SOFTWARE REQUIREMENTS SPECIFICATION

for

Simulated Computing System

Version 1.0

Prepared by António Pedro Fraga

Cranfield University

December 23, 2017

Contents

1 Introduction

1.1 Purpose

This document is a Software Requirements Specification of a project developed under under two modules, **Software Testing and Quality Assurance** and **Requirements Analysis and System Design** at **Cranfield University**. It describes the implementation of a computing control system. This document is primarily intended to be proposed to the IT department for their approval and serve as a reference for the development of the system.

1.2 Project Scope

The software is a **simulator of a job control system**. It will be used by the IT department of Cranfield University so that it can explore different strategies to their current implementation.

The developed software will include an **User Friendly Interface**, so that it can be used more easily. The simulation shall be capable of regulate its **inputs** so that it can compute a set of outputs.

The resulting application shall be a **reliable** and **efficient**, **cross-platform** program.

1.3 Standards

1.3.1 Documentation

This document follows the **Requirements Specifications template (IEEE Std 830-1998)**, selecting the most relevant topics for its scope. The **Test Plan** document shall follow the **IEEE 829 format**.

Technical Documentation

Good technical documentation is a key to the project scalability, helping future developers to have a better understanding of the whole project structure.

The project shall contain, at least, a **README** file created in its **GitHub** page. This file must include some topics:

- Tests status
- Line coverage information

- Project name
- Description
- Technologies
- Informations about the development environment set up
- Structure information (ex: location of important files)

The software shall also include documentation about the developed methods and its structure. All methods shall contain at least:

- Method name
- Description
- Arguments (type & small description)
- Return (type & small description)

1.3.2 Version Control

This project shall have a **version control** system based on **GitHub**. The repository shall contain **two** main branches:

- Deployment - Containing a product release history
- Development - Containing a merge of the newly developed features

A branch shall be created every time a new feature starts its development.
The **feature** branch shall be merged with the **Development** branch every time its development has came to an end.
The **Development** branch shall be merged with the **Deployment** branch every time a new version of the product is released.

1.3.3 Naming

A naming convention is a set of rules for selecting the character sequence to the identifiers of variables, classes, modules or implemented methods. There are different patterns that can be adopted when defining a convention:

- **PascalCase (UpperCamelCase)** - the first letter of each word composing the identifier is capitalized.
- **Snake_case** - the elements of the compound words are separated with one underscore character and no spaces. The initial character of each element is lowercased, except for the first character, that can be both upper or lower case ('snake_case' or 'Snake_ case').

Files

C++ code shall be stored in **.cpp** files, whereas functions and variables definitions should be carried by **header** or **.h** files. File names shall follow the **snake_case** convention with the first character being uncapitalised.

White spacing

Blank lines improve readability by creating sections of code logically related.

Every statement shall be correctly indented. The indentation increases by a **tab** if the previous line is:

- **{** a left brace
- [a left bracket
- **(** a left parenthesis

The matching closing token must be the first in a line, restoring the previous indentation. Blank spaces shall be used in some circumstances:

Description	Example
A keyword followed by the **left parenthesis** should be separated by a whitespace. For example, there should be a space after an if or a while keyword.	– while (true) { ... – return (1 + 1); – if (job.is_short()) { ...
The method **return type** shall be followed by a white space.	**double** get_usage_price()
Every **comma** shall be followed by a line break or a white space.	insert_state(i**,** j**,** job);
Every **semicolon** at the end of a statement shall be followed with a line break.	i++**;** j++**;**
Every **semicolon** in the control of a loop shall be followed by a white space.	for (int i = 0**;** i < v.size()**;** i++) { ...

Classes

Classes should be defined as names written in the **PascalCase** convention. Acronyms and abbreviations should be avoided, unless the abbreviation is widely used.

Methods

The name of each method shall start with a verb in the infinite form, being a **verb-name** pair. This name shall be written in **snake_case**. It should be self-explanatory and concise.

Examples: is_short(), is_medium, get_name(), get_price()

Variables

Variables shall be written in **snake_case**. The use of descriptive names is required, avoiding single characters names, except for loops. Abbreviations shall be avoided as well. All variables must be declared in the top of the method body, whit a blank line between this declaration and the rest of the method, forming an **instantiation block**. The names (**i**, **j**, **k**) are reserved for iteration purposes.

Constant declarations must follow a **SNAKE_CASE** convention. The developer shall write them entirely with capital characters.

Branches

Branches shall be written in **snake_case**. Every branch shall have a name related with the developed feature.

2 Overall Description

2.1 Product Perspective

The product is a stand-alone system. This system shall contain **at least** 128 nodes with **at least** 16 cores per node. It will be used by a set of simulated users that can be classified as:

- IT support
- Researchers
- Students

The IT support simulated users have an **infinite** budget, therefore is permitted to them to run as many jobs as they like. In other hand, the Students shall have a constant budget, which is a smaller amount compared to the Researchers budget. This budget confines the amount of jobs that a user is permitted to run. The system usage has a price per core, that shall be decreased from the simulated budget every second.

The users can use the system by submitting jobs. This jobs have a two main characteristics, the amount of time that will use the system (running time), and the amount of system cores it will use. Thus, there are four types of jobs:

- Short - can take up to 2 nodes for no more than 1 hour. 10% of the machine is reserved for these kind of jobs.
- Medium - can take up for 10% of the total number of cores for no more than 8 hours. 30% of the system is reserved for this queue.
- Large - can take up for 50% of the total number of cores for no more than 16 hours. 70% of the system is reserved for this queue.
- Huge - can only run from 1700 of Friday to 0900 of Monday, reserving the whole machine. During this time, no other job can be executed.

Every time a simulated user submits a job, a scheduler shall define whenever that job is going to run. This scheduler manages the amount of computational resources at every second.

2.2 Product Functions

An user of the simulation shall be able to regulate a set of the inputs:

- Number of jobs [1]
- Number of users [1]
- Student Budget [1]
- Researcher Budget [1]
- Simulation date - the date when the first job submission is done.
- Requests span - the time span that a job can be submitted in.
- Number of nodes
- Number of cores

The simulation shall produce a set of outputs regarding:

- The number of jobs processed in each queue (throughput) per week
- The actual number of machine-hours consumed by user jobs
- The resulting price paid by the users
- The average wait time in each queue
- The average turnaround time ratio, i.e. the time from placing the job request to completion of the job divided by the actual runtime of the job
- The economic balance of the centre, calculated by subtracting from the actual price the operating costs

2.3 User Documentation

The software shall have an **User Manual**, indicating which features can be explored. The manual should include a **screen-shot** of the **User Interface** along with an explanation of every possible action. The screen shot shall clearly indicate which **spin box** value is related to each input. The manual shall also include an explanation of the simulation output. Every output value shall be clarified, enlightening the user about its meaning.

[1] The user shall decide whether this number is randomly defined following a linear distribution or is a constant value.

2.4 Assumptions and Dependencies

For simplification purposes, it is assumed that the scheduler works on a basis of a **First Come, First Served** methodology.

The running time of each job and the interval between job submissions follow **exponential distributions**.

The number of cores each job uses is generated following a linear distribution. The limits are defined according to each job type:

- Short: 1 core to 2 nodes of total cores
- Medium: 2 nodes to 10% of total cores
- Large: 10% nodes to 50% of total cores
- Huge: total number of cores

Every time an input is defined randomly, it follows a linear distribution.

It is assumed that only huge jobs can run during weekends, no other queue can be active during this period.

The default values of the **Operational Cost** and **Usage Price** shall be assumed as £**0.000475** and £**0.000014** respectively. The operational cost, is considered to have a constant value **per second**. The usage price is considered to have a constant value **per core, per second**. Further information about the calculations of these values can be found in the **appendixes** section.

3 System Features

<This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional hierarchy, or combinations of these, whatever makes the most logical sense for your product.>

3.1 System Feature 1

<Don't really say "System Feature 1." State the feature name in just a few words.>

3.1.1 Description and Priority

<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

3.1.2 Stimulus/Response Sequences

<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

3.1.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

REQ-1: REQ-2:

3.2 System Feature 2 (and so on)

4 Other Requirements

4.1 Performance

Performance shall be an important quality metric int this system. A system with **n** jobs shall schedule their running times with a **time complexity** of **O(n log(n) + nk)**, k being the average distance between the job submission and a free slot to run the job. Regarding **memory**, the system shall have a **linear complexity**. Further explanations about this subject can be found in the appendixes section.

4.2 Scalability

The system shall be scalable. The chosen design shall permit future implementations. Code must be refactored every time a better design approach is found.

4.3 Reliability

The system must be 100% reliable, which means that it must be able to function according to its specification under normal circumstances. The probability of failure should be null.

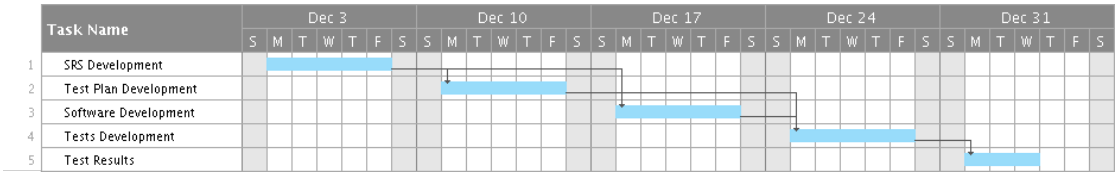
4.4 Portability

The resulting program shall be **cross-platform**, being able to execute on most of the operating systems and/or machines.

5 Appendixes

5.1 Appendix A: Project Plan

The project plan could be represented with a **Gantt Chart**. This chart is capable of representing dependencies between tasks, and each task duration.



5.2 Appendix B: Operational Cost & Usage Price

The default **usage price** is assumed to have the same value as **Archer**, the UK national supercomputing system. This system has a value of **5p** per core hour. Thus, for each second:

$$UPpercore = 0.05/60$$

The default **operational cost** is calculated with an average of costs that the centre has.

Firstly it's assumed that the simulated system is a **green** supercomputer, therefore it has a power supply of **60KW**. In the United Kingdom, a KW costs **£0.0285 per hour**. Thereby, it's obtained a value of **£0.0285 per minute**, and 0.0285/60 per second:

$$EnergyCost = 0.0285/60$$

The system is assumed to have a team of **10** elements making **£60.000** per year:

$$PersonelCost = 60000 * 10/365/24/60/60$$

The building maintenance costs an average of **£10.000** per year:

$$BuildingMaintenance = 10000/365/24/60/60$$

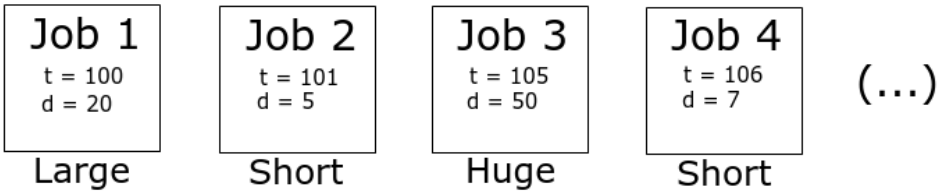
Every second, the default **operational cost** of the system is:

$$OperationalCost = EnergyCost + PersonelCost + BuildingMaintenance$$

Despite of having this default value, **the operational cost** can be changed as an input variable.

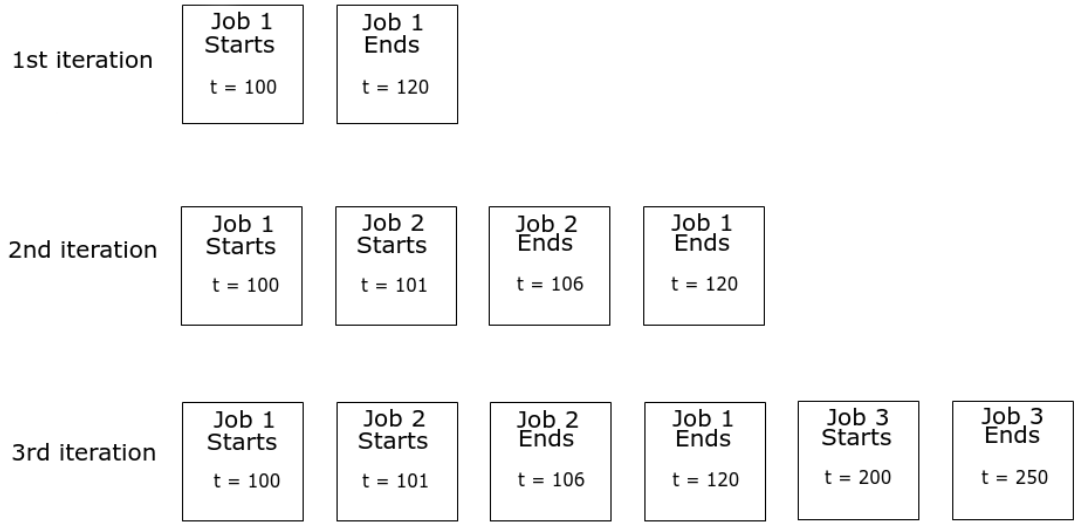
5.3 Appendix C: Implementation

Jobs will be generated randomly, having attributes characterized by the set of rules defined in this document. It is created an **array** of **Jobs**, sorted by **date of submission**. Every date will be stored as an **UNIX timestamp**, which is represented by the total number of seconds counted from **January 1st, 1970 at UTC**. This date is named as **UNIX Epoch**.



Jobs will be allocated to a random user. The **scheduler** will set a start time for each job on a basis of a **First Come, First Served** methodology. Since it will be important to keep track of the system state (queues computational resources available), it will be created a **vector** of **States**.

This vector contains the available computational resources at each **start** and **end** time of a job. Therefore, every time a job is analysed, the scheduler algorithm will add **two states** to this vector. A state representing the available computational resources once the job starts, along with the occurrence time of the event **(start)**, and a different state representing the available resources once the job running life comes to an end, containing the timestamp of the occurrence as well.



This vector shall be kept sorted every time two states are added to it. This way it will be easier to keep track of available **"running slots"**, which represent the nearest time the system is available to run a certain job.

Therefore, for purposes of performance, it can be saved the index to start looking for a free spot.

One knows that, as the **Job 2** is being scheduled, it is impossible to start it before the start date of **Job 1**. Since the array of Jobs is sorted by submission date, one can keep the **index** corresponding to future dates, or dates that will still be analysed by the algorithm.

Thus, this procedure, achieves a time complexity of $\mathbf{O(n\ log(n) + nk)}$. The initial $\mathbf{O(n\ log(n))}$, is the weight of sorting the array of jobs, whereas the extra **nk** corresponds to the iterating process through the vector of jobs, and the vector of states. Note that when iterating through the states vector, only future states are analysed. This is a good improvement comparing to the quadratic algorithm of iterating through every state every time the next "free slot" to run a job is needed.

The memory complexity of this algorithm is **linear**, because only a vector of states and a vector of jobs are kept in memory, a long with some constant values like the **index** of future dates.

5.4 Appendix D: Analysis Models

<Optionally, include any pertinent analysis models, such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams.>