

Maraton

Genomics Distribution Computing System

October 22, 2015

YHGenomics

Contents

[1. System Introduction 2](#_Toc436237077)

[1.1. System Structure 2](#_Toc436237078)

[1.2. Subsystem 2](#_Toc436237079)

[1.2.1. Executor 3](#_Toc436237080)

[2. Executor Layout 3](#_Toc436237081)

[2.1. Message Handler Layout 3](#_Toc436237082)

[2.2. Task Processing Layout 4](#_Toc436237083)

[2.2.1. Executor States Chart 4](#_Toc436237084)

[2.2.2. Task Receiving 4](#_Toc436237085)

[2.2.3. Environment Checking 4](#_Toc436237086)

[2.2.4. Data downloading and reorganizing 5](#_Toc436237087)

[2.2.5. Task Processing 5](#_Toc436237088)

[2.2.6. Local Result Cache 7](#_Toc436237089)

[2.2.7. Reliable Result Delivery 7](#_Toc436237090)

[2.2.8. Exception Processing 7](#_Toc436237091)

[2.2.9. Error Handler Hub 7](#_Toc436237092)

[2.3. Utility 7](#_Toc436237093)

[2.3.1. Resource Descriptor 7](#_Toc436237094)

[2.3.2. Appraiser 9](#_Toc436237095)

[2.3.3. Analysis Helper Base structures 11](#_Toc436237096)

[3. Third-party Library 11](#_Toc436237097)

[3.1. Libcurl 12](#_Toc436237098)

[3.2. Json 12](#_Toc436237099)

# System Introduction

Maraton is a distribution computing system that major job is genomics computing and aligning on scalable nodes.

The entire system is a computing cluster consists of one master and multiple executors.

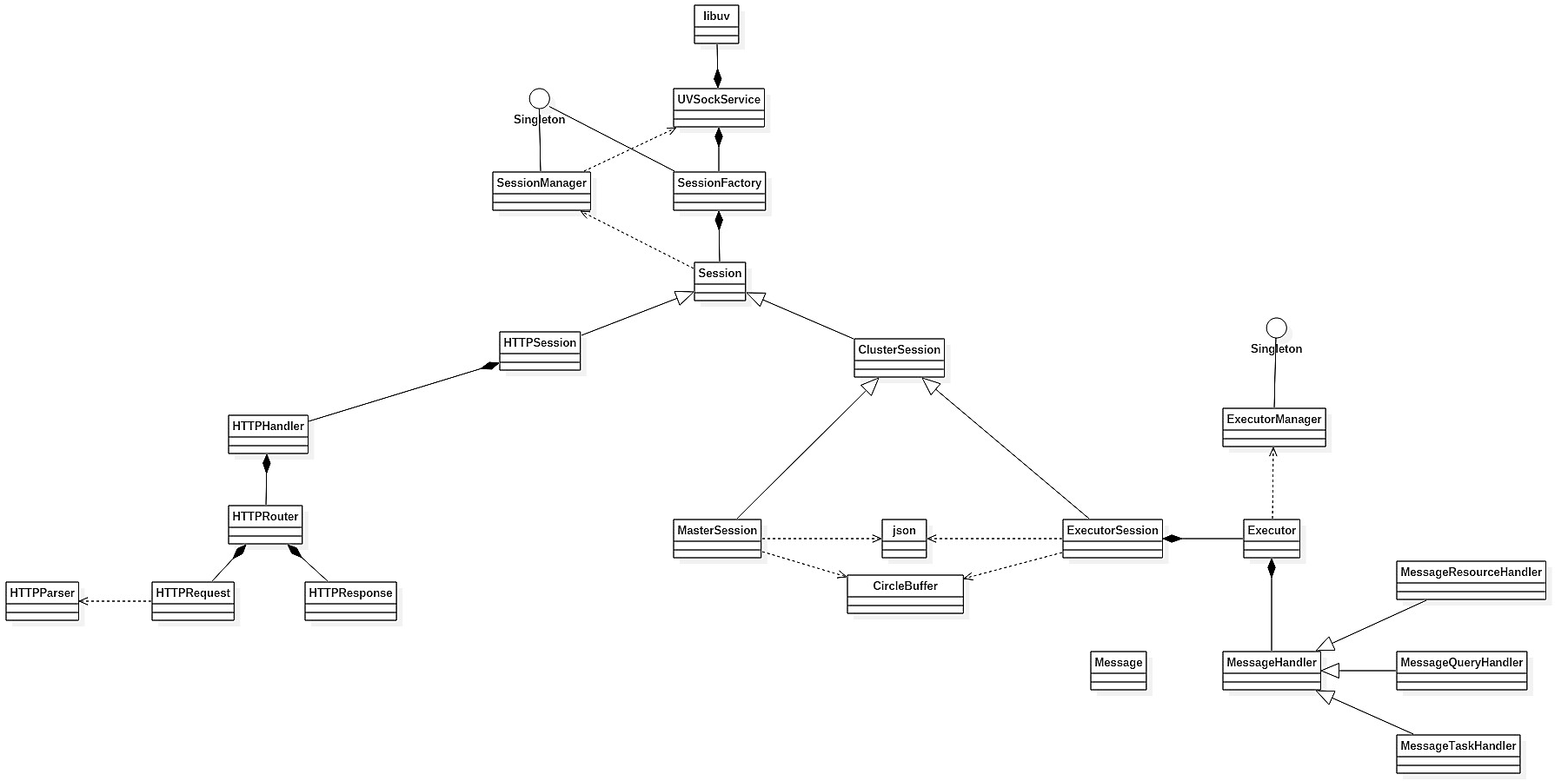
Master is a controlling and scheduling system that controls all the clients on the network using TCP/IP protocol. Also master has a high performance scheduling algorithm that can dispatch jobs to the client according to its abilities.

Executor is a computing node that may be implemented on x86 PCs, FPGAs, portable devices, routers etc. Every computable device that matches to the protocol can join the network as a computing node and be managed by the master.

A compressed JSON string which transferred over TCP/IP with special header is used as the protocol in considering of the kinds of device type may in the system.

Networks in Maraton consist of two different network. The first is the controlling network that exchange the command messages between the master and its executors. The second is the data transferring network which has a larger bandwidth need than the controlling network since the enormous size of the genome data. It is used for manipulating transfer of the large genome data file.

## System Structure



## Subsystem

There are three subsystems in the Maraton. Framework is the base of the system. Master mainly manage the tasks and executors. Executor is responsible for the specific bio-information data processing.

### Executor

An executor is a processing node that can be managed by the Master, and do the data processing job.

It can be as classical as a software program processing the data with CPU, or with a GPU acceleration, or even with FPGA.

Communication between master and slave is over TCP socket.

An executor should meet the following basic requirements:

1. Be connected to Master when receiving a command, and reply the properties of itself to Master.
2. Reports a heartbeat message to Master periodically.
3. Receives a task, and report it will be processed or not.
4. Check the environment before getting start with a task.
5. Load the pre-load data.
6. Makes the computing work done and then cache the results
7. Delivery the report of the results reliably.
8. Reports any exception that happens and shut down or restart the processing.

And further requirements are:

1. Evaluates the quality of data and result with an error-model based on statistic.
2. Discoveries kinds of faults.
3. A fault-tolerant processing.

# Executor Layout

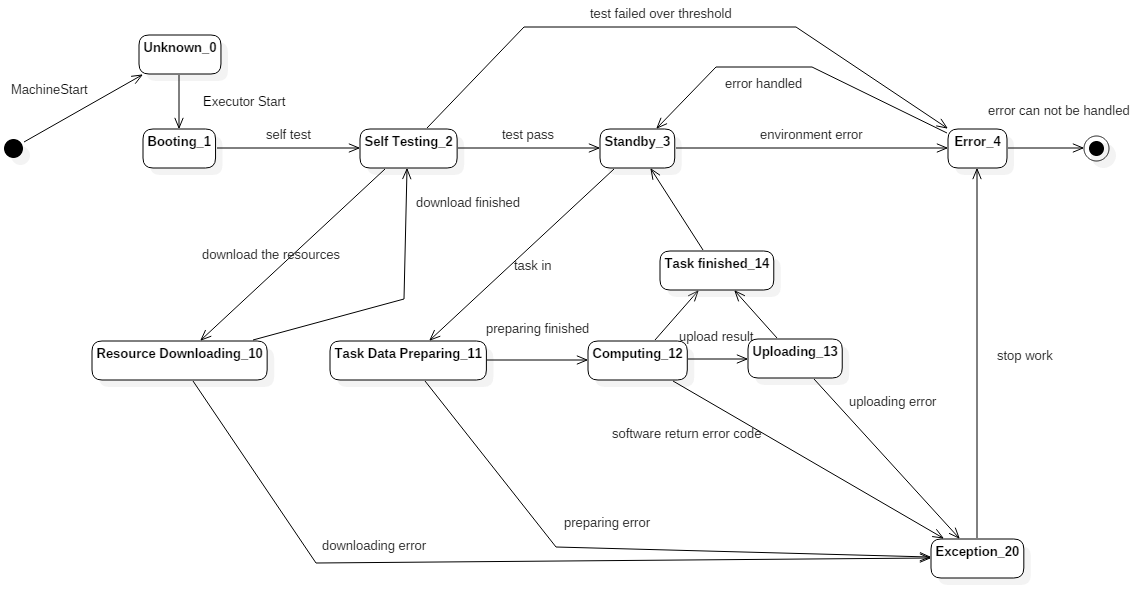
## Message Handler Layout

A high level message hub is controlling the messages received by the executor.

And for the handling of each message is called in that hub based on a predefined flags.

## Task Processing Layout

### Executor States Chart



An executor can be considered as a FSM (Finite State Machine), the whole life cycle of the executor can be illuminated in a states chart.

The executor should always try to report the state to the master whenever a change happened or under an order from the master.

These states are the states of the executor itself which means the states of a task is not considering in this chart.

Any type of exception will cause the executor to an error handle hub which will always try to recover the executor to the state of Standby, in most cases this means a drop of task, but there’s one special mechanism called reliable result delivery was designed for reduce meaningless data re-processing when the result delivery come to an exception. More details can be found in 7.2.7

### Task Receiving

The task delivering message contains information on a full description of what data to process, which processor should be used with what parameters.

### Environment Checking

The reference genome is given by the name.

First thing the executor do is ensure it has the reference genome. A default reference genome, for example hg19.fa, will be downloaded when an absence is checked.

Then, the executor makes sure it has the index and other files need for begin the processor, which means downloading of the processor and making the index file when needed.

### Data downloading and reorganizing

The huge data of bio-information is split into small files by the master before a task is delivered. The small files contain n reads with a line number of 4n each, and even if there’s some fragment read (with some lost information) it comes at the end of the original file and, of course, will be put in the end of last small file.

A list of the URIs of the small files with an order match the original file is put in the task deliver message.

So the first thing in a data prepare phase for the executor is to download the small files and put them together into exactly same huge file as the original one.

### Task Processing

#### Basic Computing

A processor, such as BWA is used in a black-box style.

It can be one single phase, or a whole pipe-line, this is based on what the task is requested.

#### The Analysis Helper

The tools set changes can the deployment of them can be a mass. And the version management of them to ensure all executor nodes process different data with the same tool could be a challenge.

So the analysis helper is there to handle the problem by deliver the task to a docker container.

Docker is used by REST API, and all the parameters are set by JSON.

The task is delivered with a certain docker image name, the job of executor is download the input files, pull the image down, set the parameters, create the container, start the container ,and the process will begin automatically. Then, just wait for the container exit with a exit status code, if the code is 0, upload the result file to the storage server.

To use the local repo append the following codes after the docker daemon

--insecure-registry [IP\_ADDRESS]:[PORT]

Analysis Helper does the job of the upper level, to fulfill it there are base structures needed which are explained specially in 7.3.3 .

#### The first BWAMEM image

The dockerfile for build this image is like this.

*FROM ubuntu*

*RUN apt-get update*

*RUN apt-get install -y bwa*

*RUN mkdir /input*

*RUN mkdir /output*

*RUN mkdir /ref*

*ENV t 1*

*ENV refgen ''*

*ENV reads ''*

*CMD bwa mem -t ${t} /ref/${refgen} /input/${reads} > /output/${reads}.sam*

The three volume needed be bind if a path contains the input sequence reads data called input, the path contains the output file sam/bam or other file format by the image, and the path of the reference genome.

Three environment parameters needed to be set are the thread number based on the computing ability of the executor, the name of the reference genome ( in case new version or special version come )，and the name of the sequence reads need to be processed.

#### Quality of Result

The quality of data can make difference in the quality of the result. So, when a task is finished, besides report the result of the processor, the quality of the result will be calculate based on an error model based on statistic.

#### Faults and Fault-tolerant

Base on the states chart of the executor, kinds of faults and exception can happen in the life cycle of an executor.

The basic strategy of fault-tolerant is drop the calculation turn the executor back to a standby status, which means post a repot via HTTP to master, and then eliminated any temp-data generated during the processing, such as the input sequencing FASTQ data, but in case of transmit faults such as net off, the output file such as sam/bam/vcf files need be store and wait for the net recovery.

### Local Result Cache

### Reliable Result Delivery

### Exception Processing

### Error Handler Hub

## Utility

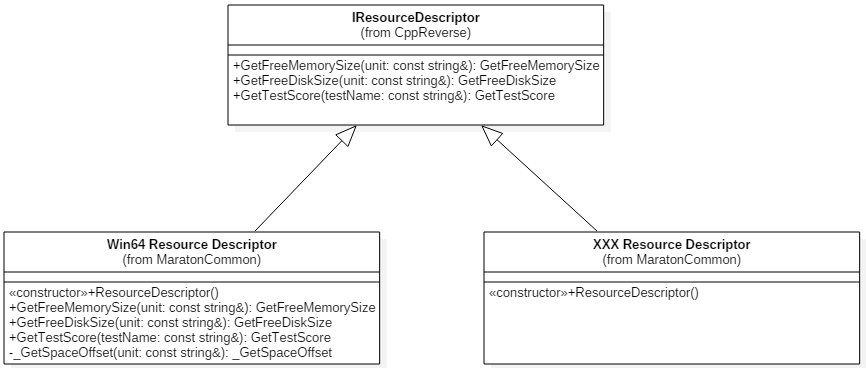
There are plenty of supports need for Executor doing its work, such as gathering the information of the processing node, handling of the messages in and out, and so on.

### Resource Descriptor

Resource Descriptor returns the measurements of the executor’s resources.

For the basic resource type which can be measure by simple factor, such as the available physical memory size, that direct values will be returned.

In a more complicated case, when a higher level abstract measurement, such as the processing ability of the executor, is needed, a black-box test returning a score is used. Such test will be high level and equally for different kinds of executor so that the return value can be comparable, for example, a processing time for same pre-load date from stage A to stage B.

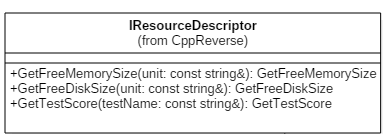


#### Basic Interface

As the description of resources changes platform to platform, in respect to the open-close principle, each platform should use their own descriptor.

So a basic interface should be designed, and a non-implement flag should be return by default.

Here, the basic interface is designed as the **IResourceDescriptor** with three virtual methods.



#### Free Memory Size

This method is defined as

virtual size\_t GetFreeMemorySize( const string& unit )

By default, this method returns -1 as a non-implement flag.

Each implement should at least with the ability to return the size of the **available physical memory** in MB.

Each implement should use the white-list strategy to handle the string **unit**, and return -1 for any unexpected value.

#### Free Disk Size

This method is defined as

virtual size\_t GetFreeDiskSize( const string& unit )

By default, this method returns -1 as a non-implement flag.

Each implement of this method should return the size of the **free disk space** size in MB.

Each implement should use the white-list strategy to handle the string **unit**, and return -1 for any unexpected value.

#### Test Score

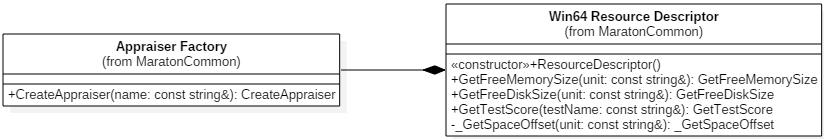
This method is defined as

virtual size\_t GetTestScore( const string& testName )

By default, this method returns -1 as a non-implement flag.

Each implement should pass a test name and let the appraiser factory can run the corresponding appraiser and return the value of the test.

#### The WIN64 Resource Descriptor



As the window APIs may not useful for the OS version in future, neither forward nor backward compatibility is supported.

**GlobalMemoryStatusEx** is used for getting the available physical memory. The **dwLength** member in the **MEMORYSTATUSEX** must be set before calling the **GlobalMemoryStatusEx** method, or the result could be over unbelievable.

**GetDiskFreeSpaceEx** is used for getting the free disk space size.

And for the getting of the test score, an **Appraiser** was used as it be told later.

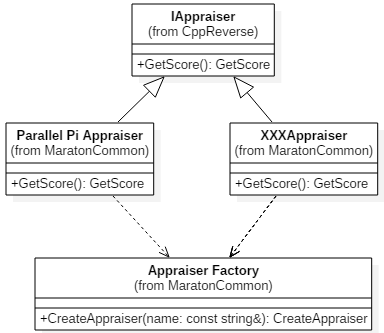
#### The linux(64bit) Resource Descriptor

### Appraiser

Technically the measurement of the processing ability of a certain processing node is hard to describe by a little numbers of low level factors, so a high level black-box test is a better idea.

Each appraiser implement a certain interface define as **IAppraiser** with a virtual method( **size\_t GetScore()** ) to run the test and give back the score( -1 by default as a non-implement flag).

A plenty of Appraisers can be created by their names with a **factory**, and more appropriate appraiser can be designed and add to the factory later.



#### Appraiser Factory

Simple factory can fulfill this task very well for now. And when the time of an Appraiser Boom coming, the configure files or reflection may be a choice.

The create method return a **unique\_ptr** of the **IAppraiser**. The raw(or naked) pointer is not recommended.

unique\_ptr<IAppraiser> CreateAppraiser( const string& name );

#### Appraisers List

A header file does the very simple management job for the plenty of appraisers. Consider it as the poor man’s configure file and mostly for getting a little tidier code in the factory implement file.

#### Parallel Pi Appraiser

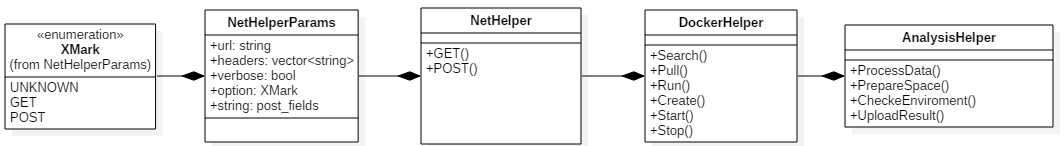
Appraise by a Parallel computing version of Pi calculation. Use **OpenMP** for the parallel processing which can make full use of available resources of the processing node.

#### Simulated100MB Appraiser

BWA MEM is used as the appraiser. To make the full use of the process ability of the executor BWA MEM is set to work in multi threads way which take as much threads as the logical processor number minus one.(Left one out for the OS.)

### Analysis Helper Base structures

To use the docker with the REST API we need a helper work as the adapter to change our parameter to a well-formatted REST API, then use the libcurl to send the POST/GET commads.



The Docker helper is used by the analysis helper, mainly for the control with the docker REST API. The function run is not directly supported by the docker REST API, so it is just put create and start in one.

The Net helper is used to send the GET and POST net command by HTTP1.1 via libcurl.

And the parameters of it is put into a class called NetHelperParams. In this way, the changes of the parameters would not force the code adjust in the method definition in Net Helper.

# Third-party Library

## Libcurl

## Json