Design and Planning Document

<Remon>
Design and Planning Document
11/28/2014, version 1.2

Document Revision History

Rev. 1.0 2014-10-31 - initial version
Rev. 1.1 2014-11-14 - update document to fit on new architecture
Rev. 1.2 2014-11-28 - update implementation plan and testing plan section
Rev. 1.3 2014-12-12 -

##Changes

System Architecture

Remon is an application monitor specialized for REEF(Retainable Evaluator Execution Framework). Remon collects the useful information from the running REEF application and provides a handy way to investigate or control the applications. Remon can reduce the effort to inspect the logs to find some metrics or launch a lot of commands during execution.
The system consists of two main components Collector and Monitor. These components work as a server-client model.

Collector

Collector resides at the REEF-side. It collects the metrics that user want to trace. REEF Evaluators periodically sends the data and REEF Driver receives the data and sends the data to the Remon Monitor. To provide comfortable way for User to add new metric easily, it should provide a simple API to register new metric and encapsulate to help user from worrying about the implementation details.

Monitor

The main responsibility of Monitor is to store data and show them to User. For the metrics registered by User, Monitor should provide a enjoyable visualization. For the usability, it is very important to choose most appropriate representation (type of graph,

...), arrange the data into some group or to have hierarchy between graphs, and so on. There will be a lot of effort to make user feel comfortable when using Remon.

One more feature of Monitor is providing a GUI to interact with REEF application. Of course many programmers are accustomed to using Command Line Interface, however, it is little bit burdensome to write long commands all the time to handle the application. In this way Monitor leverages the interaction between user and REEF application.

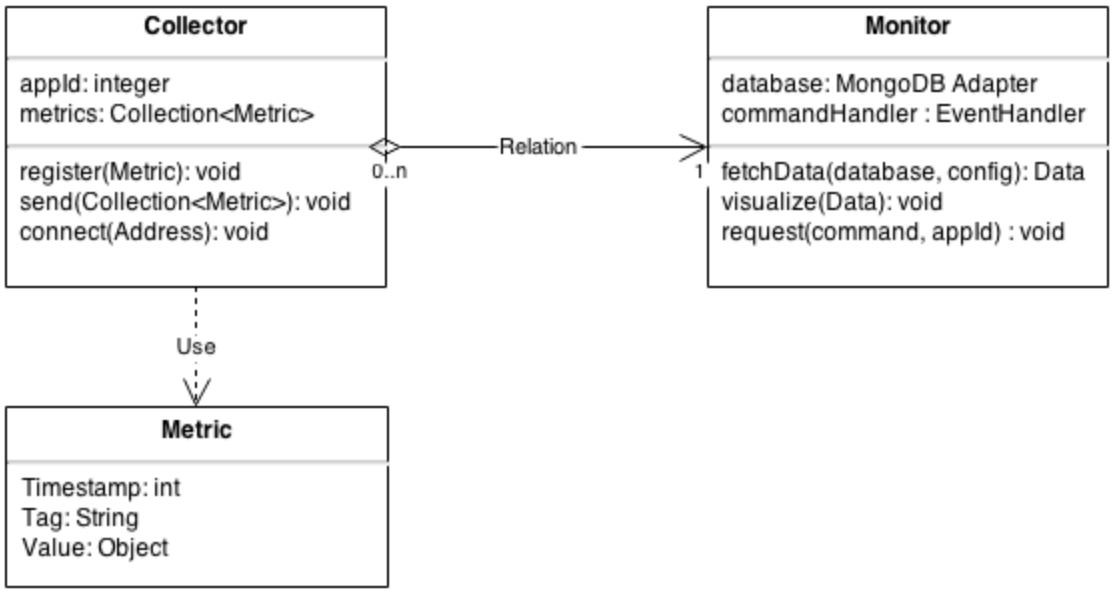


Figure 1. Class diagram for Remon components

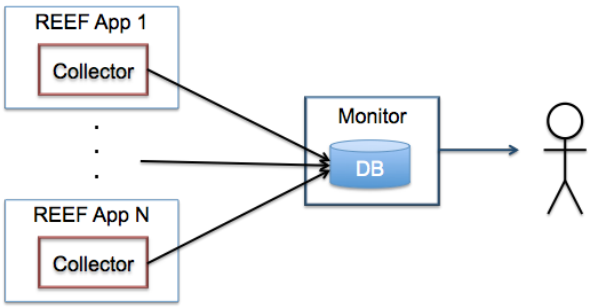


Figure 2(a). Data visualization

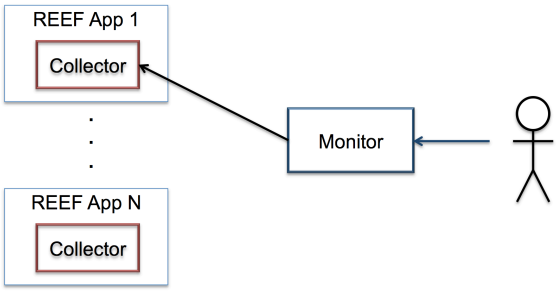


Figure 2(b). Launch commands

Design Details

In this section, we describe each components and its interfaces with more details. Our architecture is designed considering further extension such as multi-client to multi-server. ReMon is currently implemented specifically for REEF. But in later version, it may perform as a general platform monitoring tool.
Here we describe important points one by one.

###Collector (REEF-side)
ReMon provides libraries to driver of REEF application to log data that you want to monitor. Its main feature is to gather all history data and send it to Monitor. Internally, each evaluator's data is gathered by Heartbeat, to driver and driver sends it via websocket, to Monitor. Heartbeat is generally used in where gathering place of logging data such as driver.

###Monitor (WEB-side)
It is a main front-end component of ReMon. We deliver information to users with various representations such as raw texts and tables and graphs, for easy comprehension. It shows current running application status and also it provides dynamic page views with AJAX pediodically refreshing recent data. Also user can customize its view on their demand. Also it handles user request that control running job tasks such as 'restart task' or 'quit application'. It takes user's command and redirects it to REEF. User can control it with a visual button or command line interface via web.

```
{
    "app_id": string,
    "metrics": [{
        "source_id": string,
        "tag": string,
        "time": timestamp,
        "value": double,
    }]
}
```

<Table1. Main Internal API for ReMon>

###WebSocket
We choose a websocket as a commucation mean between Collector and Monitor. We considered RESTful API for communication at the first time, but we do not actually match the 'request/response' model. Moreover, the components communicate many times with short time

interval, so we decide to use WebSocket to preserve the connection between the components. Further, as the connection established already, they just can send the data without making a new connection.

###Asynchronous Web-server
As a Monitor server, we decided to use Tornado webserver. Because Tornado webserver supports asynchronous network I/O, it can establish a lot of connections at the same time, so we can make sure to provide a scalable service.

###HeartBeat
It is a mechanism that is originally existing in REEF. So we can use it as a channel collecting distributed information. It gives us a benefit that no additional overhead. Leveraging it makes our works efficient.

Implementation Plan

###Common
(DONE)
- Setup CI(Continuous Integration) testing

(TODO)
- Monitor user behavior
- Support string log message interfaces[b1]

###Monitor
(DONE)
- Support multiple application in User Interface
- Clean up Javascript code

(TODO)
- Control the appliation via Web UI
- Support Histogram visualization
- Exception handling / security issues for monitor component
- Show current app status (running or finished)

###Collector
(DONE)
- Add ML application(K-means) for demonstration

(TODO)
- Process multiple data points in a batch

To implement [b1], we must implement additional functions on Collector and Monitor. Collector sends string log messages similar with metrics. Monitor shows these messages as text.

Testing Plan

Unit testing:

- For ReMon Collector(Java), we use JUnit
  - cover 93/105 instructions of edu.snu.cms.remon.collector package
  - cover 72/72 instructions of edu.snu.cms.remon.collector.evaluator package
- For ReMon Monitor(Tornado), we use Tornado unit testing
  - cover 88/93 statements of app.py(Tornado web server)

Functional testing:

At first, we planed to test each module with additional mocks. But with renewal of overall design, total modules are reduced to only two things and interface between them become too simple to test it seperately. And main implementation was focused on stable connection of components rather than rich functions. So we thought that there is no need to test seperate functional testing with mocks. Integration testing was more reasonable to check that modules are properly works and interacts with each other. As a result, we focused on implementing overall work pipeline and testing overall connection and interaction.

Acceptance & integration testing:

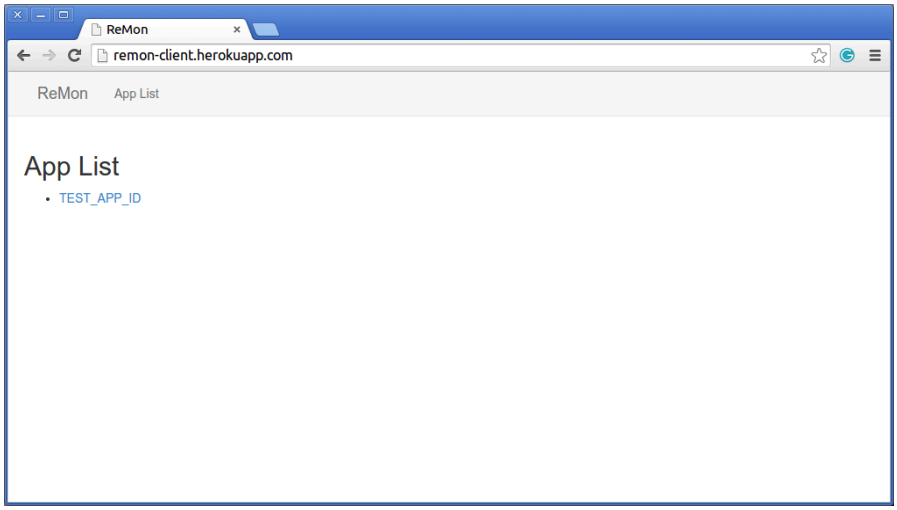
We have done this test with real-code simulation. First we deployed our Monitor(web-side) to Heroku. And we have made Collector operate with our example REEF application.

This application(named `CounterREEF`) has an evaluator continuously incrementing the counter value and consuming heap memory up to an amount of value. The memory status for evaluator is piggybacked on each heartbeat message and the driver sends the heartbeat message including the status of overall application status (# of evaluators currently) to the Monitor.

We could see the Collector and Monitor work well when combined, satisfying some of our user stories described in Requirement & Specification document.

* Show app list
* Monitor time-series data of running application with dynamic refreshing view
* Overviews overall status of REEF application
* Recognize status change of ReMon
* Connect running REEF application to ReMon Monitor

(image url : http://img.ctrlv.in/img/14/11/28/54782c898debd.png)



(image url : http://img.ctrlv.in/img/14/11/14/5465b7ab03cbe.png)

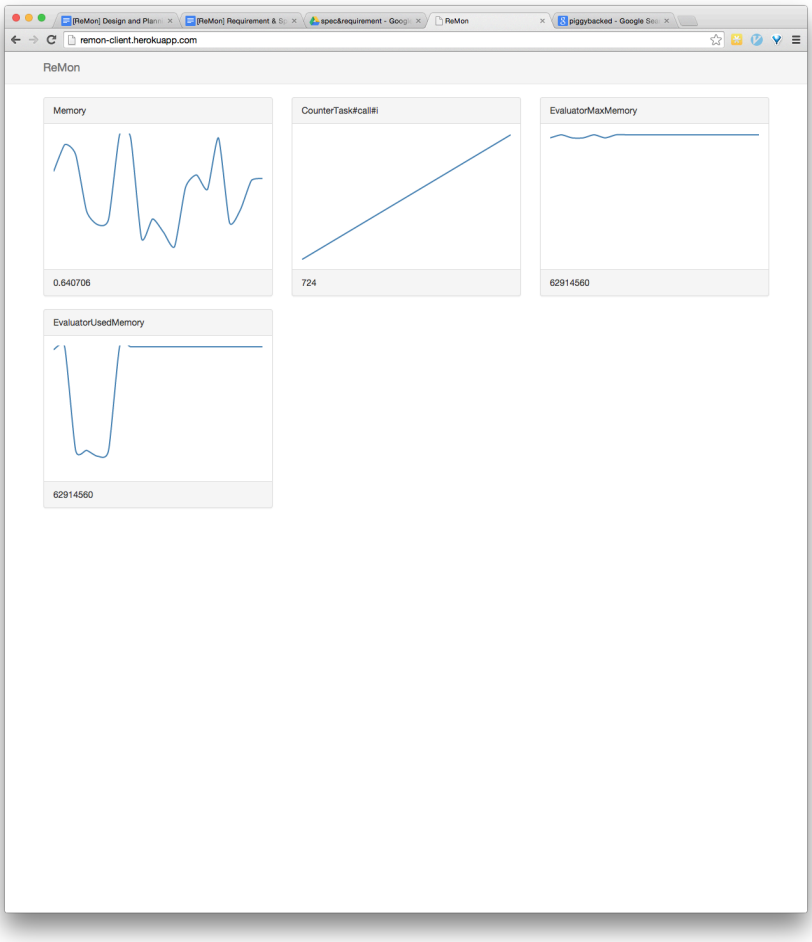
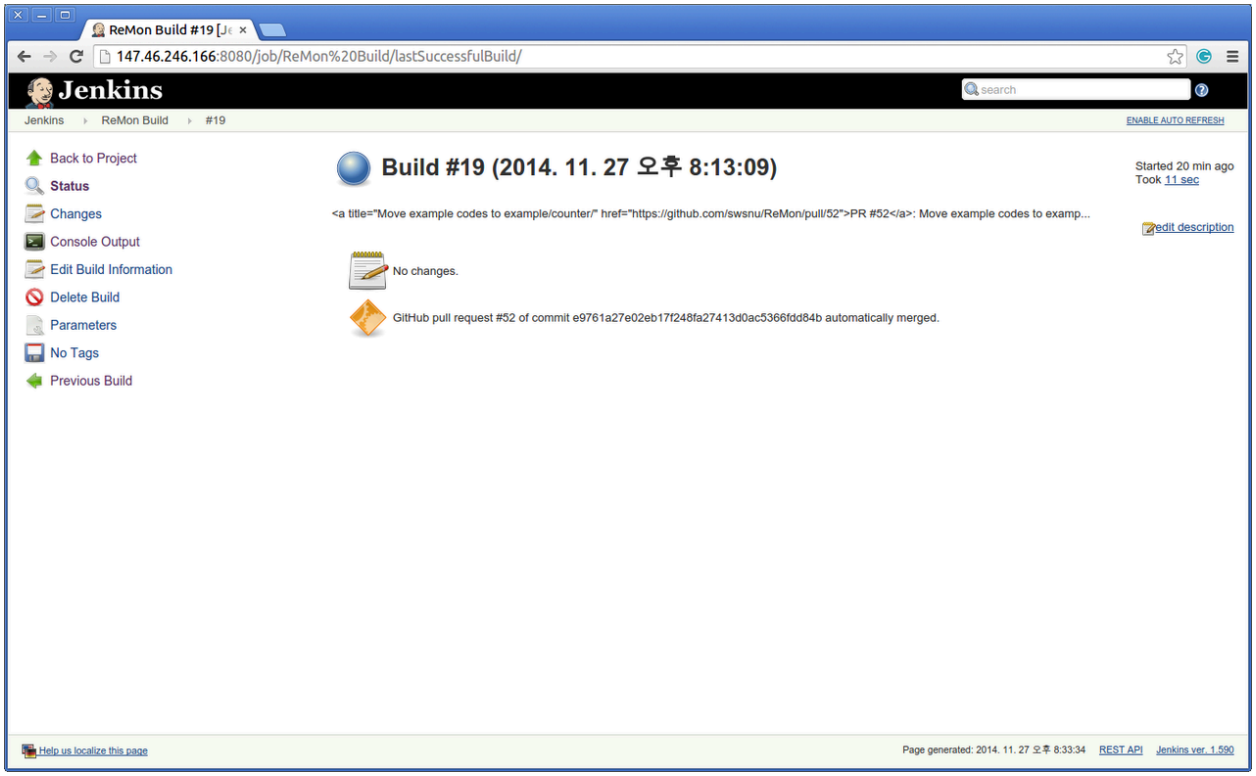


Figure. simulation of interaction between Collector and Monitor

Also, we implemented Jenkins server to setup CI(Continuous Integration) testing. We set `ReMon Build` job. It automatically executes unit testing for Collector and Monitor. If new pull request is pushed, Jenkins runs the job and show result as below.
(image url : http://img.ctrlv.in/img/14/11/28/547811e333317.png)



Reference

- [Tornado Web Server](http://www.tornadoweb.org/en/stable/)
- [REEF](http://incubator.apache.org/projects/reef.html)
- [JUnit](http://junit.org/)
- [QUnit](http://qunitjs.com/)