Requirements and Analysis Document for NNN

Table of Contents

Version: 0.4

Date: 2013-03-26

Author: Victor Nilsson

This version overrides all previous versions.

1 Introduction

A computer can not perform calculations any faster than its hardware allows – that is common sense. However, if we allowed computers to share the burden of a particular computation, it would be able to process the same amount of data in considerably less time. That was – at the very least – our line of reasoning when we decided on making a set of applications that would allow computers to cooperate in order to solve those bigger problems more efficiently.

1.1 Purpose of application

The purpose of the application is to shorten the time needed for heavy calculations by distributing those calculations across multiple computers.

1.2 General characteristics of application

This is not strictly an *application* per se; it is a *system* consisting of three separate applications. There is a *client* application, a *server* application, and a *worker* application. The client is connected to a server, to which it sends requests for different tasks to be performed. The server has several workers connected to it, to which it delegates the work to be done. A worker then acts as a drone, performing any task delegated to it. Whenever the workers have completed a task, they will send the result back to the server which in turn returns it to the client who requested it in the first place. Tasks are specified by the client application, either by choosing from a predefined set of calculations, or by writing custom scripts in Python.

1.3 Scope of application

The application will, in the end, probably not be much more than a prototype of a proper system for work distribution. One of the things that this system will lack is the ability to split a bigger computation into smaller workloads. More likely is that we will be able to

provide the client with a tasklist, and distribute those tasks among the available workers. We will, however, probably be able to run user-provided scripts, allowing advanced users to make arbitrary calculations not supported by default. The predefined list of tasks will probably consist of simple – but heavy – calculations mostly in place for testing and demonstration purposes, while all other calculations will need to be handled by user-provided scripts.

1.4 Objectives and success criteria of the project

Our objective with this project is easiest explained by providing a simple example. Let us assume that we have a list containing a couple of thousand numbers, each number having at least six digits. If our system can calculate the prime divisors of these numbers faster than an individual computer, then we have really succeeded. Even a slightly slower system is an accepted accomplishment however, as long as we can get the distribution to work properly and see that additional computers connected to the server results in less time spent calculating. A properly established connection between our three applications is the first goal for the project, the next being the ability to send tasks over the aforementioned network.

1.5 Definitions, acronyms and abbreviations

We will probably write most about the client, the server and the workers. The client is, in this case, the application that will be used to define what calculations the system should work on, and will also be used to define the manner in which the work is split.
The server is the middle hand of the client and its workers, and will be useful for managing the system in the case of multiple client connections. The worker is the computer hosting the worker application, and will perform the vast bulk of the actual computations. One can think of the complete system as a simple consultant firm, wherein the client is – well – the client; the server is the consultant company that hands out work to its consultants; and the worker is an individual consultant. We will use the term “task” whenever we refer to a calculation that the client wants to be performed; in other words, the whole progress from the client sending a request to the client receiving a result will be referred to as a task. We will also refer to a particular worker’s current part of a task as its “workload”.

While we refer to the applications by different names, we will also refer to the users of each application with different names as a means to distinguish between them. The user of a client is simply a *user*, whereas server user is the *administrator*, and a person who is running a worker is a *volunteer.*

2 Requirements

2.1 Functional requirements

The most basic functions are the ability to define a task to be processed – such as the previous example with prime factorisation – and then be able to follow this task through the whole process for demonstration purposes. Also for demonstration purposes; a client should be able to communicate with the workers, to be able to show that the connection is up and running. This will also include showing the current status of the tasks at hand, for instance showing the number of tasks currently queued by the worker and – if possible and applicable – how much of the current task that still needs to be processed.
A client should be able to stop a task in progress whenever it deems necessary, while a worker should be able to stop the part of the task delegated to it. In the case of a worker cancelling its task, the workload of said worker should be returned to the server in order to redistribute it to another computer. The worker should also be able to at any point decline any further work, and focusing on completing its current workload. And of course, the client will be able to choose (and in some ways customize) a number of preconstructed tasks to send to the workers.
The server also needs to be able to be shut down, and clients and workers must be able to handle this event gracefully.

2.2 Non-functional requirements

2.2.1 Usability

This application will probably require a basic understanding of Python in order to use customized tasks. For other tasks, the GUI should be clear and concise even for a non-programmer, even though our users can still be expected to have above average computer skills. The server might need to support both a graphical and a command-line interface, for administrators with different needs.

2.2.2 Reliability

Above all, the connection should be stable, and errors need to be handled gracefully.

2.2.3 Performance

The transfer speed for task descriptions will hopefully not be a problem, though we need to be sure that this is indeed the case if we plan to – for example – send large matrices to perform operations on. The pre-defined tasks themselves need to be carried out in a sufficiently efficient manner to be able to show an actual gain in throughput.

2.2.4 Supportability

We do, of course, aim to make the program as expandable as possible when it comes to adding new predefined tasks to the client. Furthermore, the GUI is to be as loosely

coupled as possible, in order to allow logging and tracking of the aforementioned tasks, for instance.

2.2.5 Implementation

To use this system, all of the computers need to have JRE installed and configured. Moreover, the computer hosting the server must, of course, allow incoming connections and provide access to the specified ports.

2.2.6 Packaging and installation

The applications are easily downloaded and started as normal JAR files. However, the administrator must provide both the users and the volunteers with the IP-adress to the server in order to connect to it.

2.2.7 Legal

NA

2.3 Application models

2.3.1 Use case model

Abandon workload (Worker)
Abort current task (Client)
Establish connection
Kill (Worker)
Process customized tasks
Process predefined task
Show current status (Worker)
Show current status (Client)
Show current status (Server)
Show statistics (Worker)
Show statistics (Server)
Show statistics (Client)
Shut down (Worker)
Shut down (Server)

2.3.2 Use cases priority

The majority of this priority list is self-explanatory; the use cases with high priority is the bare minimum of our success requirements. However, the medium priority use cases are also very desirable to have in the final product, considering that a user would very much prefer to have control of the process - if the system is to even remotely resemble anything else than an educational project, these are close to mandatory. Low priority use cases would definitely make

the it possible to get a better overview of the tasks at hand, which would also help for demonstration purposes, but there are multiple solutions to that problem.

High:

- Establish connection
- Process predefined task (Client)
- Shut down (Server)
- Start (Server)

Medium:

- Abort current task (Client)
- Shut down (Client)
- Kill (Worker)
- Process customized tasks (Client)
- Show current status (Worker)
- Show current status (Client)
- Show current status (Server)

Low:

- Abandon workload (Worker)
- Shut down (Worker)
- Show statistics (Worker)
- Show statistics (Server)
- Show statistics (Client)

2.3.3 Domain model

When observed at a high level of abstraction, the model of this system is utterly trivial. What we have is, after all, is a client, a server, a worker and a task that we send between the first three. Of course, the worker also provides a result to be sent back, but See fig. 1.

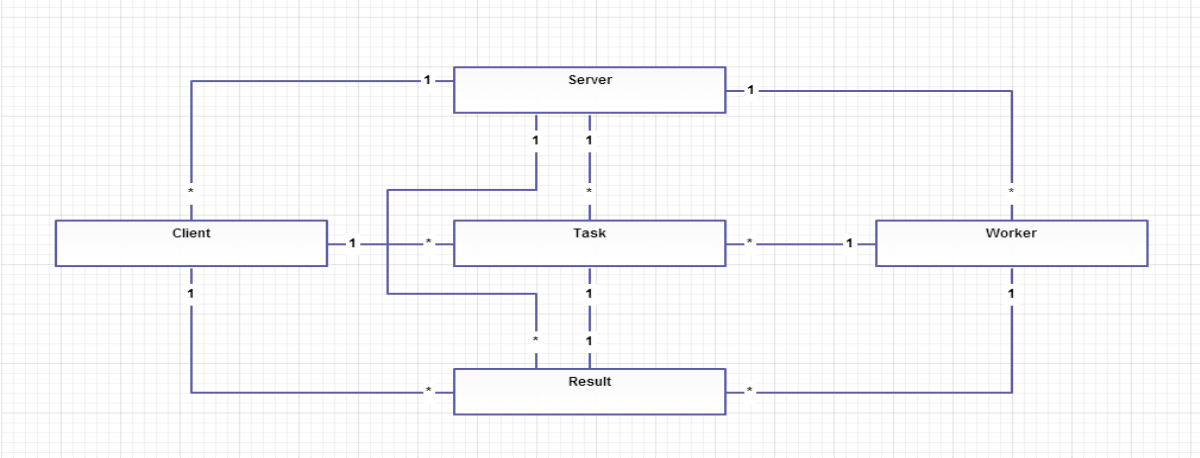

Fig 1: Domain model of the project.

For pictures of the analysis model, see the appendix.

2.3.4 User interface

The GUI for this project will mostly be for adding predefined task in a simple manner. It is a very lightweight and easy-to-follow GUI that allows the user to keep track of his sent tasks. The worker and server modules will be able to see statistics regarding their current workload (worker) and the number of connected modules (server) and so forth.

2.4 References

APPENDIX

GUI

Domain model

Use case texts