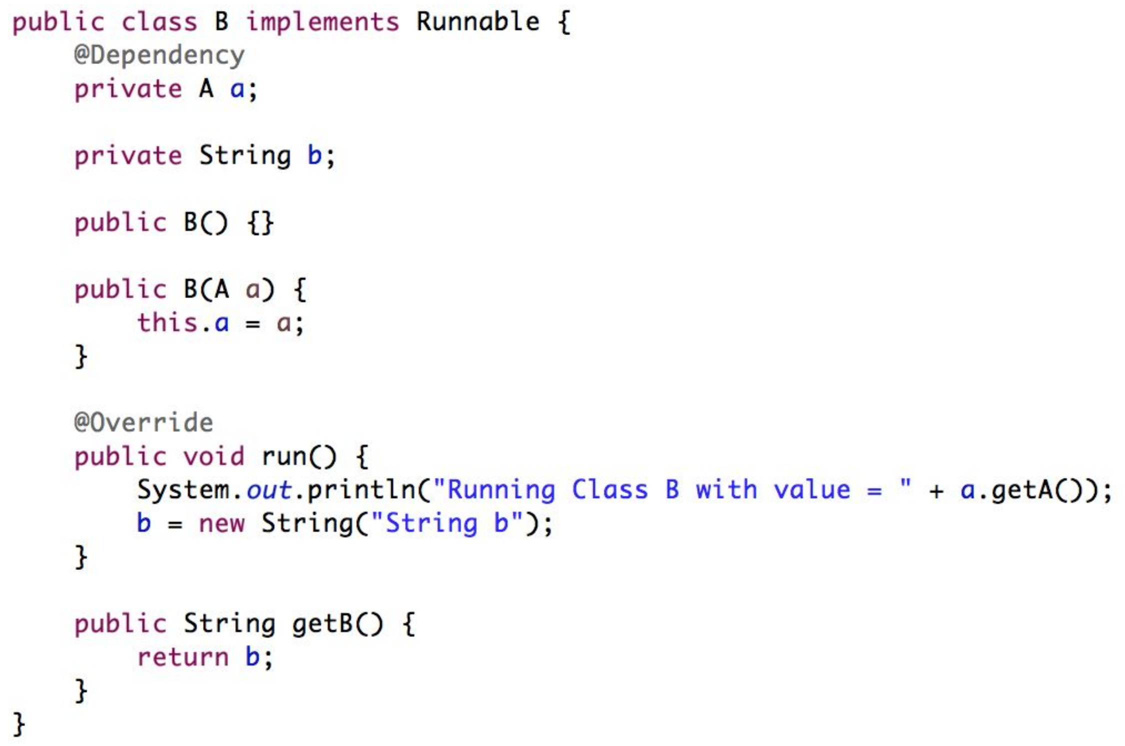


```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response ProcessRequest(Request request) throws Exception {

    Aggregator aggregator = Aggregators.newAggregatorService();
    aggregator.addTasks(new A(), new B(), new C(), new D());
    aggregator.execute();

    return Response.ok().build();
}
```

Fig 6

## Dependency Identification and Injection

For developers to specify dependencies among tasks we have created an Annotation using Java Annotations which is **@Dependency**. Developers need to insert this annotation in each task which has dependencies above their references to these dependencies. Fig 7 describes the usage of **@Dependency** in Task B which has a dependency on Task A.



```
public class B implements Runnable {
    @Dependency
    private A a;

    private String b;

    public B() {}

    public B(A a) {
        this.a = a;
    }

    @Override
    public void run() {
        System.out.println("Running Class B with value = " + a.getA());
        b = new String("String b");
    }

    public String getB() {
        return b;
    }
}
```

Fig 7

## Aggregator Library Architecture

Below is architecture diagram describing individual components of our aggregator library.

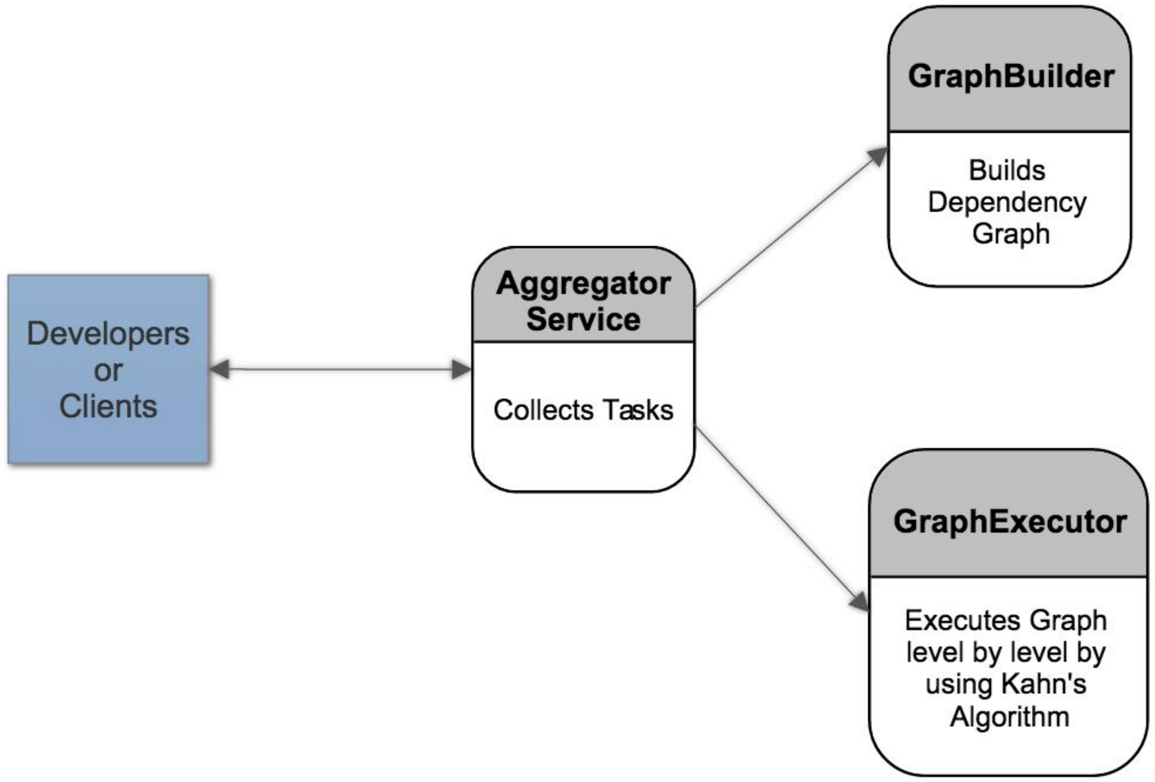


Fig 8

From the above diagram we can see that, our library mainly has three components which termed as **AggregatorService**, **GraphBuilder**, **GraphExecutor**. Our first component AggregatorService interacts with clients to collect their tasks, and passes them first to GraphBuilder. The GraphBuilder's main purpose is to perform dependency injection for tasks and build a graph using the dependencies. Then AggregatorService checks if there exists a cycle in the graph and if so it throws an exception and exists. If the graph is acyclic, then it injects the first level of tasks into GraphExecutor. Now comes the role of the GraphExecutor which manages the thread pool that it has created to execute the tasks in parallel. GraphExecutor implements the Khan's Algorithm of Graph level by level traversal to execute the tasks. It starts executing the tasks as soon as their dependencies finish executing.

## DESIGN

### Class Definition for Each Node/Task

The user needs to define a class which implements Runnable for every node/task he wishes to add. This class must override run function. The user can specify the tasks on which this task is dependent on by making use of the @Dependency annotation. A sample implementation of a task is illustrated in Fig 7.

### Thread Pool

We have used a Factory design pattern for the thread pools. But we have currently implemented only one version, a singleton cached thread pool. We have designed a singleton thread pool because we want to have only one thread pool at any given point of time. We have currently fixed number of threads in the thread pool to be equal to the hardware concurrency, but we plan to implement functions which allow the user to set the number of threads in the next version. The thread pool picks up tasks from the thread-safe task queue and runs them.

### Graph Builder and Execution

We made use of Kahn's algorithm to build the graph and execute it. The Graph Builder calculates the in-degree and out-degree for each task. In-degree is the set of nodes the current task is dependent on, and Out-degree is the set of nodes which are dependent on the current task. After calculating in-degrees and out-degrees, Graph executor will select all the Tasks which have in-degree as 0 (which are independent tasks). These tasks will be pushed to the Graph Executor which communicates with the Thread Pool to execute the tasks. The Graph Executor will return completed tasks to Graph Builder. For each completed task, Graph Builder reduces the in-degree of completed task's dependents. Finally, for each dependent it checks if the it's in-degree has been reduced to 0. If so, it will push it to the Graph Executor else skips.

## Demo Application Details

To test our Java Aggregator Library, we have created 3 REST Webservices which handle real time requests from clients. We have created a single client application which bombards requests to all three REST Webservice to test how our library is performing.

**3 REST WebServices**

- **Sequential REST Service** (executes tasks in Sequential Order. Fig 3)
- **Parallel REST Service** (executes independent tasks in Parallel. Fig 5)
- **Aggregator REST Service** (uses Aggregator Library for task execution. Fig 6)

For each request, all the three services will be executing tasks which have dependency graph shown in Fig 9.

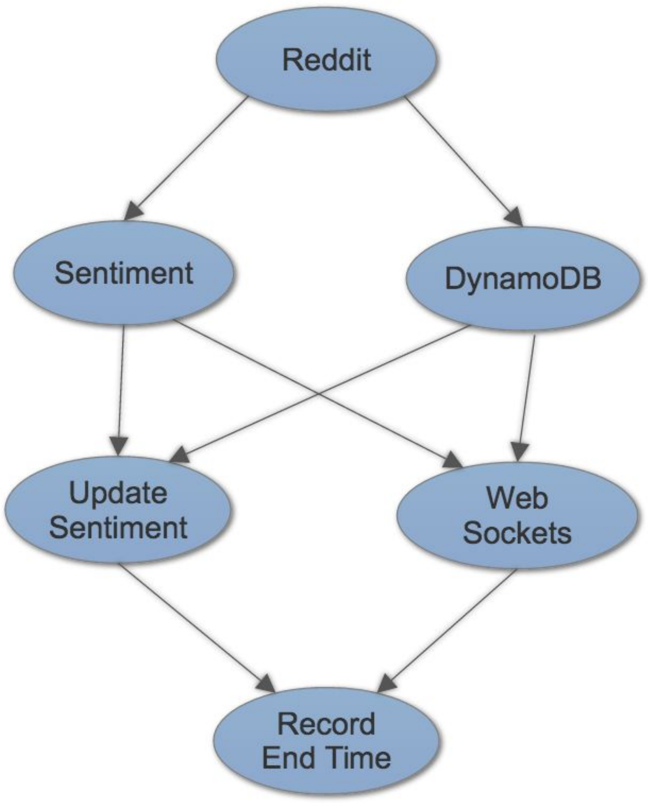


Fig 9: REST Services Dependency Tasks Graph

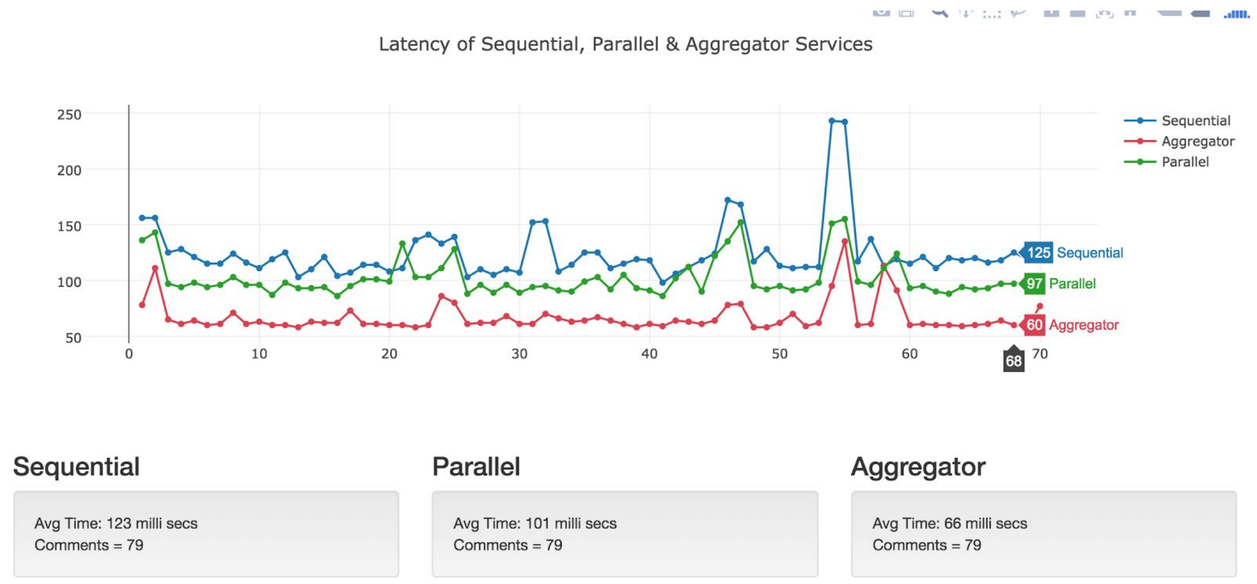
Once the Java webservice is up and running we will start monitoring each webservice for the below metrics.

- **Average Request Latency**
- **Average Request Throughput**
- **Average CPU Utilization**

Our client application listens to a streaming api of **Reddit** website. The client collects the comments posted by Users and calls all the three services at a time. Once each service receives a request it performs sentiment analysis by calling **sentiment140** webservice and storing the reddit comment in **DynamoDB**. These two tasks are executed in parallel. Then, the next level of tasks are to update the sentiment information returned from sentiment140 into **DynamoDB** and emitting the event to **Web-Sockets** to publish the data onto our website. The final task is to record the time taken and store it again in DynamoDB for the incoming request id. So the final task is dependent on the fourth and fifth tasks.

## RESULTS

Fig 10 shows the Latency results of the 3 Rest services. We can see that Average request handling time for Sequential(123 ms) is highest followed by Parallel(101 ms) and finally Aggregator(66 ms). One can expect for the sequential to be larger than the other two and Parallel and Aggregator to be on par or Aggregator to be on the higher side. However, in our case it got reversed. **Aggregator REST Service performed far better than Parallel REST Service** in terms of Average Request Handling Time**.**



Sequential

Avg Time: 123 milli secs
Comments = 79

Parallel

Avg Time: 101 milli secs
Comments = 79

Aggregator

Avg Time: 66 milli secs
Comments = 79

Fig 10: Average Request Latency of REST Services

## FINDINGS

The reason for Aggregator REST Service performing far better than Parallel REST Service in terms of **Average Request Handling Time** is that, in case of Parallel REST Service, for each request, **multiple threads are spawned and are destroyed once the request gets completed**. For each thread created, there is an overhead in any system to allocate stack frame memory and deallocate it once the lifetime of the thread is completed.

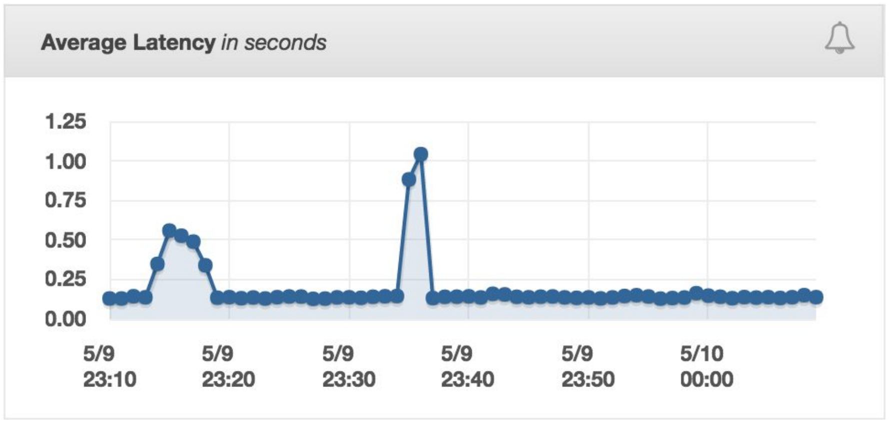
Also, another major problem with the Parallel REST service is that the number of threads created is in proportion to the number of incoming requests. However, there is a limit to the number of threads that can be created or the number of threads that can be supported by any system. Ideally, any application has to find out the number of concurrent threads that can be supported by the system on which the application is running and limit its threads creation to a number that the system can support. In case the application fails to adhere to this property, then the system undergoes into **high context switch between multiple threads by the processor** and can lead into to degraded Average Request Handling Time. This is what is happening in the case of Parallel REST service. The Parallel REST service is creating threads with respect to incoming requests count and does not adhere to the number of threads the system can actually support. These are the main reasons that have degraded the Parallel REST Service's Average Request Handling Time.

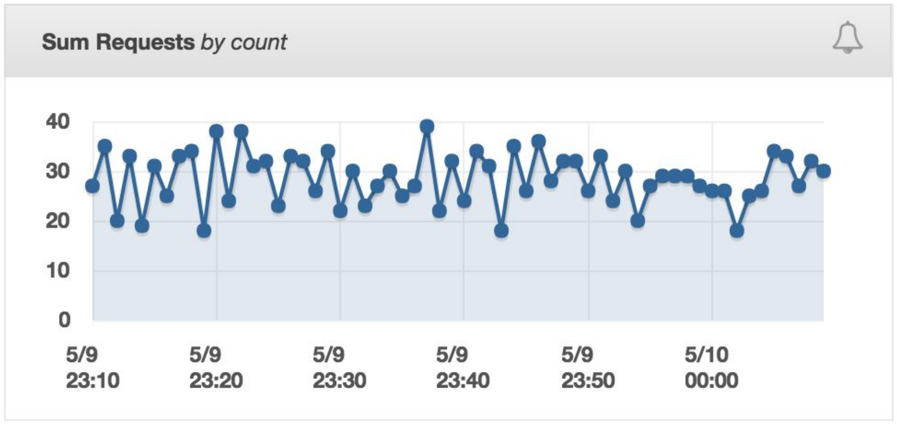
In the case of Aggregator REST Service, the thread creation and management is handled by the Aggregator Library. Aggregator Library does does create threads for incoming requests. However, it does not destroy the threads once their work is completed. It reuses them for the new incoming requests. So, the Aggregator Library saves thread destruction time by reusing the threads for the new incoming tasks. Also the Aggregator Library does not create threads indiscriminately. When the Aggregator Library is first invoked, it calculates the number of threads that the system can support by calling the **Runtime.getRuntime().availableProcessors()** method of Java. **This method returns the number of available processors that can support parallel execution, which is used by the Aggregator Library to restrict its thread creation so that the processor does not go into high context switch**. Due to these reasons, the Aggregator REST Service outperforms Parallel REST Service in terms Average Request Handling Time.

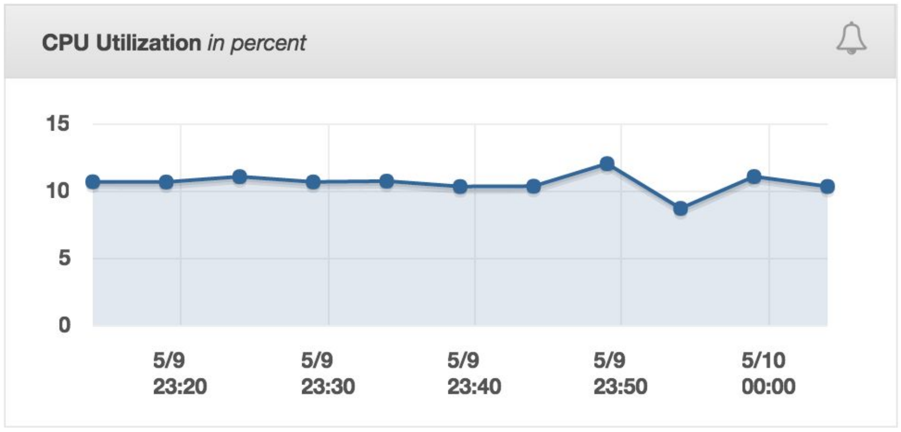
## Metrics

Below figures show the **Latency, Throughput and CPU Utilization** metrics of the 3 REST Services respectively. Apart from deviation in Latency metric for which we have explained in Findings section, we didn't find any deviations or abnormalities in CPU Utilization or Throughput in Requests which can be seen in the following diagrams.

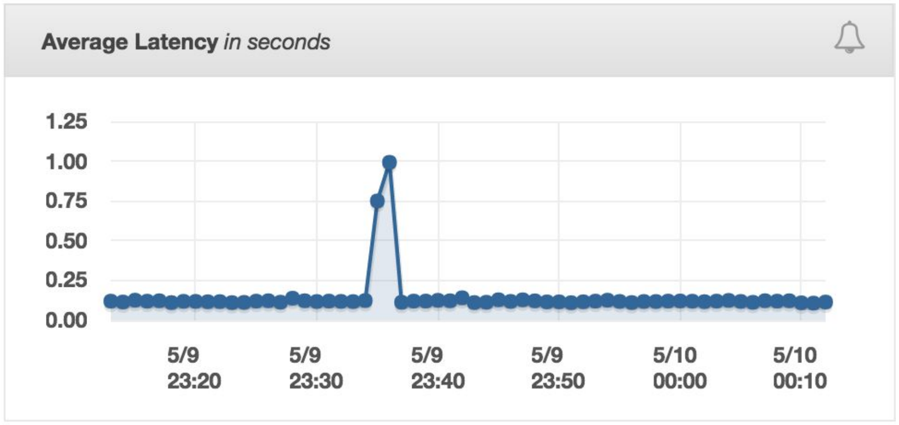
### Sequential REST Service

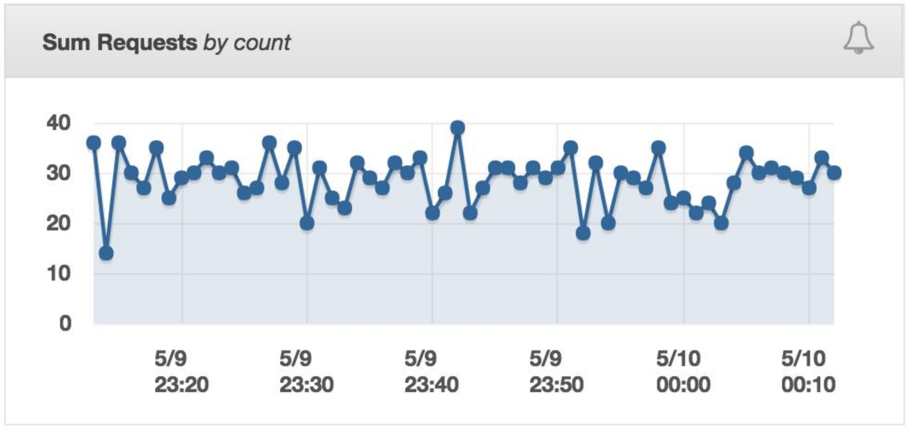


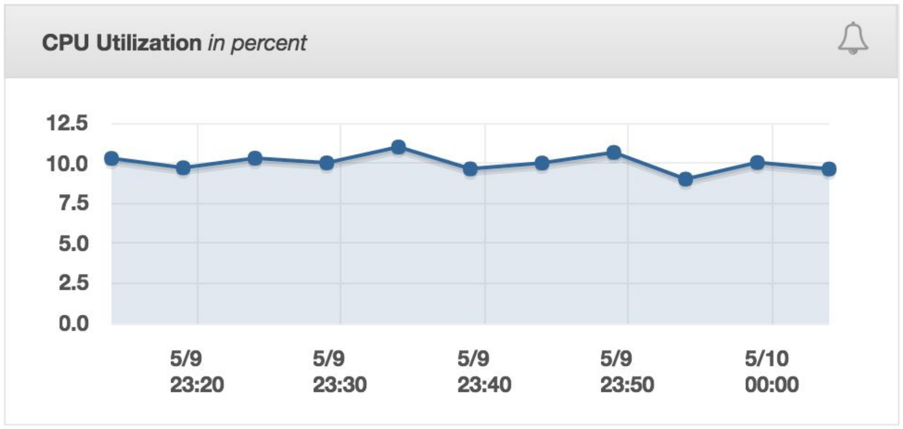




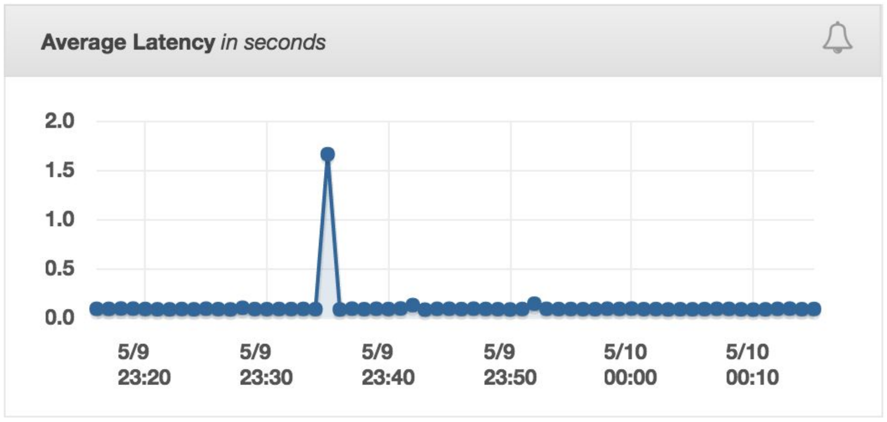
## Parallel REST Service

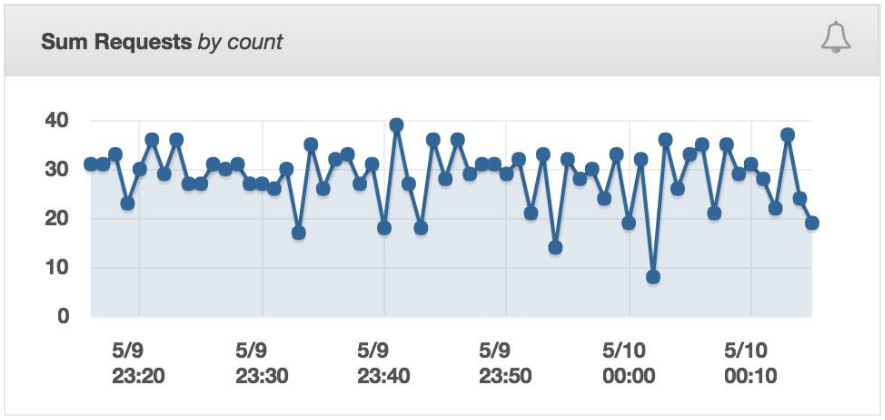


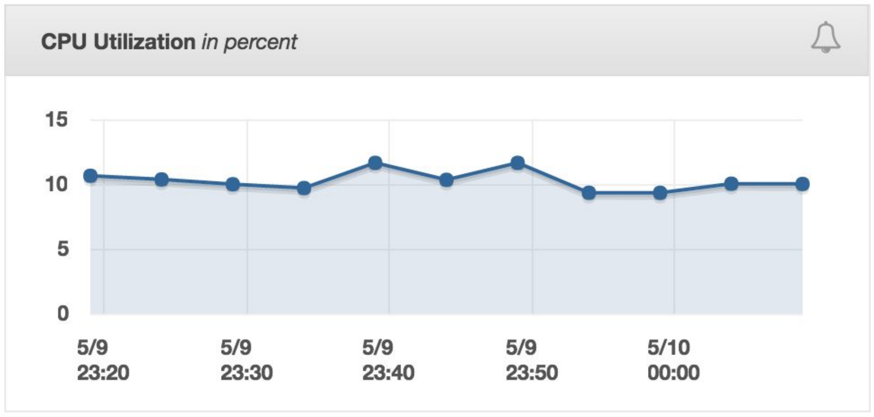




## Aggregator REST Service







## What we learned ?

**Rakesh**

I came up with the project idea of Aggregator Library. I thought this as an extension of my Mid Term Paper on REST vs SOAP Architectural Styles. After working on my paper I identified that many real world web services, be it SOAP or REST, do communicate with downstream services, and that these tasks can be parallelized which will be an overhead for developers. From this idea, we have started designing our Aggregator Library. For implementing Graph Builder component, I learned about Khan's Algorithm for Graph Traversal and implemented it to suit our needs. In the case of Graph Executor component I learned about Java Threading concepts such as Thread Pools and Blocking Queues. I learned about Java Reflection for identifying dependencies and injecting them into Tasks. Also I learned about Java Annotations for implementing **@Dependency** annotations and using them with Java Reflection for Dependency Injection.

**Sharan**

I have implemented the AggregatorService component of our Library for which I have learned Factory Design Pattern to create an Aggregator Factory. For implementing the Client side reddit streaming application I have learned about NodeJS server technologies. I have got hands-on experience in using AWS Technologies such as AWS Elastic Beanstalk application. We have used Elastic Beanstalk to deploy all our REST services along with our Reddit Website and Client side Streaming application. Finally, I was able to learn how to generate Javadocs for our Java Aggregator Library Documentation which will help our clients or end users in using, understanding source code and debugging our library.

## Any Existing Libraries ?

We have looked up in internet, questioned in Java Webservices discussion groups and forums to identify if there are any existing libraries which solve the problem that our Aggregator Library is addressing, **so that we are not reinventing the wheel**. However, we haven't come across any such library or framework which addresses this problem.

## Prospective User Community

Our Aggregator library is targeted towards webservice developers or clients who have dependency on downstream services for processing their requests. However, our

Aggregator library isn't restricted to just webservices. Even standalone applications can use for executing their tasks. Our Aggregator Library is similar Java's Collection Framework where users or clients can plug and play with them.

## Planned to do But Didn't Do ?

There wasn't any feature that we left or didn't implement. We were able to complete all the features of the Aggregator Library as proposed in Project Proposal. Also, we were able to test our application statically (using unit tests) as well as in a real world scenario (using demo web services application). We analyzed the results of the metrics and were able to identify why there was a deviation in Latency for Parallel and Aggregator services i.e. why Aggregator outperformed Parallel (explained in Findings section).

## Challenges Encountered

Mainly, the challenges that we encountered was during the implementation of the GraphExecutor component of our Aggregator Library. We were in a dilemma whether, to create Threads on the fly and destroy them once their work is completed, or to create a Thread Pool and reuse its threads. We wanted to reuse our Threads instead of re-creating and destroying them. To achieve that we have designed and implemented the **GraphExecutor** component which creates our **ThreadPool** and reuses the threads. This helped our Aggregator REST Service to outperform Parallel REST Service in terms of Average Request Handling Time (Latency).

## CONTRIBUTION

Rakesh

- GraphBuilder
- GraphExecutor
- Creation of Services
- Website for tracking metrics

Sharan

- AggregatorService
- Client Application
- Monitoring of Metrics
- Documentation