|  |
| --- |
| Towson University, Fall 2015 |
| Groovy Distributed Task Management System |
| https://github.com/mannyrivera2010/barium |

|  |
| --- |
| COSC 880  Towson University  Department of Computer and Information Sciences  Advisor: Dr. Josh Dehlinger  Emanuel Rivera  Eriver6@students.towson.edu |

Contents

[Acknowledgements 4](#_Toc437866082)

[Keywords 4](#_Toc437866083)

[Abstract 4](#_Toc437866084)

[Introduction 5](#_Toc437866085)

[Motivation 5](#_Toc437866086)

[Overview 5](#_Toc437866087)

[Terminology 5](#_Toc437866088)

[System Architecture 6](#_Toc437866089)

[High-level Architecture Concept 7](#_Toc437866090)

[Task States 7](#_Toc437866091)

[Owner Point of View 7](#_Toc437866092)

[Worker Point of view 7](#_Toc437866093)

[Job Life Cycle 8](#_Toc437866094)

[Task Life Cycle 8](#_Toc437866095)

[Publish/Subscribe and Queues 8](#_Toc437866096)

[REST Interface 8](#_Toc437866097)

[Technology Background 8](#_Toc437866098)

[Backend Focused Technology 9](#_Toc437866099)

[Netty 9](#_Toc437866100)

[Hazelcast 9](#_Toc437866101)

[Groovy/ Java 10](#_Toc437866102)

[Gradle 10](#_Toc437866103)

[Node.js 10](#_Toc437866104)

[Front-end Focused Technology 10](#_Toc437866105)

[AngularJS 10](#_Toc437866106)

[Grunt 10](#_Toc437866107)

[Implementation 11](#_Toc437866108)

[Barium 11](#_Toc437866109)

[REST Interface 11](#_Toc437866110)

[Message Formats 13](#_Toc437866111)

[WebSocket JSON Message Format 13](#_Toc437866112)

[WorkerEvent Topic JSON Message Format 14](#_Toc437866113)

[BariumUI 14](#_Toc437866114)

[Dashboard 14](#_Toc437866115)

[File Server 15](#_Toc437866116)

[Development Process 17](#_Toc437866117)

[Issues during development 17](#_Toc437866118)

[Future Improvements 17](#_Toc437866119)

[Owner/Worker Framework Backend Improvements 17](#_Toc437866120)

[Distributed class loading between different nodes 17](#_Toc437866121)

[Isolation of the Task Execution 18](#_Toc437866122)

[Worker Failure Case 18](#_Toc437866123)

[Historical view of jobs 18](#_Toc437866124)

[Owner/Worker Framework Front-end Improvements 18](#_Toc437866125)

[Keep developing and improving Barium UI 18](#_Toc437866126)

[RAML 18](#_Toc437866127)

[Better Software Testing 19](#_Toc437866128)

[Major Code Refactor 19](#_Toc437866129)

[Fully feature Website 19](#_Toc437866130)

[Multi-Broker Support 20](#_Toc437866131)

[Lessons Learned 20](#_Toc437866132)

[Conclusion 20](#_Toc437866133)

[Appendix 21](#_Toc437866134)

[Proposal 21](#_Toc437866135)

[Project Description/Abstract: 21](#_Toc437866136)

[Project Topics to be studied: 21](#_Toc437866137)

[Project Objectives: 21](#_Toc437866138)

[Deliverables: 21](#_Toc437866139)

[Anticipated Project Timeline: 22](#_Toc437866140)

[Reading/References List: 22](#_Toc437866141)

[Completed Graduate Courses: 22](#_Toc437866142)

[Projects Done in Other Courses: 23](#_Toc437866143)

[Project Description 23](#_Toc437866144)

[References 23](#_Toc437866145)

# Acknowledgements

I would like to express my appreciation to my manager and coworkers at Comsort for the constructive suggestions during the planning and development of this project. I would like to also express my appreciation to my advisor Josh Dehlinger for the support during this project with feedback. Finally would like to thank the graduate committee board to reviewing my paper and presentation.

# Keywords

Distributed Architecture, Distributed Application, AngularJS, Grunt, Front-end, Back-end, Groovy, Netty, Hazelcast, Worker, Task, Task Management, Distributed Task Management, NodeJs, Express, File Server, Java, Express Framework, Gradle, Gradle Build Automation System, Jade, Multer, Server-index node package, Amazon Web Services

# Abstract

The system is designed to meet the needs of generically using many machines for distributed computing by providing a framework to software developers. It works by dividing a computing job that takes large amounts of processing power into independent smaller computing tasks. The benefit of this is that you can use many machines and reduce the processing time. Developers are responsible of developing two classes. The Owner is responsible on putting tasks into a queue and monitoring progress of a job. The task class is used for the worker nodes to execute.

# Introduction

## Motivation

This project started while I was working at Comsort; they had a need to meet the demands of the ability to deal with growing data size and processing graph data. With the growth of data, the time needed to process and load data became infeasible to do with one machine. One question that I had was “Can we make the data loading faster and utilize more of our servers by distributing computing power usage?”. The purpose is to facilitate a way to serve the application’s computation power between many different machines. I started to create a system based on queues and publish/subscribe model called the Owner/Worker framework to distribute work among many computers.

## Overview

The Owner/Worker framework has the following requirements:

* Should have ability to use the processing power of many CPU cores and RAM over many machines
* Should have reliability that a unit of work (task) will be done, if the task fails the developer shall handle the rescheduling the task for re-execution
* The framework shall be generic so it can be used for other processes that require the use of many machines for tasks
* Tasks shall be based on using Interfaces which allows the developer to put their implementation code
* System shall have minimum configuration and easy to deploy (currently the framework has one configuration file for connections information)
* Shall use message passing between processes
* Extend TaskOwnerBase Abstract class to make your own TaskOwner to handle the Task
* Framework shall have the ability to add more workers to execute more tasks without the developer needed to deal with of coordination between nodes

## Terminology

During the development of the system, terms were created to describe the different aspects of the framework.

**Owner/Worker Framework (Barium)** - an asynchronous distributed task management system framework that allows developers to have a generic way to execute code on multiple machines. The owner/worker framework is also known as Barium.

**BariumUI-** BariumUI is a front-end which consume barium RESTful API powered by AngularJS web application framework. It is used to view the progress of task and receive notification via WebSockets.

**Owner** – the owner is responsible for generating and putting tasks into the queue for workers to be executed in a distributed way. The owner is also responsible for monitoring tasks, providing a REST interface to client, and notifications with WebSockets.

**Task**- a task is a unit of work, internally this creates a thread on a worker’s thread pool which creates an instance of a class wrapper using the class path of the location of the task class thread running on a worker instance. The task wrapper class is used to monitor the execution of the task executed to notify the owner of the state. The worker will publish a message to the owner for both failures and success.

**Job**- a job is the execution of all tasks generated by an owner till it has finished all the tasks.

**Worker**- the worker is responsible for executing a task and publishing all results to the topic that the owner listens to.

**RESTful Interface**- rest represents a software architectural style that communicates via the HTTP protocol to expose resources using HTTP verbs.

# System Architecture

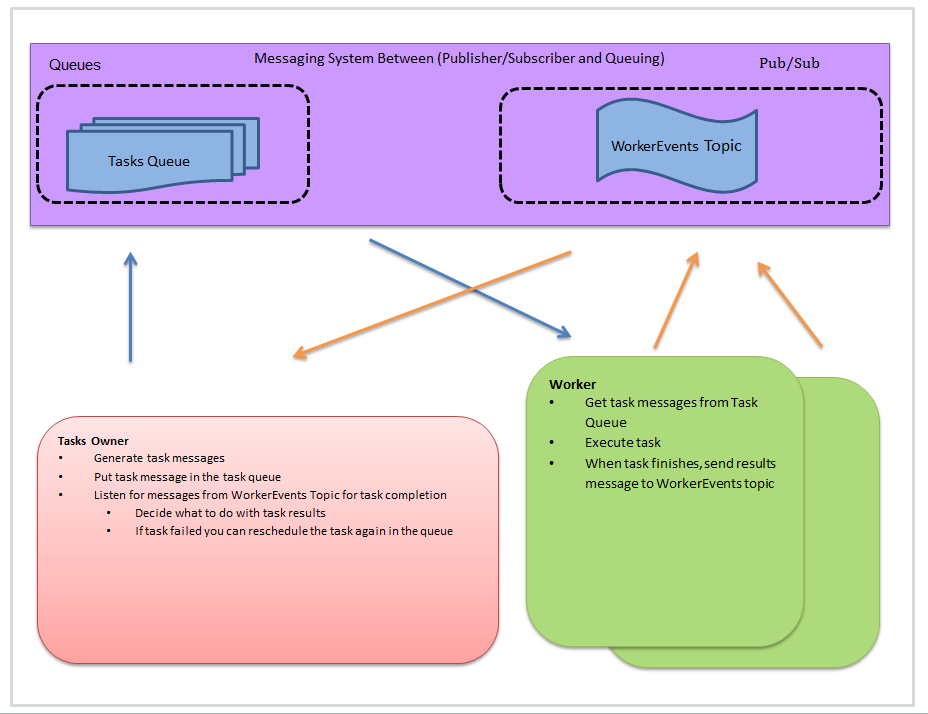
The framework is comprised of four key components: Owner, Worker, Queue, and Publish-subscribe system. Each component has a designated role in the system. 

Figure - This diagram shows the Owner/Worker Framework’s main components

## High-level Architecture Concept

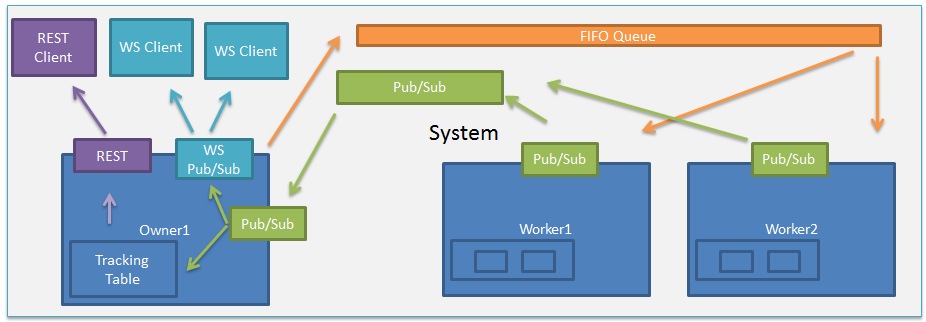


Figure – This diagram show all of the Owner/Worker Framework components

The system architecture above shows a more detailed view of the Owner/Worker Framework. The sections below explain more in depth how each component works and interacts with other components.

### Task States

A task can have 4 different states: Scheduled, In Progress, Finished, and Failed. A scheduled task means that the owner has scheduled it in the task queue. When an Owner put a task in the queue, that task will become ‘Scheduled’ state. Each task of the system has a unique task id. It will use a map data structure to persist the task’s state into memory. The owner asynchronous waits till it receives a message from the worker event topic (publish/subscribe) with the task id stating a worker got the task for execution. At that time the owner changes the state of the task as ‘In Progress’ using the task id to change value in the map store. The owner then waits till it gets another message stating that a task has been finished executing successfully or with an error. The owner will accordingly change the state in the map with each operation that happens.

### Owner Point of View

An owner starts the job by scheduling tasks into the tasks queue. The owner has the ability to do a barrier to wait for all previously tasks. A barrier for a group of tasks in the implementation of the owner means any task in-progress state tasks must stop at this point and cannot proceed until all other tasks reach to the finished state. When the owner schedule’s a task it updates the tracking table as scheduled state. The owner asynchronous waits it get a message from pub/sub topic with the new state of the task. At this point the task can either be finished state or error state. For both of those state will it will update the state in the tracking table.

### Worker Point of view

The worker is responsible for executing tasks it pulls from the task queue and publishing the state of task. The worker has a thread pool used to run instances of tasks. The thread pool has a default size of the number of cores of the machine it is running on. For example if the worker is started on a machine with 6 cores the thread pool will have 6 available threads to run tasks with. The tasks are ran concurrency which means that all 6 tasks running at the same time independent from each other. The worker will publish one message to the owner when it starts working on the task and another message when it has finish executing the task. The task object from the queue contains the class path of the task which is used to dynamically make an instance of the class.

### Job Life Cycle

A job life cycle can be defined as a group of tasks that get generated, executed, and finished. The owner schedules 5 tasks it represent the start of the job life cycle, when the owner receives that all 5 task has finished it represent the end of the job

### Task Life Cycle

A task life explains how a task views the process of getting class initialization and completed. The worker obtains the task class path string value that it needs to be able to dynamical load the task class. A task has the ability to send a message anytime to the WorkerEvents Topic that owner listens to. This can be useful for publishing miscellaneous message that could be used for different use cases.

### Publish/Subscribe and Queues

Communication between the owner and workers use the publish/subscribe model and queues. The queue is used to store tasks object. The publish/subscribe model is used to communicate between the worker and owner nodes. The owner is the consumer of WorkerEvents topic with all the workers publishing the results of task to it. The owner is responsible for putting task into the queue for the workers to get. The WorkerEvents topic is mainly used to communicate the status of the tasks.

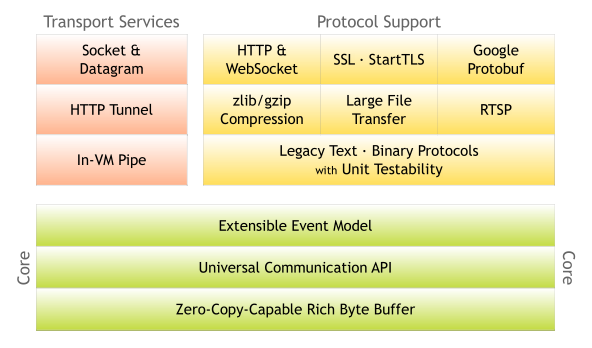
### REST Interface

The REST Interface allows the clients that used the HTTP protocol to communicate with the services provided by the owner. RESTful interface serves clients with JSON. The REST interface allows for a client to have knowledge of the state of all tasks and other statistics.

# Technology Background

The Owner/Worker framework uses different technologies for the scheduling, execution, and providing a web user interface for tasks. The complete stack of the technologies is focused on being backend and front-end driven. The backend technologies are used for the management and execution of tasks. The front-end technologies are used to display a web application that consumes the backend services.

## Backend Focused Technology

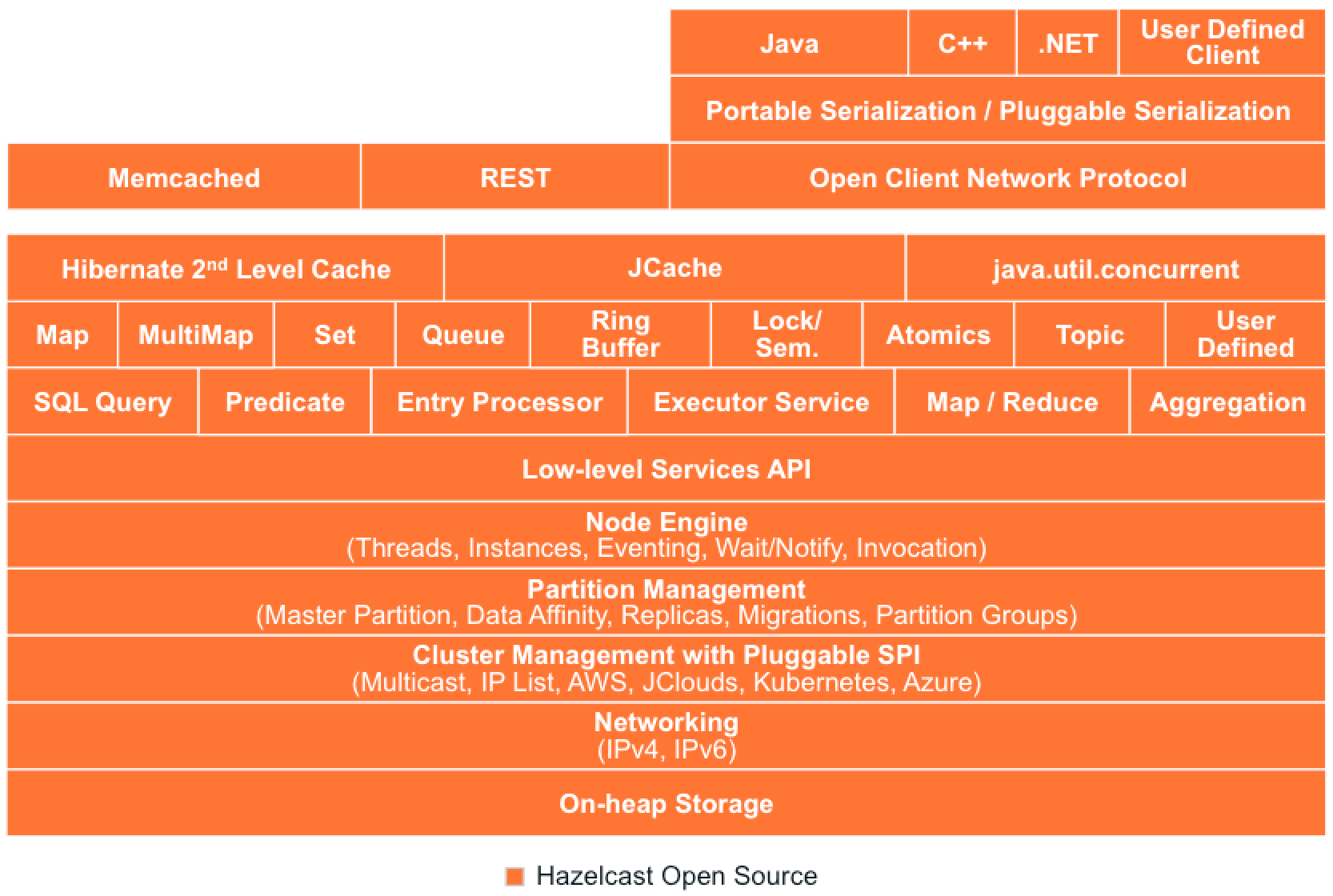
Netty 

Figure

Netty is an asynchronous event-driven network application framework which provides support for different server and client protocols. Figure 1 shows the different layers of the Netty architecture. The red box is what is being used in the project. In the context of this project it is used to provide a RESTful interface and websocket interactions with the client.

### Hazelcast

Hazelcast is a multi-use distributed computing system classified as an in-memory data grid. It provides scale-out computing, resilience, a good programming model, dynamic scalability, and an elastic main memory.



Figure

In context of the project, I am using it to provide a distributed version of a queue, cluster capabilities, and topic (Publish/subscribe pattern).

### Groovy/ Java

Groovy and Java are computer programming languages which provide ability to develop executing code. Groovy and Java was used during the development of the project.

### Gradle

It is a build and dependency management system written in groovy. It is used to compile code into an executable jar file and gather all the libraries dependencies.

### Node.js

Node.js is an event-drive, non-blocking I/O server using Chrome’s V8 JavaScript engine. It is used to provide a file server used in the demonstration portion of the project.

## Front-end Focused Technology

### AngularJS

AngularJS is a front-end framework for developing web application that consumes rest interfaces. It is used for BariumUI.

### Grunt

Grunt is JavaScript task runner used to automate repetitive task like building, unit testing, and medications of code. In context of the project it is used to build the front-code to a deployable zip. The zip is then uncompressed into the Barium’s static web directory.

# Implementation

The implementation of the Owner/Worker framework can be described as two different sub-projects called Barium and BariumUI.

## Barium

Barium contains all the logic for the Owner, Worker, queuing of tasks, WebSockets communication, and REST Interface. Barium was implemented with all the technologies in the “Backend Focused Technology” section.

### REST Interface

The REST Interface of Barium allows developers to communicate with the backend using HTTP. There are various REST endpoints that provide information of how tasks progressing over time. There are endpoints to the backend service which are called webstore, taskstore, status, and queue. The webstore endpoint serves a JSON Object of a map structured object that the developer has access to modify. The taskstore endpoint serves a JSON object with the history of all the tasks that have been scheduled, took by a worker, and finished by a worker. This endpoint allows developer to be able to trace how a task progresses and have access to the full payload of task information. The status endpoint serves a JSON object with some statistics and node information. The queue endpoint serves a JSON object with the queue size, status of overall progress in percentage of completion, tasks that are currently running on the worker nodes, and simplified version of task history.

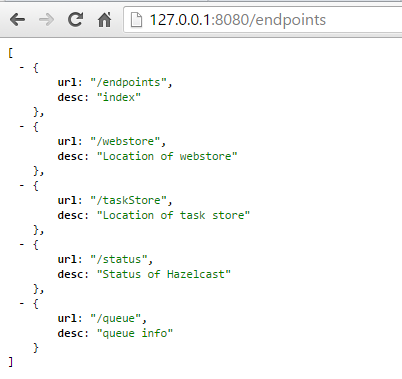


Figure – This screenshot shows all of the REST API endpoints made for the users to interact with



Figure – This screenshot shows the status endpoint

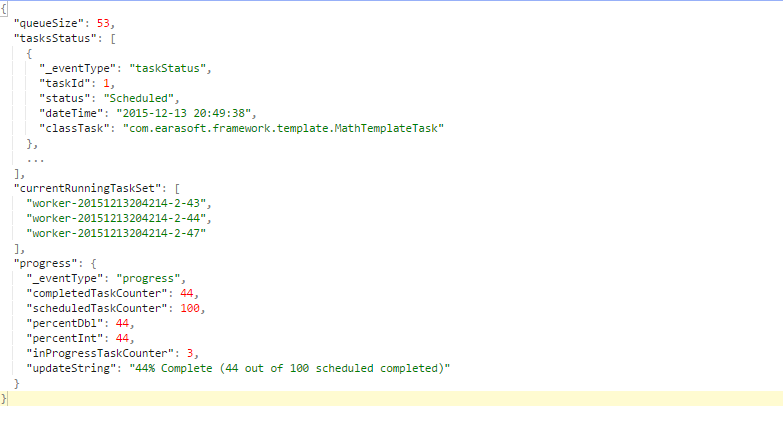


Figure – This screenshot show the queue endpoint



Figure – This screenshot shows the taskStore endpoint.

### Message Formats

This section describes the message formats used for communication between the nodes of the framework and the WebSocket consumers.

### WebSocket JSON Message Format

This message format is used for communicating with WebSocket clients. The message is required to have the flowing fields: eventType, and eventObj. The system currently has two event types.

|  |
| --- |
| {  "\_eventType": "progress",  "completedTaskCounter": 9,  "scheduledTaskCounter": 81,  "percentDbl": 11.11111111,  "percentInt": 11,  "inProgressTaskCounter": 1,  "updateString": "11% Complete (9 out of 81 scheduled completed)"  } |

The above JSON message is a sample message that was sent to the WebSocket client from the owner stating the progress of job.

### WorkerEvent Topic JSON Message Format

The JSON below illustrates a sample message format for the backend pub/sub system between the owner and the workers. Each Message has the following mandatory fields: EventType, NodeId, taskContext, TaskClass, ThreadName, Owner, and MessageTimestamp.

|  |
| --- |
| {  "workerUuid": "ee4b5cae-19c9-425e-9007-1bf002e02209",  "threadName": "worker-pool-graph-executer-7",  "results": {  "success": false,  "exception": "Exception String" ,  "taskStartDate": "2015-04-17 18:02:34",  "taskTookMs": 2525733,  "taskTookMinutes": 1.09555,  "taskEndedDate": "2015-04-17 18:44:39"  },  "taskClass": "com.comsort.personify.loading.customermasterdata.CMDLoadingTask",  "taskContext": {  "threadId": 54,  "count": 10000,  "maxOrgId": 7173584,  "type": "getOrg",  "projectList": [  "RespiratorySyncytialVirus2014"  ],  "loadDate": "20150417",  "cassandraHostname": "lctcvt2004",  "\_id": 81,  "\_failureCount": 1,  },  "hasResults": true,  "eventType": "FinishedTask“  } |

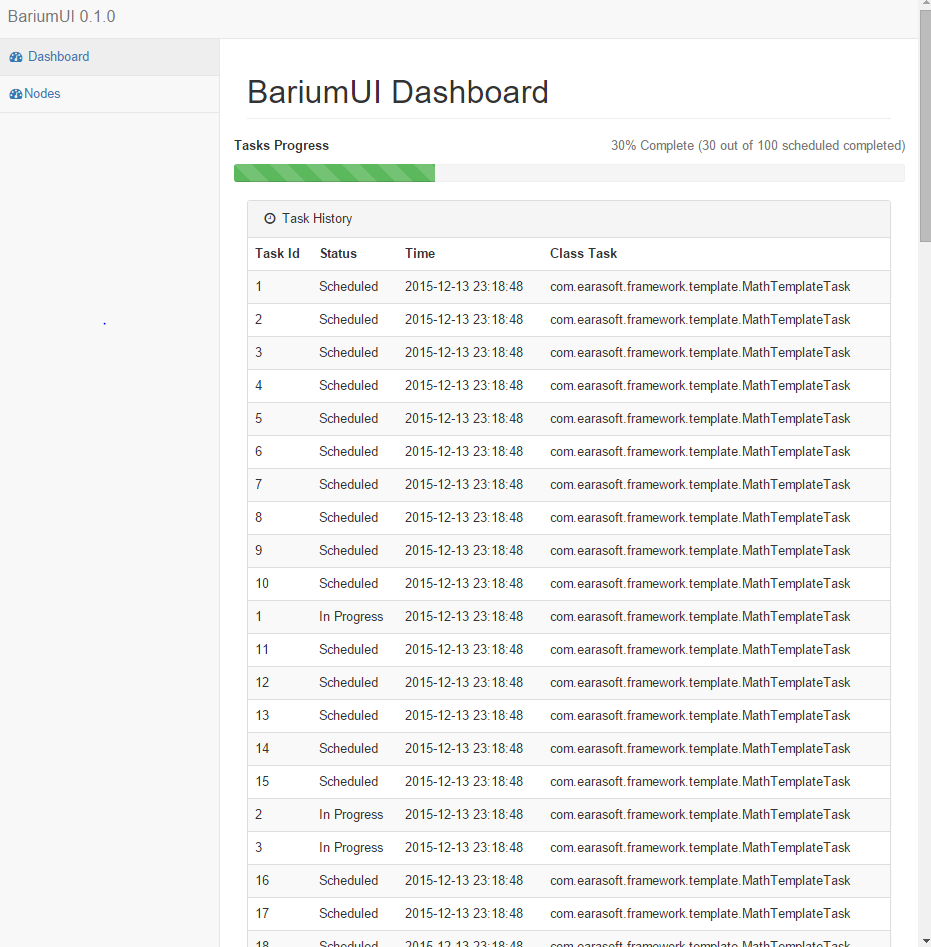
The above JSON message is a sample message that was sent to the Owner from a worker via WorkerEvents Topic.

## BariumUI

BariumUI is the front-end web application that consumes the REST API and WebSockets connections of Barium. The front-end was implemented using AngularJS which consume Barium’s REST Interface. BariumUI was implemented with all the technologies in the “Front-end Focused Technology” section. Due to the limitation of the time to do the project many features and views of the data were cut.

### Dashboard

The dashboard provides an interface to monitor the progress of the tasks. It has a progress bar which is updated in real-time using WebSockets from the backend service. It also has a table which shows a more detailed report on the status of the tasks.



Figure

## File Server

The file server was implemented using Node.js and the Express web application framework to provide a simple http interface to file system and ability to upload large files. It uses the serve-index middleware Node.js package for serving pages that contain directory listings for the data directory. It uses multer middleware Node.js package for handling ‘multipart/form-data’ which allows user to upload files into the data directory of file server. The file server can used to share files between all of the worker nodes.

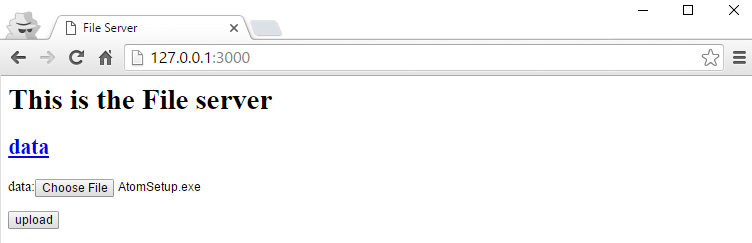


Figure – This screenshot shows the index page of the file server. This page will allow user to upload file to data directory.

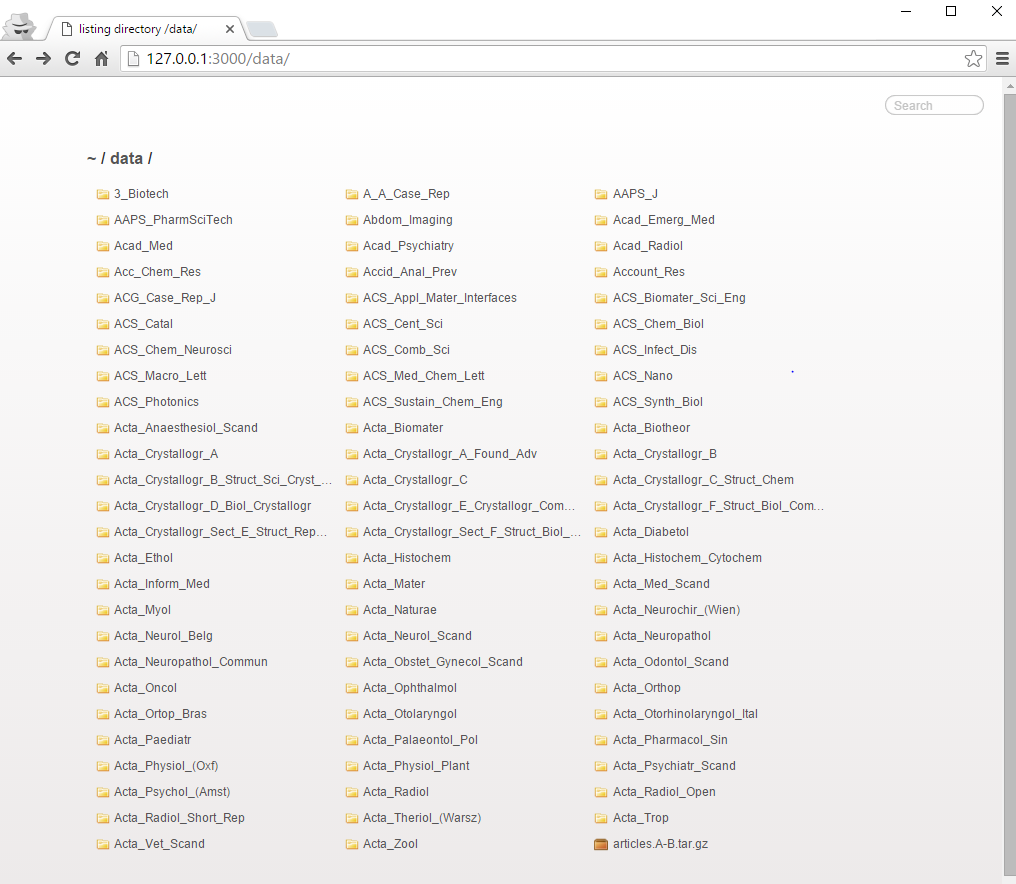


Figure – This screenshot shows the file server directory listing page for the data directory

# Development Process

Development was done in an iterative way inspired by Lean StartUp method [17]. It describes the concept of starting with ideas to build a product in a way that is iterative. The idea for the Owner/Worker framework began by coming up with a way to reduce the time to processing jobs by using spare idle machines. I had the idea of having some type of worker node and one central place will tasks could be scheduled and picked up by a worker. The owner should wait till all tasks are finished by monitoring on the status. Incrementally it grew to include many features and have two sub-projects. Most of the software development occurred in Eclipse’s integrated development environment.

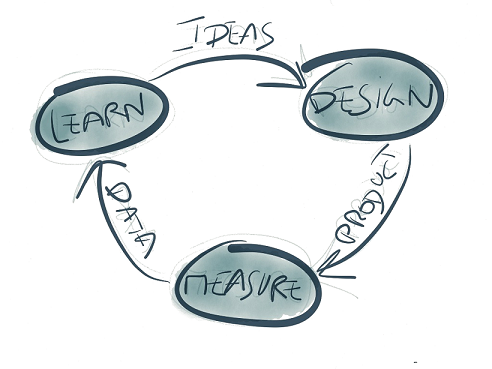


Figure – The diagram shows the Idea->Design->Product->Measure->Data->Learn cycle.

# Issues during development

Trying to figure out which features would be implemented to meet requirements during length of time of the project was a challenge. There were networking issues, configuration issues, serialization issues for the messaging backend, and a learning curve for various technologies.

# Future Improvements

There are many improvements that could make usability, reliability, and adoption of the framework to grow.

## Owner/Worker Framework Backend Improvements

This section describes the improvements specific to the Owner/Worker framework’s backend service.

### Distributed class loading between different nodes

Currently developers need to compile and distributed the main executable JAR file, dependencies, and any configuration file associated to a Job to every node manually. The Owner schedules a task with the class path of the task it needs to find it in the current running JVM. If the task’s class path is not in the running JVM it will not be able to dynamically load and make an instance of the class. By distributing the executable jar manually before you start the JVM mean that that task’s class will be in the class path. It would be beneficially to automate the process of distributing the task with the dependencies to make deployment and ease to use the framework. By automating the process that means that the developer will not have to compile the code and distribute code to every node for simple code changes.

### Isolation of the Task Execution

Currently a worker executes all tasks under a thread pool in the same JVM. It is possible for a task to crash JVM with an out of memory error and any other issues causing the JVM to shut down. Separating the execution of the task into its own JVM would increase the stability of the system. This would isolate the task from affecting other task that is also running in the same worker node since it would not shutdown the other task’s JVM. The worker’s JVM would also be safe if any task’s JVM crash and have the ability to publish that the task failed for the owner.

### Worker Failure Case

There is no logic that will handle the failure or loss of a worker node in the system. A failure can occur if the task crashes the JVM. The loss of the worker node can occur if somebody kills the process or a machine has a hardware failure. When the worker node gets a task it sends a message to the owner saying that task is in progress. If a worker node dies, it does not send anymore messages regarding task that was running on that node. From the owner point of view, the task is in progress but will never receive that the task will finish. This makes it possible for a job to never finish. It would be beneficial to add logic to handle the worker failing unexpectedly.

### Historical view of jobs

All the results of what happens in the Owner/Worker framework stay in RAM without the data being persisted to disk. The means that when the Owner’s process shutdown all the history is lost. It would be helpful to provide a way for the results to be saved into a database so that it could be retrieved later. It would require development to the backend and front-end parts of the code.

## Owner/Worker Framework Front-end Improvements

This section describes the improvements specific to the Owner/Worker framework’s front-end service.

### Keep developing and improving Barium UI

BariumUI should be developed more to provide better interaction with Barium and users. These improvements should cover more of the of the framework’s backend service endpoints. One of these improvements would be to have an interface to view historical information of the jobs.

RAML

RAML stands for RESTful API modeling language which makes it easy to manage the entire API lifecycle. It lets you define resources and REST endpoints actions. It allows designing the API by using a human readable like document based on YAML which describes everything in a concise matter. There are tools which help you build servers and client application by generating code for it. The tools are supported by multiple programming languages and systems such as Windows, Linux, IOs, Java, .net, PHP, Ruby, Node.js, and Go. You can also use it for testing, documentation creation, and sharing API with people. It has a sizable open source community in building an ecosystem to make it more usable. This would improve the project by having a formalized API for the Restful clients and developers trying to develop. There is a saying that an API is as good as the Documentation.

## Better Software Testing

I believe for a system to be maintainable and less error prone to bugs in the long term is to have a testable modular system. Each module is design to do one function, and do that function really well. You should be able to test that module for unexpected behavior and make sure it behaves without error with different modules. Modules can be a collection of classes or just one class which has one specific responsibility. By having a separation of concern for each component of the system it will help with having a better design of code. Unit and Integration Tests are great ways to automate testing of code.

Test Plans can help with coming up with tests that meet the requirements of the software.

## Major Code Refactor

I began by writing the code trying to quickly implement everything for prototyping the owner/worker framework. I started by simply creating two classes representing the owner and worker with Hazelcast for communication between them. I then separated the messaging aspect of the system into its own class for modularity. During this time I would to go to open source project on Github to view the source code to learn the best practice of programming and get idea on how to implement things better. Elasticsearch and Cassandra open source projects has help me to get some patterns for programming that I believe have good software engineering practices. I saw that Elasticsearch was using dependency injection based on Google’s Guice library. I became interested in learning about dependency inject and dependency inversion principle. They both refer on techniques for decoupling the different part of system. I came across SOLID concept which helps software developers build a maintainable and extendable system which could be combined with test-driven mythologies. Decoupling code could help developers reuse same code in multiple places.

I started to organize the code using Guice library to better decouple each component of the system. I organized it into 3 different components with static utility code which are common for more than one system. Owner, Worker, and Messaging make up the 3 different components. I would like to fully structure the code to meet the new ideas to make system more maintainable. At first I started to use Jetty for the REST interface. During the project, real-time notification and monitoring became a requirement. I thought of implementing WebSockets as the solution to communicate with the front-end. Jetty did not support WebSockets as easy as Netty. I found good example looking at the Netty sample source to implement both REST Interfaces and WebSockets. I replace Jetty with Netty in the source code but did not follow best practices.

## Fully feature Website

I believe that a fully feature website for the owner/worker framework could be beneficial. A website is a good way to capture attention to make people aware of the project and more adoptable for developers to use it. The website would have explanation of how the owner/worker framework, with a good documentation on how to develop using it. The project is currently on hosted on Github as an open source which allows developers commit code. A goal would be to start catching enough attention to help build a community of developers and an ecosystem of tools.

## Multi-Broker Support

Multi-broker support means to the ability to use different messaging systems for communication in a plugin fashion. Currently the owner/worker framework use Hazelcast for communication using a queue and publish/subscribe models. It would be beneficially to be able to use more scalable system such a Kafka. Kafka is a high-throughput, distributed, publish-subscribe messaging system.

# Lessons Learned

I have learned a lot about distributed computing and being a more well-rounded software engineer.

I believe there is always going to be a more efficient and elegant way to do software engineering which results in software being more reliable and performant. This project was a great opportunity to learn and expands my skills in different technologies and tools. I learned that there must be a great balance between designing perfect system and a working product. You could design a perfect system but not implemented a working system within a time of a deadline. You could develop a working system in a short time with bad software design smells which results software not being maintainable in the long term. Software design and code smells is referred to bad software practices such as having duplicated code, large class, dead code, inconsistent names, and other error prone code practices. I believe designing a performant and reliable system, you must have experience in actually taking the time to code which might not work well first time which gives you knowledge. As time passes and with determination you pick up better software practices, design patterns, and understand on implementing less error software using patterns that has better performance. The goal is to build a minimum viable product to meet the needs of the business goals. We live in a world in which business needs supply the demand of software production. The balance comes down to building software that is maintainable, performant, and has expected behavior in a finite time of development.

Before I start to architect a new system, I feel that doing research, looking at technology that currently exist, following technology trends, reading academic papers, and exposing yourself to new ideas and ways of doing different algorithms could be beneficial. I have grown to appreciate open source projects and academic papers more. I believe that the merging and evolution of computer science ideas help to create different technologies. A good example of this happening is Cassandra. Cassandra has ideas of the academic papers that Amazon and Google published. Google published a paper on Big Table which is a distributed multi-dimensional sorted map. Amazon published a paper on Dynamo which is highly available key-value storage solution. Cassandra is a scalable and highly available database with rich data model database. After learning to build one system you get ideas how to create next system better. Implementing conceptual ideas into source code and algorithms is important.

# Conclusion

The purpose of this project was to build a distributed, scalable and reliable task management system. Several improvements as described in the “Future Improvements“ section of this paper were cut from the final deliverable that would have significantly enhanced the final product due to time. This project was a great opportunity to learn new technologies that I would have not normally been able to look at. I believe that I have grown and improve my skills as a software engineer while developing the framework.

# Appendix

## Proposal

### Project Description/Abstract:

In this project, I will implement a Distributed Tasks Queue System in Groovy/Java that will be able to scale horizontally. It will provide fault tolerant to task failures and worker node failures. The system will be comprised of two main nodes types (owner and worker) and a messaging backend for node communication. The “owner” node will be responsible for task definitions to put in the queue and listening to the results of the computation of the workers nodes. The worker nodes will be responsible for getting task from the queue, computing the task, and then sending the task results back to the owner node. The main motivation is being able to use computational power of many machines to be able to complete a task in less time. An example use case is to be able to use the computational power of 10 machines to be able to load and process data into a database.

### Project Topics to be studied:

* Agile Methodologies
* REST Interface
* AngularJS
* Web services
* JavaScript
* Bootstrap
* Distributed Computing
* Groovy
* Java

### Project Objectives:

* Gain experience in distributed computing
* Developing a system that is able to distributed computing power of a task
* Research about distributed file systems and distributed databases
* Research mechanism for distributed fault tolerance

### Deliverables:

* Web front-end with Distributed Tasks Queue System
* Installation, Usage, and Tutorial Manual
* Deployment Scripts
* Code that follows the Groovy and Java Coding Standards
* Compressed Source Code
* Application Design Diagram (Use case, Architecture)
* Demo Application

### Anticipated Project Timeline:

|  |  |
| --- | --- |
| Gather requirements and create user stories | Aug 1, 2015 – Aug 14, 2015 |
| Work on user stories | Aug 14, 2015 – Dec 1, 2015 |
| Meet with advisor at least once a month | Aug 14, 2015 – Dec 1, 2015 |
| Develop web service interface | Each user story |
| Development and testing | Each user story |
| Prepare presentation and report | Dec 1, 2015 – Dec 9 , 2015 |
| End of Semester | Dec 16 , 2015 |

### Reading/References List:

* The Principles of Object-Oriented JavaScript, Nicholas C. Zakas, No Starch Press, 2014
* JavaScript Patterns, Stoyan Stefanov, O’Reilly, 2010
* <https://angularjs.org/>
* Sparkjava.com
* <http://www.celeryproject.org/>
* <https://docs.oracle.com/cd/A97630_01/server.920/a96521/jobq.htm>
* <http://www.fullstackpython.com/task-queues.html>
* <http://quartz-scheduler.org/>
* <http://disco.readthedocs.org/en/latest/howto/pipeline.html#pipeline>
* <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>

### Completed Graduate Courses:

|  |  |  |  |
| --- | --- | --- | --- |
| ***Number*** | ***Course Name*** | ***Semester*** | ***Grade*** |
| COSC *600* | ADVANCED DATA STRUCTURES | Fall 2012 | A |
| COSC *612* | SOFTWARE ENGINEERING I | Fall 2012 | A- |
| COSC 650 | COMPUTER NETWORKS | Fall 2012 | B+ |
| COSC 603 | SOFTWARE TEST & MAINTENCE | Spring 2013 | A- |
| COSC 716 | OBJECT-ORIENTED METHODOLOGY | Spring 2013 | A- |
| COSC 601 | SOFTWARE REQ ENGINEERING | Fall 2013 | B+ |
| COSC 617 | ADVANCED WEB DEVELOPMENT | Fall 2013 | A- |
| COSC 519 | OPERATING SYSTEMS PRINCIPLES | Spring 2014 | B |
| COSC 611 | COMPUTER SIMULATION | Spring 2014 | B |
| COSC 578 | DATABASE MANAGEMENT SYSTEMS I | Fall 2014 | B |

### Projects Done in Other Courses:

|  |  |
| --- | --- |
| **Course Number** | Project Description |
| COSC 650  Computer Networks | * Client Server Chat Application using Encryption * Learned how to use wireshark and encryption for network traffic for TCP and UDP traffic * <https://github.com/earasoft/COSC650_NetworkProject1> |
| *COSC 716*  OO Design | * Restaurant Simulation using multiple design patterns (Command, Iterators, Factory, Singleton) and Swing Programming * <https://github.com/earasoft/OO_Project> |
| COSC 603  Software Testing and Maintenance | * Learning how to test large project and software best practices * <https://github.com/mcomp2010-school/cosc603-manny-jane-bruno> |
| COSC 578  Database Mgr Sys | * Created a Timesheet management system with Python/Flask and Java/Swing * <https://github.com/earasoft/COSC578DatabaseProject> * <https://github.com/earasoft/swing-database-learning1> |
| COSC 617  AdvWebDev | * Created a Developer’s Forum Website * <https://github.com/earasoft/AdvWebDev2013Project> |

# References

1. <https://hazelcast.com/>
2. <https://angularjs.org/>
3. <http://yeoman.io/codelab/setup.html>
4. <https://git-scm.com/>
5. <https://www.github.com>
6. <http://www.csun.edu/~shan/comp696-698/Resources/Thesis-Outline-Guide-rev1.pdf>
7. <http://www.easterbrook.ca/steve/2010/01/how-to-write-a-scientific-abstract-in-six-easy-steps/>
8. <http://oanasagile.blogspot.com/2012/11/test-driven-business-featuring-lean.html>
9. <http://netty.io/>
10. <http://sd.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first/>
11. <http://blog.codinghorror.com/code-smells/>
12. <http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>
13. <http://www.cs.washington.edu/events/colloquia/search/details?id=437>
14. <http://kafka.apache.org/>
15. <http://gruntjs.com/>
16. <https://nodejs.org/en/>
17. <http://theleanstartup.com/principles>