**Design Document concerning metrics collections**

By Anthony Cali, Edde Anthony Lopez, Hongchao Yang

Version 1.0

# Table of Contents

**Introduction**

What are metrics?

What we intend to accomplish

Audience

**Subsection I. Data Collection**

Epic

**Subsection II. Prerequisites and Installations**

Development Environment

User Installation

**Subsection III. Java Native Interface (JNI)**

**Subsection IV. SQL Database**

Epic

User Stories

Error Handling

**Subsection V. Graphic User Interface (GUI)**

Epic

User Stories

**Subsection VI. Testing**

Epic

User Stories

**Subsection VII. Specifics and concerns**

**Subsection VIII. Stretch goals and improvements**

# Introduction

This project will collect metrics of all processes on a Linux system. This information is useful to system administrators and application vendors. Seeing real world performance outside a laboratory environment can be a daunting task. Small variations in system configuration or deployment can cause major differences in how software runs overall.

## What are metrics?

1. Anything logged within a system that can be used to measure software or hardware performance.
2. Quantified data within a specific machine
   1. Memory usage
   2. Compute throughput (CPU utilization)
   3. etc

## What we intend to accomplish

1. Build a metrics collection software.
2. Understand how this kind of software mirrors all real production software.
3. Experiment with design.

## Audience

System administrators may need this information in order to track down slow machine performance and memory hogs. Any owner of a personal Linux machine may be able to use this information to stop processes running on their machine.

# Subsection I. Data collection

**Epic**

Every Linux system has a \proc folder that contains information about the processes that are running on the system. Each process has a PID number that identifies it. The \proc folder has a subfolder for each PID number, and these subfolders hold files that contain information of the process that possesses its PID number. Each of these subfolders would be polled for metrics that would eventually be collected in a SQL table and the GUI.

**There were two ideas that were considered for obtaining metrics:**

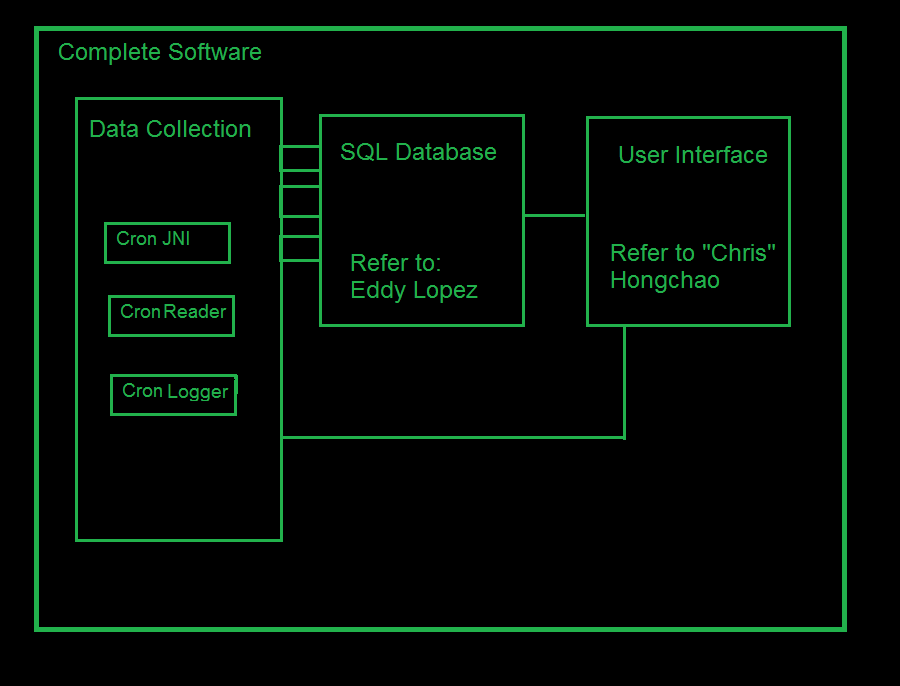
1. ~~Multithreading the data collection subsystem.~~
2. Individual Java programs for data collection [See Diagram 1].

We decided to use individual Java programs rather than multithreading because individual Java programs are easier for maintenance and have a smaller scope than multithreading. Cron also logs automatically, which would cut down the programming required for logging errors and progress, and would especially be useful for error tracking.

GNU/Linux simplifies this to a degree by outputting most of the internal system logs to the directory /proc/#PID (where #PID is the PID number of a process)

* total size (%s)- create memory map base
* file name (%n)
* file creation (%w)
* Parent PID
* All other information can be derived from the above (e.g. memory map base, memory usage, time active, etc).
* Each process may be sent to the SQL and the GUI as a process object.

Logs will only be generated upon errors and stack traces. When testing the program, command line arguments may be used, but they will be removed in the final product.



**This diagram represents an individual program approach for data collection.**

**Diagram 1**. Shows an individual program layout for the data collection code and potential dependencies. Not all dependencies shown. Not all programs shown.

# Subsection II. Prerequisites and installations

## **Dev environment**

* Maven
* Netbeans
* Javascript/HTML (GUI)
* GCC
* java via openJDK 1.7
* PostgreSQL

## **User installation**

* postgres (either by the user or the package manager).
* Run the package in the package manager.

# Subsection III. Java Native Interface (JNI)

Thus far Java User Interface (JNI) is only required for retrieving the CPU clocks into usable time. This can be achieved by importing the time.h header into the C program.

We, however, need to exercise care with integer conversion between C (little endian) and Java (big endian). It may be easier to convert from little to big endian using C functions before converting to Java.

# Subsection IV. SQL Database

## **Epic**

All SQL functions will be called in Java. The SQL tables will be manipulated by calling stored procedures. This project will be using PostGreSQL, due to our familiarity with the structure and how we interface it in Java. PostGreSQL uses relational databases, which means each row will have a fixed number of attributes, or columns. Ad hoc queries and sorting will be easier (e.g., finding the process that uses the most memory), as opposed to NoSQL and other non-relational databases.

One table will store the data of each PID currently running on the system. This table will store:

* The name of the process and its PID. The PID can be used as the primary key.
  + The PID will have a table spawning from it containing all of its children processes.
  + If a process dies and a new one is spawned, the new process make take the PID of the old process.
* The PID of its parent. This PID is used as a foreign key to reference its parent table.
* The complete command line that executes each command, which is indicated in the “cmdline” file.
* The current process working directory, found in the “cwd” file.
* The execution time on both kernel and user mode.
* The memory usage.
* The page fault counter.

The SQL table will then interface with the GUI to show processes. The SQL table will be uploaded every 15 seconds by default. A user may be able to adjust the time intervals, based on how many processes are running in the Linux system at any one time. All the SQL data will be stored in a list that will be transferred to the Java section of the program in order to be displayed in a GUI.

In addition, the SQL table will not be used again the next time the program is run. This is because whenever a new session in UNIX is run, the PID for a certain process may be different from the previous PID. The SQL table may be stored in an external folder for future reference.

## **USER STORIES**

* Store all metrics stored in the SQL table
  + PostGreSQL will be used due to our familiarity with this over all other SQL databases.
  + It is easier to call SQL functions through Java, rather than C.
  + SQL inserts will be programmed using stored procedures, in order to save lines of code.
  + This piece relies on the user story of collect metrics from each process.
  + PostGres is good for multiprogramming.
* The SQL table will need to be updated every 15 seconds by default.
  + Will need to determine how many clock ticks are in 15 seconds.
  + Will need the Java Native Interface to rely on C programs that will determine how many clock ticks in 15 seconds.
  + As a subtask, the SQL table should be purged of any PIDs that are no longer running. However, this may be done less often than the actual update interval (e.g. 4x the actual interval).
* Display each row on the SQL table to the GUI
  + The GUI will get data directly from SQL table.
    - This is more efficient than having the same Java program that writes to the SQL table also update the GUI.
* Upon program termination, SQL tables are saved to an external folder for future user reference
  + The SQL database is not used again upon the next startup of the program. Each startup of the program will create a brand new database.

PRIMARY KEY is composite key {PID, processname, machinename}. Doing this makes it better for people who want to expand this to different Linux machines.

## **ERROR HANDLING**

**Problem**: There is an error writing to the SQL table.

**Possible causes**: Concurrency error (e.g. there are two writes for the same record)- may cause a deadlock (race condition)

**Solution**: Increase the update interval or purge the database more frequently.

**Problem**: Database is unreachable by a reader.

**Solution**: Fail softly, log, and then retry in the next interval.

**Problem**: The program may write a process with a PID that already exists in the table.

**Possible** **causes**: A process has died or has been killed in the Linux machine and a new process has spawned using the same PID as the dead process. (However, we should also be aware of the name of the process).

**Solution**: This must be logged.

- A possible solution is to identify any PID in a previous update that isn’t present in the current update and then delete the entry with the PID in the table.

- If the above doesn’t work, the process may be stored in a temporary Java List until the PID is deleted from the table.

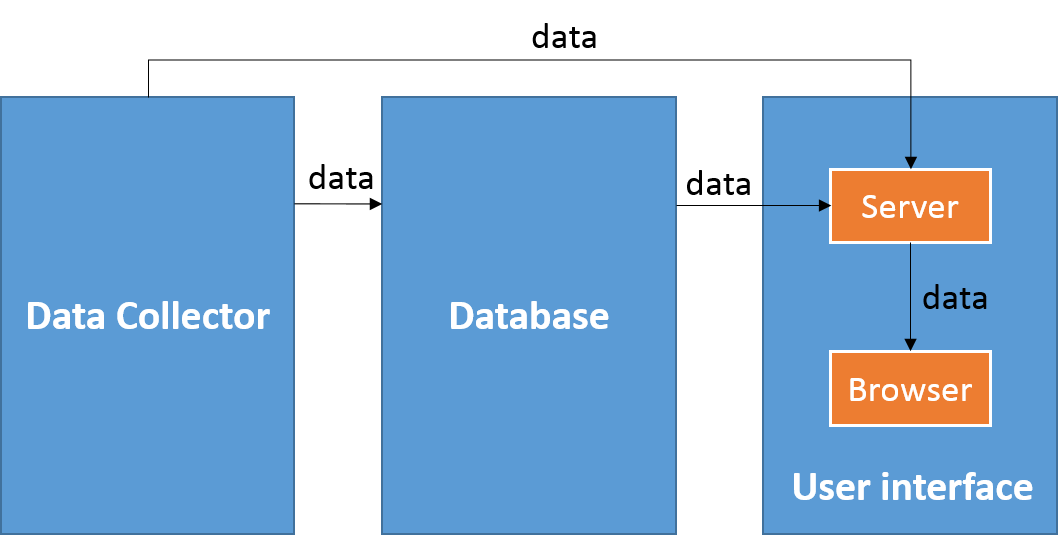
# Subsection V. Graphic User Interface

## **EPIC**

 All process metrics can be viewed from the user interface. The user interface for the metrics collector will be BS( Browser-Server) based. With BS base metrics collector, users can get the process metrics from any computer connected to the server besides the server itself. And users can get the process metrics from any operating system though the metrics collector will be development in Linux. When we show process metrics in the browser, we can use Javascript to draw a lot of expressive charts, which can be user friendly.

The browser side will be implemented by HTML and JavaScript. From the browser side, users can get information including:

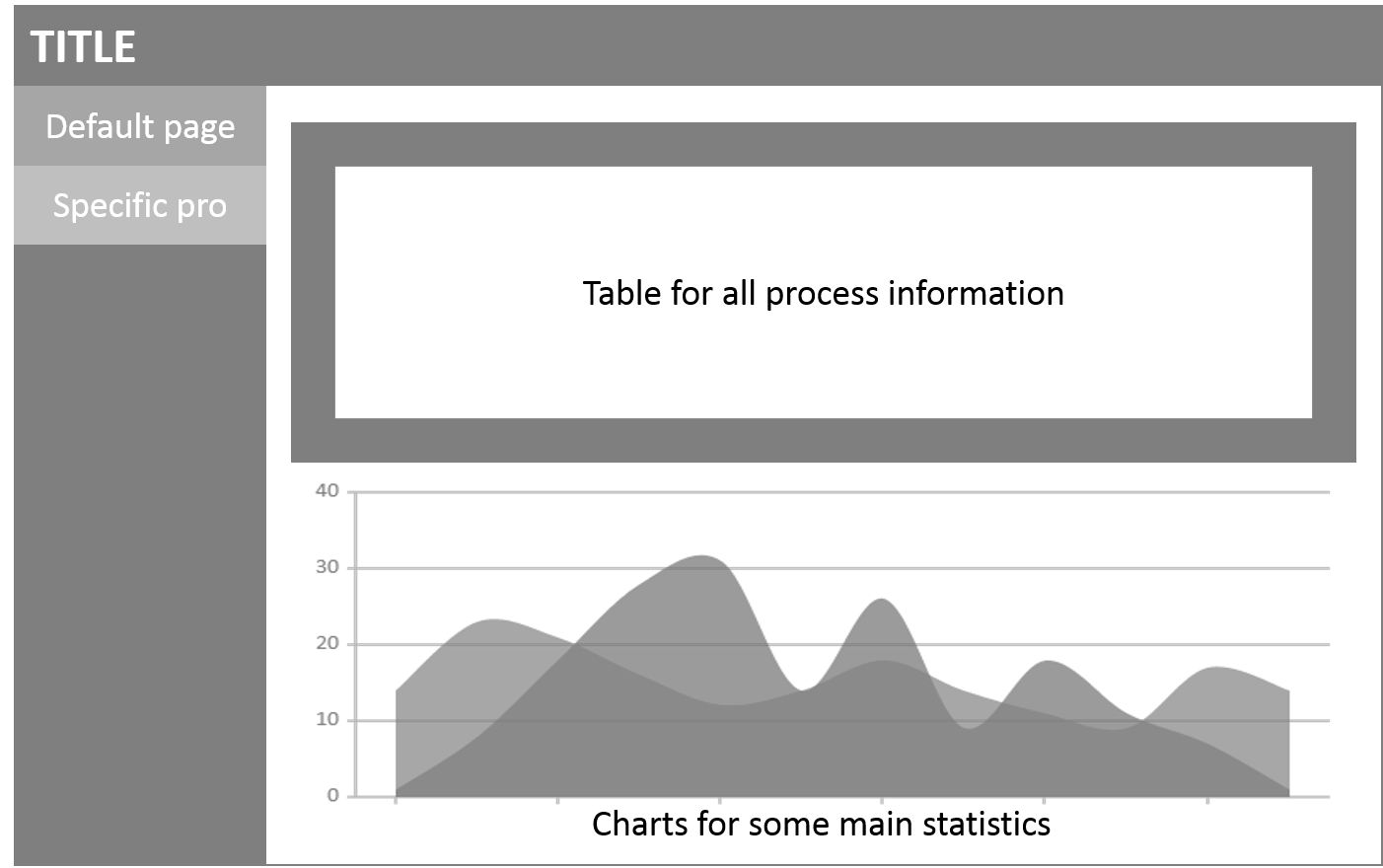
* The outline of all processes, the outline will be described with both tables and charts. Details in the outline including the number of processors, total memory usage, page fault counter, etc.
* The detailed information of every process, which will also be described with tables and charts. Users can query the information about a specific process by its PID or the name of the process. The information stored in the database will be shown.

The server side will be implemented by Java/JSP. The server side will interact with database and data collector. It will get data from the database or data collector periodically, like every 15 seconds.

**Diagram 2**. Shows how server and browser organized in user interface and how they interact with data collector and database.

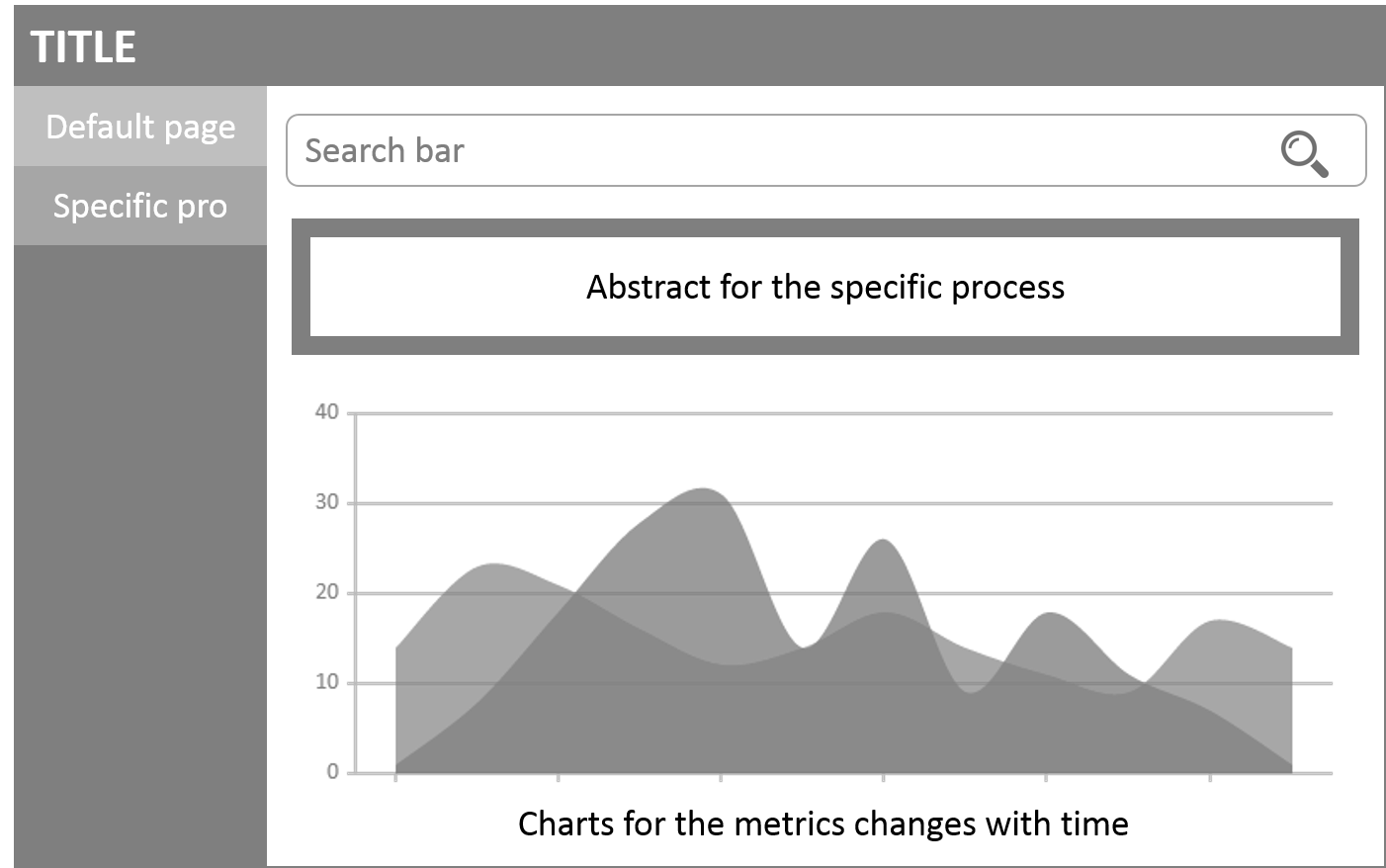
## **USER STORIES**

* Get process metrics from the database periodically.
  + Query process information from the database by stored SQL procedures every 15 seconds by default;
  + Cache the information got from the database for 15\*10 seconds by default;
  + The server package all the information in JSON to send it the browser page.
  + Users can adjust the frequency (every 15 seconds by default) to query the database by the given interface in the browser.
* Show the outline of all processes in the default page of the browser.
  + A table will be shown in the default page, which contains all the rows in the database.
  + Some charts will demonstrate some basic statistics of the processes in the default page.
  + All the information in the default page will be updated once the browser get the latest process metrics from the server.



**Diagram 3**. The abstract design of the default page of the user interface. Outline of all processes is shown in this page.

* Show the detailed information of the specific process.
  + There will be a page from which users can query the information of any specific process by its PID or its name.
  + Some charts will demonstrate how the metrics of this process changes with time.
  + If the user stays in this page, the information of this process will be updated once the browser gets the latest information from the server.



**Diagram 3**. The abstract design of the page which can get the detailed information of a specific process.

# Subsection VI. TESTING

- Test suite

- Maven- This will be the easiest for our style. Maven already has a precompiled library for JNI. Any automatic tests will be determined based on our manual testing.

- Unit testing

- Individual stored procedures will be tested first, using a separate Java program.

- Then, any method that calls stored procedures will be tested using another Java program.

- System testing

- Testing for when metrics are sent to the database at the will of the user/tester.

- Testing updating metrics after every 15 seconds.

- Testing updating metrics after a time specified by the user.

- Testing sending the data to the GUI after updating the SQL table.

- Acceptance testing

- Manual testing can be used as a basis for automated regression testing.

- simple cron replacement initializer (JNI or Java)

- Human testing

- Most of the testing in GUI will be human testing.

# Subsection VII. Specifics and concerns

- Weak points

- Java passing JSON to GUI.

- Reading in database, how do handle database errors?

- How do we handle logging with cron systems?

- How do we use a test suite for a system that uses JNI?

# Subsection VIII. Stretch goals and improvements

* Determine the hard links for each process.
* The user may be able to change the interval in which the SQL is updating.
* We may require login information and password to enhance security.