Utrecht University

Faculty of Science
Department of Information and Computing Sciences
MSc in Game & Media Technology

Master Thesis

Schedule and Distribute Personal Tasks

Remote Distribution of Personal Tasks for Large Game Studios

Author:
Tim de Haas

Student number:
4101472

Supervisors:
Dr. Wishnu Prasetya
Utrecht University

Frank Compagner
Guerrilla Games

Prof. Dr. Marc van Kreveld
Utrecht University

Dr. Ir. Marjan van den Akker
Utrecht University

March 23, 2016

Abstract

AAA game development studios often use continuous integration to monitor the state of development. The continuous integration processes usually run on a build farm. To prevent faulty changes from reaching the build farm the developers have the option to compile and test their work locally. Developers do not always consistently test. Faulty work can cause company-wide delays.

We present a novel framework that distributes and performs these personal tasks on a remote machine. This makes it easier for developers to test their work. Only a limited amount of machines are available for the remote personal tasks. This presents a scheduling problem. We build a framework to facilitate both the distribution and scheduling of these tasks. The framework allows for comparison between different scheduling algorithms, is light-weight, scalable, and can reliably distribute tasks to remote build machines.

We ask what scheduling algorithm presents the lowest average time a task spends in the scheduling queue. Three scheduling algorithms are implemented to answer this question. Random scheduling, the Highest Response Ratio Next algorithm (HRRN), and a derivative of the Heterogeneous Earliest Finish Time algorithm (NHEFT). We test these scheduling algorithms to determine which has the shortest average waiting time. Experiments are run using different amounts of build machines and different orders in which requests are made. A second experiment is run using five times the amount of tasks to find further data comparing random scheduling to HRRN.

The first experiment shows significantly longer average waiting time for NHEFT compared to both other algorithms. No significant differences are found between HRRN and random scheduling. The second experiment shows significantly shorter average waiting time for HRRN. We conclude that HRRN results in shorter average waiting time than random scheduling. NHEFT was implemented with a bug which caused higher average waiting time. No conclusions can be made regarding NHEFT.

Acknowledgements

Many people told me it was going to be hard to write a Master thesis during an internship. It was not easy. However, with the help of my supervisors and colleagues at Guerrilla Games this thesis is now a reality. I want to thank the people that made all this possible.
The very best times in your life are when you both enjoy going to work and enjoy returning home.

First, I want to thank my supervisor dr. Wishnu Prasetya, Utrecht University. I thank him for taking the risk in supporting me in my endeavor to write a thesis at Guerrilla Games. His guidance, support, and critical questions regarding my research have greatly contributed to this thesis. Special thanks are in order for dr. ir. Marjan van den Akker. Her advice and help in finding a valid simulation model for this research and finding the required techniques for verifying that simulation model was incredibly important, as was her assistance in using the verification algorithm.

Second, I want to thank my supervisor and boss at Guerrilla Games, Frank Compagner. I am very grateful for the opportunity to do my research at Guerrilla Games. His faith that I could contribute to the company has made this possible. I am very grateful for his support, his wise words which will make me a better developer, and of course the fun conversations at lunch.

Third, besides Frank I would like to thank all of my colleagues and friends at Guerrilla Games. They are an amazing team. They expect excellence and work very hard, yet they are always helpful and in for a laugh or two. Be it incredibly advanced discussions during lunch, playing Mario Kart, having drinks, or just saying hi at the coffee machine, it made it feel a little bit like home. One colleague in particular deserves mentioning. Emile van der Heide has helped me to learn the ropes from the very first day. He has contributed much to my research. From trouble-shooting my Python specific issues, to writing the RPC framework on which the Validation System has been built. His support, together with Frank and Denis Barth, has greatly helped and guided me in my research. Finally, I wish to mention the IT department at Guerrilla Games. I am grateful for Oliver Kogelnig and his team's patience which has allowed me to perform the second set of experiments.

Fourth, I would also like to extend special thanks to Wouter Boereboom. A former classmate who has supported me during the entire process of my research and writing. Our daily conversations kept me on track while we offered each other mutual linguistic, writing, and technical support. My six amazing roommates have also been of great help listening to my thesis issues, sending me to the library to work, and very important: being quiet at night.

Finally, I would like to thank my family, Anja Lissone, Johan de Haas and Steven de Haas. Without you none of this would have been possible. Thanks for the support in difficult times, the proof reading, and all the tips and skills you have taught me through the years.

Contents

1 Introduction

1.1 Onset

Software and game developers alike spend a lot of their time waiting on builds and tests. This waiting is often inefficient and can be experienced as frustrating. We research a possible solution to a part of this problem. This research is done at Guerrilla Games (Guerrilla).

Guerrilla Games is a AAA game development studio based in Amsterdam. At Guerrilla 200+ developers work on a single software product which takes several years to complete. All the developers contribute to a single source base in version control. The source-base contains all code, audio and graphical assets for the game Guerrilla creates. The source-code and assets in this source base are continuously built and tested. These builds and tests are performed on a build farm. The build farm consists of a number of computers with very high processing and storage capacity.

Whenever an error occurs in the build farm that error causes the build to fail. This build failure, or build breakage, prevents the creation of new usable versions of the game. The developers use the latest version to run, test, and debug their work. Build breakage results in delays throughout the entire company because Guerrilla works on a single product.

To prevent faulty changes from reaching the build farm the developers have the option to compile and test their work locally. However the process of building and testing personal changes can take a significant amount of time. During this time the developer's workstation can become unusable when the build or test process takes up all of the workstation's resources. A developer should also not edit anything in his local repository during the building and testing processes. This is why some developers do not test their changes but submit them straight into the main source repository. These processes test the current local state which may differ from the state in version control. More recent changes in version control may conflict with the changes made by the developer.

We want a system that performs these personal tests on a remote machine. This will alleviate stress from the user's workstation and allow him to continue working during the tests. It is possible to assign several of the build farm's build machines to this purpose. However these build machines may not have enough power to easily perform all tests during peak usage.

1.2 Problem

There are over a hundred developers and less than twenty potential build machines available at Guerrilla. The developers may execute a large amount of tasks every day. The limited amount of build machines combined with the high amount of requests presents a scheduling problem. It is important for the developers to minimize average waiting caused by the queuing of requests. More or less build machines may become available depending on which phase development is in. We need to use a scheduling algorithm that can deal with a varying amount of build machines and a varying amount of requests. All requests must be processed, starvation of a request is not allowed.

To solve this problem we need to build a framework that can handle the requests, schedule the requests, and assign the requests to a variable amount of build machines. The framework should meet the following requirements:

1. The framework should be able to present the required data for any scheduling algorithm to run. It should also present the required functions for a scheduling algorithm to efficiently schedule workflows.

2. The framework should be scalable such that it must be able to handle build machines being added to or removed from the build environment.

3. The framework should be able to deal with high peak usage. Performance should remain within acceptable limits.

4. The user's personal changes need to be reliably integrated into the remote machine's local repository. These changes should be tested in an isolated state on the latest validated source in version control. No other changes should influence testing.

5. The framework should be approachable through a front-end tool. The user should get timely, meaningful and clear feedback about the state of the processes and any errors. Either through the front-end tool or other channels.

Most scheduling algorithms require estimates of the duration of a process. Tasks as complex as those in a build system can have a strongly varying duration. The unpredictability of task duration makes scheduling algorithms like Shortest Job First less reliable. We will need to test several scheduling algorithms to determine which has the shortest average waiting time.

1.3 Research Questions

The problem statement brings us the following two research questions:

1. How can we make a framework which can facilitate different scheduling algorithms whilst adhering to the above mentioned requirements?

2. The use of what scheduling algorithm results in the shortest average waiting time?

1.4 Research method

We describe the framework developed to solve the stated problems. We elaborate on the ways we implement the framework such that all requirements are met. We explain which functions have and have not been implemented in the framework. Finally we show which functionality the framework could and/or should have in other case environments.

To answer the second research question we test the framework using three scheduling algorithms: Highest Response Ratio Next, a simplified version of Heterogeneous Earliest Finish Time, and random scheduling. The duration of the tasks in the build system at Guerrilla Games is quite long averaging between 3 and 30 minutes with outliers nearing four hours. The changes that cause shorter or longer task duration vary greatly. To fit enough test sessions within the scope of our experiment we choose to simulate task duration. The simulation is based on historical task duration from the continuous integration build farm at Guerrilla.

1.5 Contributions

We present contributions to the field of science as well as to AAA game development.

- We present a novel framework to enable developers to distribute their personal build/ test/conversion processes to build machines. It is a light-weight alternative to other tools as can be found in subchapter 3.3, aimed more closely at our given problem. The framework is built to handle a variable number of requests and resources. It is scalable as such that it allows resources to be added or removed to the system at any time. This framework can take into account heterogeneous resources, large amounts of requests, as well as interdependent tasks and workflows. The framework allows different scheduling algorithms to be implemented. The framework is created using *Python 2.7.6*.
- We have translated higher level complex tasks such that they can be scheduled by lower level scheduling algorithms. We have compared three scheduling algorithms based on their ability to minimize average waiting time.
- Contribution to AAA game studio development. We present a light-weight, scalable framework that can distribute tasks in a manner similar to cloud computing. We provide a comparison of scheduling algorithms to show which algorithm minimizes average waiting time. The framework should increase developer efficiency by removing negative effects caused by waiting on builds/tests.

1.6 Thesis Content

This thesis is organized as follows. Chapter 2 provides a description of the case in which we implement our solution. In chapter 3 we show related work and previous research on which our solution is based. In chapter 4 we describe the framework we created to solve our problem and answer the research questions. Chapter 5 shows our research method. The results of the experiments are shown in chapter 6. We criticize these results and the drawbacks of our research and implementation in chapter 7. In chapter 8 we answer our research questions. In chapter 9 we make recommendations for future work. Appendix A contains the results for our experiments with 1 and 2 build machines.

2 Case Description

Figure 1: Guerrilla Games

2.1 Guerrilla Games

Guerrilla Games (Guerrilla) is a Sony owned game development company based in Amsterdam, the Netherlands, with a branch in Cambridge, U.K. The company was founded in 2000 by joining three small Dutch studios and was acquired by Sony in 2005. Guerrilla Games is known for the Killzone game series, which can be played on most of Sony's console and hand-held platforms. This thesis will not contain any further references to the Cambridge studio.

Since it was founded Guerrilla has grown as a company with currently around 200 employees. The entire company works on a single product, the game "Horizon: Zero Dawn". The game was announced at the Sony's press conference during the E3 2015 conference in Los Angeles. All employees work on the same version of the game, there are usually no branches. Contrary to common practices in the software industry everyone at Guerrilla Games adds to the same source in version control. This means that everyone works and tests in the most recent version of the full game, be they programmers, artists, or designers. The source is managed using the version control system Perforce. The version control system has since been renamed to Perforce Helix. Perforce is used by many large electronic entertainment companies e.g., Disney-Pixar, Ubisoft, Nvidia [13].

2.2 Continuous Integration

New versions of the game are continuously built and tested to ensure that all employees can test their work on the latest version of the game, while using the latest tools, all without any errors. At Guerrilla Games this process of building and testing is performed by the build farm. The build farm consists of a set of over 40 build machines each with a very high processing capacity. These machines have been assigned one or more specific tasks. Each of these tasks is necessary in the process of building the full game product, from the main executable to a deliverable package.

Several of the build farms main processes are:

1. Compiling and building the game's executable.
2. Checking linked references between game assets.
3. Converting art assets into data which the game can interpret.
4. Run tests to verify basic gameplay and AI functionality.

There are many more processes run by the build farm. However, in this thesis we focus on these four main processes. The build farm runs these processes continuously whenever a build machine is free and relevant changes have been submitted to Perforce.

Because of the many changes that are submitted each day there is a high possibility of mistakes and bugs making its way into the source. These bugs and mistakes create errors. These errors indicate problems that make the latest version of the game unfit for testing. This unfit new version will not be submitted into Perforce and thus it will take at least until the next round of builds/tests for a new version to reach the developers. This can create delays in the development. When this happens to the entire company it can add up to a lot of lost time.

2.3 Local Build

It is possible for all members of the team to test their changes locally. This can be done by using the local build tool (LBT). The local build tool is currently a batch script that allows the user to select a build or test process and performs this selection. The tool then returns the process output and will communicate whether the process was successful. It does this by turning the command window background color green or red respectively for successful and erroneous results. Using the LBT is not mandatory within Guerrilla Games. Developers do occasionally skip this step and submit their work without verifying whether it may break the build.

The LBT presents a large subset of the processes that can be run on the build farm. These processes are run on the users local machine, ergo local build tool. Some of the processes can not be run on local machines as they lack processing power compared to the build farm. In these cases the build farm process is replaced with a light-weight version of that process. These local tests give a very reliable indication whether personal changes contain any errors or bugs. Note that it is important to perform these tests only after obtaining the latest working version from source control. This will likely ensure that the personal changes do not cause bugs because of changes that have been made since the last synchronization between the user's local machine and Perforce.

The personal changes can be submitted to Perforce after the user has run the relevant processes locally and if none of those processes were erroneous. After submission the build farm will run its extensive processes and verify the changes together with the changes of other users. The newly built executable is only submitted to Perforce when all processes in the build farm finish successfully.

2.4 Simplification of the build tasks

The tasks performed by the LBT are of a complex nature. Any task is run by executing a single script. However, this script may contain many executable calls. These steps differ for each task but are equal for all LBT users. To be able to schedule these tasks we choose to view each task, however complex, as a single process with a beginning and an end. This approach is valid for all the tasks in the LBT as they all start and end. For our new framework there is a desire to run multiple tasks instead of having to run tasks one by one.

2.4.1 Workflows

We call a set of these tasks a workflow. A workflow can contain any number of tasks but should contain at least one task. Each task can occur only once in each workflow. Tasks within a workflow can be dependent on other tasks but this is not required. Workflows can be interdependent. This interdependence of tasks must never be cyclic to avoid deadlocks in a workflow. The dependency between tasks will prescribe whether some tasks can be executed in parallel or whether they should wait until one or more other tasks are complete.

Example:

A workflow contains three tasks: 1. Compiling the game executable 2. Converting game assets and 3. Checking links between game assets. 'Checking links between game assets' does not depend on the completion of the other tasks and it can thus be completed in parallel. 'Converting game assets' requires the 'Compiling the game executable' to be complete. In the case that executing in parallel is disabled all tasks will be executed in order.

The next chapter shows the relevance of simplifying complex build tasks into tasks and workflows.

3 Related work

In this chapter we present the ways the industry has solved problems similar to the problem addressed in this thesis. In subchapter 3.1 we elaborate on Continuous Integration. We show Grid Computing and two of its derivative forms, Cloud Computing and Public Resource Distribution in subchapter 3.2. We explain why currently available software is not sufficient to solve our problem within the case environment in subchapter 3.3. In subchapter 3.4 we show different scheduling algorithms that could be used. Finally we explain how we use simulation and how to verify the credibility of a simulation in subchapter 3.5.

3.1 Continuous Integration

Continuous Integration (CI) is the automated continuous compiling, building, converting and testing of software source code and (art) assets. One or more processes continuously build and test the game, using the latest source code and assets available.

The paper by Fowler [8] presents a clear description of what CI is and what is needed to make a successful product. We describe the requirements presented by Fowler that relate to the requirements in subchapter 1.2:

- When using continuous integration it is important to maintain a single source repository to build from. This source repository should have as few branches as possible.
- It is important to automate the build to be able to continuously compile executables in an efficient manner. The automated system should test the builds to a certain level and commit the tested executables. This way all developers get access to the latest successful build.
- Considering the problem in this thesis Fowler [8] marks that it is very important that personal or branched changes should be tested *before* merging with the main branch.
- Fowler suggests that the integration tests in the previous point should be performed on a separate integration machine.

We intend to meet these requirements in our solution. However, the problem of scalability and the limited amount of build machines remain.

Stolberg [16] presents a review of the implementation of continuous integrated testing in an existing agile development environment using the guidelines presented by Fowler [8]. The paper shows the practical demands needed to take into account when implementing the required baseline on which this thesis is built. It confirms the importance of continuously integrated builds and shows a case in which only post-build tests are performed. The question of implementing a distributed and scheduled pre-build testing system remains unsolved.

3.2 Grid-based technologies

3.2.1 Grid computing

Grid computing solves problems very similar to the problem in this thesis. Grid computing uses a set of possibly heterogeneous computers as nodes. These nodes can be accessed either locally or remotely. The nodes are used to execute a set of one or more processes. These

processes are executed on a subset of a large data-set. To be able to account for higher and lower process load, nodes can be allocated to the grid dynamically. This way the grid is able to scale up and down according to its usage [10].

Grid computing can be implemented to offer different services like data services, computational services, or application services. Baker et al. [3] describe these services and present the necessary requirements for grid computing. The four base requirements presented are:

- "Grid fabric", the hard- and software resources that the grid consists of. The grid fabric is already in place in our case environment.
- "Core Grid middleware", the remotely accessible portal on the grid machines.
- "User-level Grid middleware", middleware that channels and schedules user requests to the grid machines.
- "Grid applications and portal", user interface tools to assign work and receive feedback.

The Core and User-level Grid middleware as well as the user interface will be implemented in our solution.

To be able to find the data necessary for a scheduling algorithm to work, we need to look at implementations of grid computing. Zhang et al. [20] present a recent approach to grid computing optimization scheduling. This approach combines a random-based algorithm with a genetic algorithm (NSGA-II) where grid nodes compete to maximize their output. This paper contains many variables that are needed to efficiently schedule grid processes. These variables help us to determine what information and data the scheduling algorithms will need to be able to receive.

We wish to make a system that allows the scheduling of a set of tasks on a scalable set of heterogeneous build machines. However, grid computing is a relatively old technology. Next we investigate technology that is derived from grid computing to see how it has evolved. This gives us the opportunity to see where those evolutions present solutions to our problem.

3.2.2 Cloud computing

In 2003 Figueiredo et al. [6] published a paper presenting a way to schedule virtual machines as grid nodes. This makes it possible to have as many nodes as are required without being dependent on machine specific variables. Cloud computing is essentially based on these principals.

Cloud computing is a popular term in the current IT business. Similar to grid computing, cloud computing offers processing capacity as a service. Cloud computing brings us closer to the future where computing resources are available in the same universal manner as electricity. Buyya et al. [4] give a thorough early overview of cloud computing. It also explains the underlying principles, uses, and risks. Foster et al. [7] give a clear comparison between grid computing and cloud computing. The virtualization of the process capacity is important in cloud computing. Computing, data resources, and applications are offered as a service. This service is paid for based purely on usage and is very scalable. Grid computing is created to be able to deal with heterogeneous systems while in cloud computing this is handled by the provider. As a user, scheduling in a cloud computing system could consist of as little as assigning fewer or more virtual machines.

Recent research aims to optimize the usage of the cloud machines and improve the autonomous automation of the cloud computing process. Imai et al. [9] present a middleware system that can autonomously scale the amount of cloud virtual machines to deal with high load. This is done by migrating applications away from systems with a high load when more processing power is made available. Xiao et al. [18] present an alternative to the aforementioned approach. This approach aims to lower power cost by intelligently assigning virtual machines. This is achieved by migrating processes to keep machine temperatures between certain thresholds.

3.2.3 Public volunteer distributed computing

Public volunteer distributed computing (PVDC) is another alternative to grid computing. Where cloud computing is a business driven technology, PVDC is based on unused processing power available on user work stations. Work stations are often idle and do not use their computing capacity. These work stations may be part of a university, a company, a science lab, or they are volunteered by private individuals. Idle computing time is often available when a computer is left running but the user does not use it. PVDC aims to use this idle processing time. Part of the available processing capacity is used to make various calculations. These calculations contribute to a larger centralized project. A good example of PVDC technology is the application BOINC. BOINC is developed by Berkeley University [1]. BOINC hosts the calculations of several huge public projects, e.g. filtering radio noise from space which is collected by telescopes. Volunteers can install BOINC on their home, work, or university computer and contribute computing power to their choice of projects.

Anderson et al. [2] give a deeper explanation of BOINC and the challenges it has to deal with, for example reliability and connection instability. More research has been done to improve the speed of the individual BOINC calculation. For example Costa et al. [5] use the bittorrent networking infrastructure to reduce time lost on network I/O. Both these papers take into account specific requirements aimed at solving problems with the nature of the grid nodes. Because the nodes in a PVDC system are inherently unreliable, a lot of redundancies need to be implemented. For example, nodes can have hardware failure, the network connection can fail, or the node may go offline for an unknown amount of time. The systems in place to reduce errors based on these problems are overly redundant for our problem. However, we kept these systems in mind when designing our solution.

3.3 Available tools

In this subchapter we discuss several existing technologies that are currently available. We describe what these technologies do and why they are not optimal as a solution to our problem within our case environment. We will not provide an example for grid computing systems as cloud computing and PVDC are evolutions of grid computing.

3.3.1 Ownership

First we want to state that regardless of the advantages and disadvantages of any retail software package there is one important reason to build a tool from scratch: ownership. Building a software tool is usually more expensive than buying a working retail alternative. We assume an acquired tool has a community that uses the tool, extensive documentation,

and continued support after purchase. However, there are downsides to this. It is more difficult to debug when encountering a bug in a licensed tool than when one is found in an in-house tool. Without strict Service Level Agreements the user is dependent on the vendor's priorities to fix bugs. The retail tool may have features that remain unused and the tool may require extensive configuration before the tool works as desired.

Creating an in-house tool gives the advantage of access to the tool's creators. This can make debugging and fixing problems a much easier and quicker process. With the source code available debugging the tool's code becomes possible. More importantly, by building the tool internally it is possible to create a tool that matches the specifications exactly. Only those features that are needed are implemented, thus avoiding unnecessarily complex tools. The common practice within Guerrilla Games is to build tools in-house for most minor functionalities. Major software packages like Perforce, Visual Studio, and Maya are exceptions to this rule as well as several proprietary software packages that are part of Sony.

3.3.2 Distributed continuous builds Jenkins

Jenkins is used by companies like eBay, Sony, bol.com, and NASA. It is an open source application for assigning and monitoring continuous integration. Jenkins can automatically run builds, sync source versions, and return feedback in multiple ways. Jenkins is open source, supports several build languages, and has extensive documentation. Jenkins seems like a very good solution to our problem. However, Jenkins is not scalable and it does not schedule a large set of requests. Most importantly Jenkins will not be easily compatible with the current build system at Guerrilla Games. The tool is created to perform tasks that are also handled by Guerrillas in-house build system. This in-house system would be replaced when Jenkins is used which is undesirable.

3.3.3 Cloud computing Amazon web services EC2

Amazon web services (AWS) EC2 is Amazons cloud computing product. EC2 stands for Elastic Compute Cloud. The user can assign different size virtual machines and a different amount of virtual machines on demand. This could prevent the problem of queue overflow. These virtual machine instances are used to run software assigned by the user. The instances are accessible like any other machine thus providing the ability to debug remotely. Using EC2 is relatively cost efficient and will guarantee sufficient build machines as long as the financial budget allows it. At Guerrilla Games processing power is already available which lowers the necessity for a cloud solution. Because the build machines are located very close to the Perforce repository syncing data within the company is potentially much faster than pushing the necessary data over an internet connection to the cloud. Cloud computing has high potential for future build farms but an in-house solution is preferred as it will be easier to integrate with the current build farm and local build tool.

3.3.4 Public Volunteer Distributed Computing - BOINC

BOINC is the textbook example of a PVDC system. It allows companies and research studies to set up distributable projects and delivers a user-friendly client for volunteer machines. BOINC is created to take into account many variables that are not present in our current study. As the content of the product being developed at Guerrilla Games and most other

AAA game development studios is top secret it is unwise to distribute calculations to volunteers. BOINC projects perform a single type of processing over a very large data set. This means that every step in the build process requires a separate project. This is inefficient and doing so defeats the purpose of using the PDVC architecture. In conclusion PDVC is not a usable solution to our problem.

3.4 Scheduling algorithms

We have shown the evolution of grid computing and recent scheduling algorithms. These scheduling algorithms often solve problems that are not relevant to our current problem. For example Xiao et al. [18] solve for environmental issues that apply on a much larger scale than the Guerrilla Games build farm. Although the focus of Imai et al. [9] on requesting resources can be a great asset to our framework we have chosen not to implement it. An explanation why can be found in paragraph 4.7.4.

We want to find scheduling algorithms that are more suited to our specific problem. We are able to use more simplistic scheduling algorithms because we simplified the tasks performed by local build as shown in subchapter 2.4. The book *Metaheuristics for Scheduling in Distributed Computing Environments* contains a chapter on *Workflow Scheduling Algorithms for Grid Computing* by Yu et al. [19]. This chapter contains a clear overview of which algorithm would fit which scenario. The HEFT algorithm presents itself as a logical choice because our workflows have dependencies between tasks and the workflows are scheduled individually.

Based on the requirement to minimize average waiting time we also look into other algorithms. The Min-Min heuristic in the chapter by Yu et al. gives shorter tasks or workflows a higher priority. Shortest Job First (SJF) scheduling provides exactly this heuristic. Finishing the shortest jobs before the longer jobs should decrease the overall waiting time when the number of users is high. SJF does risk starvation of tasks with a long execution time. Scheduling algorithms like Highest Response Ratio Next (HRRN) solve this problem by prioritizing using a combination of expected execution time and the time a task has already spent in a queue.

3.5 Simulation

Simulation is the process where a randomized model is used to imitate real world events. Based on the model the simulation can generate output. The randomization of the output is based on parameters within the model. Chapter 5 of the book *Simulation modeling and analysis* explains how to create a valid simulation model and how to verify this model [11].

Simulation is used when it is not realistic to perform real testing. For example when launching expensive rockets into space or when a large amount of experiments should be performed within limited time and budget. A simulation must present output that resembles the real version of the experiment as closely as possible. To be able to achieve this we take the following steps: determine what output to generate, determine what variables the simulation is based on, create a simulation program and validate the output. Validation is important to verify whether the simulation is an acceptable representation of a real envirnoment.

3.5.1 Variables

The first step in creating a valid simulation is selecting what output the simulation should have. What variables should be randomized? Which aspects of the system would normally be measured? In case the simulation will be used to substitute a part of an already existing experiment the answers to these questions are straight forward. When this is not the case it is important to design an experiment. Data needs to be measured to produce results for the experiments. These data will define the output of the simulation. By determining which variables the output is based on it is possible to control which variables are randomized and which are not. The level of simulation detail must be determined when all the variables are known. The variables in the simulation should be of a relevant level of detail. Too precise or complex variables can lead to errors or may not attribute any significant changes to the output. However, variables that do not have enough detail may cause the output to be too imprecise to analyze. Once all the variables and their level of detail are determined a simulation program can be created that randomizes the variables and generates output. [11]

3.5.2 Validation

It is necessary to verify whether the simulation program's output is a valid representation of what the output would be in a real environment. This way the credibility of the simulation can be confirmed. A simulation program can be inaccurate in many ways. No simulation can ever compare to the real environment it models. However, by validating its credibility it can be determined whether the simulation is acceptably close to reality. [11]

Validating a simulation can be performed using different approaches. Because our simulation is not very complex we present two of the techniques that could be used to validate the simulation in our experiment. The simulation program needs to be tested regularly during its development. Doing so ensures separate parts of the software work as intended. By tackling problems early on it should also prevent bugs that are complex and difficult to debug from forming. Another method of validation is verifying the distribution of the simulation results against historical data. In cases like our experiment where historical data are available these data can be compared to the simulation output. The simulation can be validated by comparing the distribution of the output to the historical distribution. Examples of important variables to compare are the mean, variance and the range of the data. [11]

Our simulation is based on an empirical set of data measurements and the simulation strongly resembles an empirical cumulative distribution. The Kolmogorov-Smirnov Z test (KSZ test) is used to test whether the hypothesis that two data sets originate from the same distribution or data set is true. When the two sets of data points differ too much the hypotheses is rejected. We compare the mean and variance of our simulation as well as using the KSZ test. The KSZ test compares the historical and simulated data. An implementation of the KSZ test can be found in the work by Marsaglia et al.[12]

4 Framework

In this chapter we describe the proposed framework with which to solve the requirements in subchapter 1.2. We present a brief overview of this chapter in subchapter 4.1 and a technical description of the framework in subchapters 4.2 to 4.6. Finally, we present several possibilities to expand the framework in subchapter 4.7.

4.1 Overview

Our solution is built in two parts. The TLocalBuild tool replaces an old command line batch file with a user friendlier GUI application. The old batch file only allowed for one task to be requested at a time. TLocalBuild allows the user to select a workflow of tasks. Task output is enriched and shown in separate tabs per task. The GUI offers the user the option to build remotely. The Validation System is the framework that manages these remote build requests (workflows) from all users in the company. It stores these workflows in a queue and lets an assigned scheduling algorithm assign the workflows to a pool of build machines. Each build machine is assigned to the pool as a single Validation Builder. The framework allows different scheduling algorithms to be implemented. The following three algorithms have been implemented for our experiment: a simplified version of Heterogeneous Earliest Finish Time (NHEFT), Highest Response Ratio Next (HRRN) and a scheduling algorithm that randomly assigns tasks to Builders.

The Validation System consists of four parts. The Validation Coordinator (Coordinator), the Validation Builder (Builder), The Validation Client (Client) and the scheduling algorithm running within the Coordinator. Here we shortly illustrate the function of each part. The Validation Coordinator is available as a service. It runs the scheduling algorithm when a request is submitted or cancelled, when tasks or workflows are completed by the Validation Builder, or when a Validation Builder assigns or removes itself from the queue. The Validation Builder runs tasks when they are assigned to it and manages the local repository of the build machine it runs on. The Validation Client is run by TLocalBuild when the user chooses to build remotely or cancel a request.

4.2 TLocalBuild

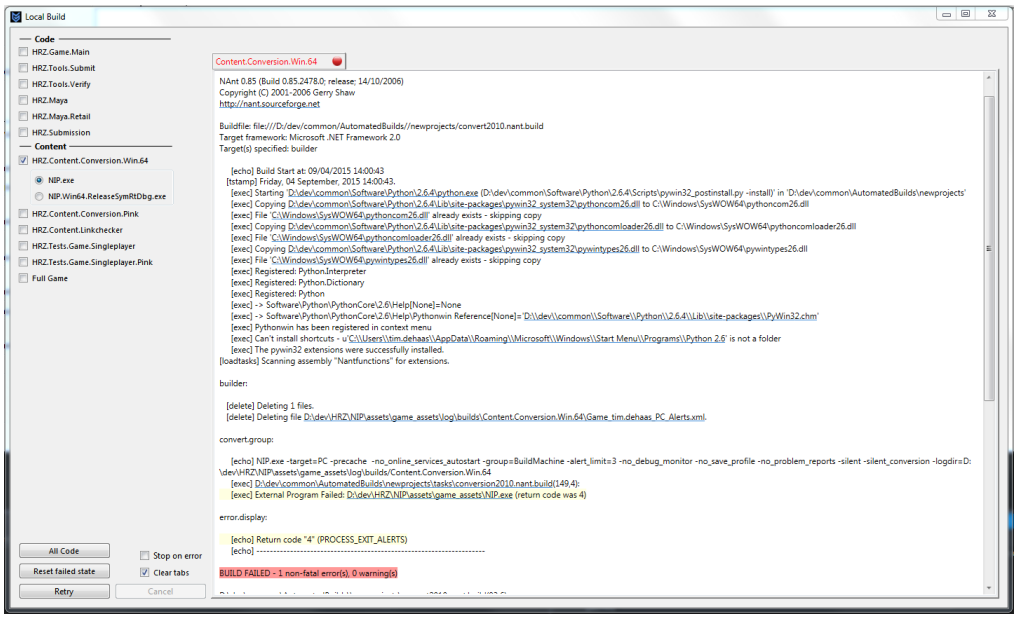

Figure 2: TLocalBuild UI Interface

The original way to locally run the Continuous Integration's (CI) tasks at Guerrilla Games is via batch file. The program contained in the batch file allowed for a single task to be run simultaneously. The resulting output was presented in a command line window. The command line window's background was colored red or green when the task executed unsuccessfully or successfully respectively.

TLocalBuild is created to improve the usability when running CI tasks on a local machine. This tool allows the user to combine tasks to create a workflow. The tool is written in C# and is integrated on all work machines company-wide. Figure 2 shows the layout of the tool. On the left the user can choose a set of tasks, i.e., a workflow. The user can also input a limited amount of parameters. After selecting the desired tasks, the user runs the chosen workflow by clicking the Build button. The tool executes the necessary programs in order and presents the program output on the right. The output for each task is put in a seperate tab. The output data are presented as text. Lines of output that contain indications of errors or warnings are highlighted. After the completion of a task the tab clearly shows whether the task has failed or succeeded. This is done by coloring both the tab and the text background green or red.

The user can choose to check the Remote option which is not shown in Figure 2. When this option is chosen and the user clicks Build, the tool will ask the user to choose a changelist. This changelist should contain all changes that need to be tested. After a changelist is chosen the tool requests a remote build through the client script of the Validation System. When the workflow is run on the remote Builder, TLocalBuild will show the task output in a tab. This output is shown in an identical manner to running locally. There are two differences for the user between local and remote building. First the user needs to choose a changelist when building remotely. Second it may take more time before process output is shown. However, when running remotely the local machine will not be taxed by the task execution.

4.3 Validation System

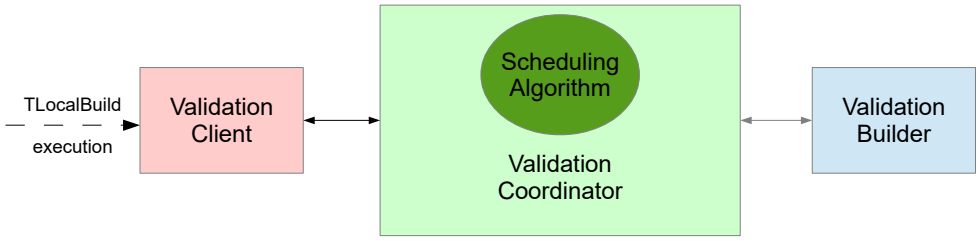

Figure 3: Simplified Validation System

To be able to implement and test different scheduling algorithms we create a remote build validation system called the Validation System. Figure 3 shows a simplified diagram showing the structure of the system.

When a user chooses to run his workflow remotely TLocalBuild sends a request to the Validation System. This request is sent by running a client side Python script, i.e., the Validation Client. This script sends the request to the Validation Coordinator (Coordinator). The Coordinator puts the request in a queue and schedules all requests to be assigned to a pool of Validation Builders (Builders). These Builders then perform the tasks in their assigned workflow and send the task output back to the user. The Coordinator runs a desired scheduling algorithm. It facilitates all the data needed for scheduling and it provides the necessary functions for the scheduling algorithm to work.

The sequence diagram in figure 4 shows the main steps of the Validation System. The main loop in Validation Coordinator runs continuously but it only runs the scheduling algorithm when the request queue or Builder pool has changed. Unlike the Builder and the Coordinator the Validation Client does not initialize on start up. It is run only when a request needs to be sent and exits after the request is complete.

The Validation System is written completely in Python. The current implementation is built upon a Remote Procedure Call (RPC) system which is also written in Python. Python scripts can call certain functions on other machines using RPC. The RPC system used in our experiments was built in-house by Guerrilla Games. The system notifies the caller whether their recipient still exists. In other words the Coordinator can recognize whether Builders or Clients are still online and available as a service.

Figure 5 shows the three Validation System modules. Their internal classes are shown in boxes with the class name underlined. The Validation Coordinator starts a loop in *Init* which runs one of the three scheduling algorithms. The arrows indicate where functions call each other using RPC. The *RequestQueue* contains *WorkflowRequests* with a set of *Tasks*. The *BuilderPool* contains *Builder* objects which contain a Validation Builder RPC object. The NHEFT algorithm is implemented using the Python version of HEFT created by Matthew Rocklin [15]. To be able to implement as much of HEFT as possible in NHEFT we have three extra functions. *ComputationCost* to compute the cost of a workflow on a given Builder, *CommunicationCost* to compute the cost of transferring a process to a different

Builder, and *CreateDAG* to create a DAG for a given set of tasks. The HEFT algorithm requires the ability to transfer a currently running process from one machine to another. Although NHEFT does not have this feature it is available through *TransferToBuilder* to facilitate the first requirement of our problem statement.

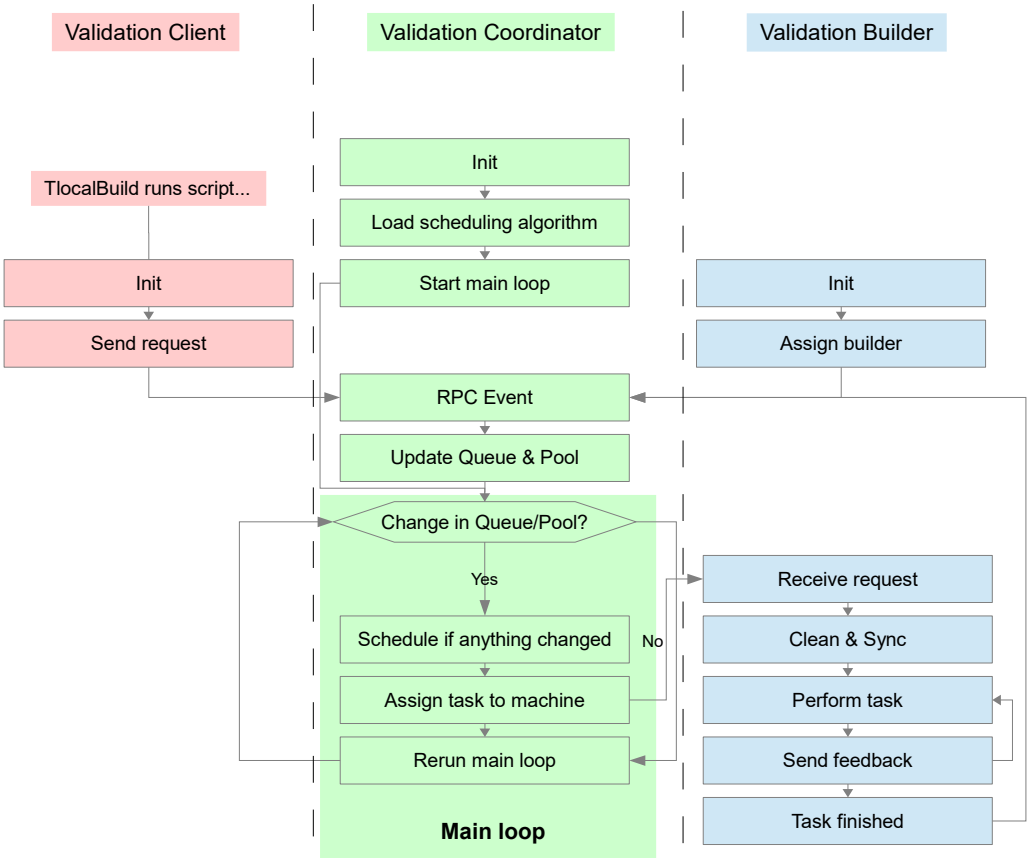

Figure 4: Validation System Sequence Diagram

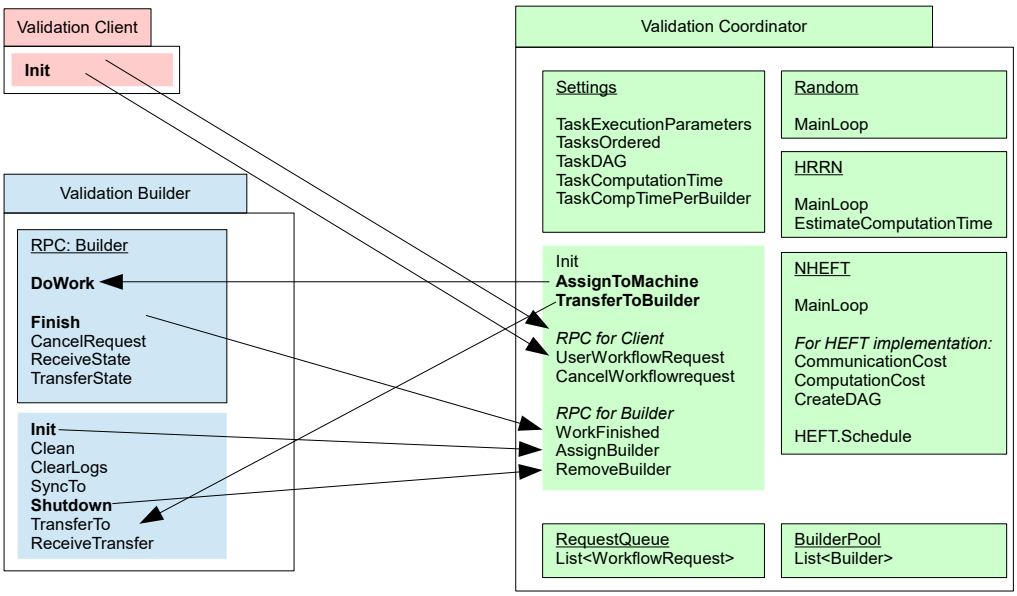

Figure 5: Validation System Class Diagram

4.4 Validation Client

To facilitate communication between the Validation System and TLocalBuild we use a client-side script. In our implementation of the Validation System this client-side script is called the Validation Client (Client). This script parses workflow requests and sends these requests to the Coordinator. The Client can also request cancellation of the user's current workflow request. Cancellation is possible for workflows still in the queue as well as for workflows being executed on Builders.

The Validation Client is a simple communication script. Besides translating and sending workflow requests it can act as a service to relay data. For example: expected queue times, task output, or the request's location in the queue. To prevent complications we have chosen not to include these communication channels in our implementation. However, when the Client exits it does return the Coordinator's unique identifier for the request. TLocalBuild can use this identifier to cancel the request.

4.5 Validation Builder

The Validation Builder (Builder) is essentially a straight-forward script. A single instance of the Builder script runs on each build machine. It executes tasks and sends its output to a recipient. However, to do this correctly the Builder needs to do more.

Because the goal of the Validation System is testing whether a set of changes will break the Continuously Integrating build the Builder needs to stay up-to-date. When the Builder receives a new task and this task does not belong to the same workflow as the previous task all personal changes should be reverted. However, the changes made in the same workflow must be retained. Reverting the previous changes will prevent those changes from contaminating the results of the next task. Contamination could lead to false positive or negative results. It would also make it very difficult for users to analyze the output and debug their changes. After reverting the previous changes the Builder also synchronizes to

the latest working version of the source code. This ensures the local source code is up-to-date. The Builder then applies the changes in the changelist associated with the workflow request to the local source code. Only then does it execute the requested task.

While executing the task the Builder sends the task output to TLocalBuild. The output can be sent through the Coordinator, directly to the Client when it acts as a service, or straight to the user tool. We have chosen to implement the latter. TLocalBuild sets up a TCP listener to receive the task output directly from the Validation Builder. It may be more robust to use the Coordinator or Client because handling connection issues may be more straight-forward within the Validation System. Finally, after the Builder finishes a task it notifies the Coordinator and becomes available for a new task request.

4.6 Validation Coordinator

The Validation Coordinator is the heart of the Validation System. The Coordinator manages a queue of workflow requests and a pool of Builders. It runs a scheduling algorithm and assigns workflows and tasks to Builders. The Coordinator also contains a set of settings and stores data that are needed for the scheduling algorithms to function. The Validation Coordinator allows us to use and compare different scheduling algorithms.

The Validation Client can request or cancel workflows via RPC. The Coordinator adds these workflow requests to a queue. The Validation Builder script will assign itself to the Coordinator upon which the Coordinator adds the Builder to a pool. Within the Validation Coordinator a loop runs which checks whether the queue or pool has changed. When a change has occurred the Coordinator runs the assigned scheduling algorithm to schedule or reschedule the Builders and the requests. If the scheduling algorithm does not directly assign workflows and tasks to Builders the Coordinator will do so after scheduling has occurred.

Scheduling algorithms can be added to the Coordinator as long as they are written in Python. The algorithm can use all the Coordinator's available data and functions. It should contain a function that is called every cycle of the scheduling loop called MainLoop. Only when the queue or pool have changed will MainLoop be called. In our current implementation we have integrated "random assignment" of workflows, the Highest Response Ratio Next (HRRN) algorithm, and a simplified version of the Heterogeneous Earliest Finish Time algorithm called Not-HEFT (NHEFT). The details regarding the implementation of NHEFT can be found in paragraph 5.1.3.

Because the tasks in our case are either linearly dependent or independent of other tasks we have chosen to assign a single workflow to a single Builder. That Builder then runs the tasks in the workflow in sequential order.

4.7 Framework Potential

The framework as tested in our experiments is created to specifically cater to the build farm at Guerrilla Games. There is more potential in the framework than has currently been implemented. The following paragraphs show some of the potential contained in the framework.

4.7.1 Estimates

Various scheduling algorithms require to know the estimate time a process will take to run. The current state of the framework uses manually set estimates for each task. These estimates are based on prior experience with the Continuous Integration system at Guerrilla Games.

The execution time of each task after completion of said task should be stored. By doing this a more accurate average time to completion can be calculated. The size and complexity of the source repository will change. This is why it is important to weight newer measurements more strongly than older measurements. Removing measurements older than a certain length of time is advised, e.g. measurements older than two months.

4.7.2 Heterogeneous build machines

The current Validation Coordinator has its estimates stored per known Builder. This becomes very useful when heterogeneous build machines are used. During the experiments at Guerrilla Games build machines with identical specifications were used. Scheduling algorithms like HEFT become much more useful when build machines have different specifications.

The available resources may also differ from build machine to build machine when parallel computation is possible. This would require scaling the estimated time to finish a task by the expected percentage of available resources. Parallel computation or resource sharing is not feasible in our case environment. However, adding less used build machines to the Validation System can be beneficial in other environments.

4.7.3 Build Interruption & Transfer

Preemptive scheduling algorithms can stop or halt a process to allow higher priority processes to run in their stead. Stopping a running task is a part of the current framework. However, transferring the data on which the task is performed to a different machine is not feasible. Neither is storing the state of a task at the time of interruption. The build machines at Guerrilla Games have a single local repository from which they build. This repository is many gigabytes in size thus transferring costs would be very high. So high that transferring the local repository would never be considered in the scope of our experiments.

4.7.4 Resource requesting

In Cloud Computing it is possible to automatically allocate extra resources during times of high system load. The case environment build farm consists mostly of machines with a dedicated purpose. These machines can therefore not be allocated to the Validation System unless their purpose is sacrificed. We did not add resource requesting functionality to the framework for this reason. However, it becomes an important part of the Validation System when using cloud scheduling algorithms.

4.7.5 Queue time communication

It would be useful for users to have an insight in the length of the queue. For the user's time scheduling purposes it is important to have an estimate of when the results of the requested

workflow are available. The estimates should be available to calculate using most scheduling algorithms. When estimates per workflow or per task are not available, for example when using random assignment, the size of the queue itself could be communicated to the user. Communication should very likely occur via the client UI which is TLocalBuild in the case environment.

5 Research Method

In this chapter we describe our research method. First, we explain which scheduling algorithms are implemented in subchapter 5.1. Second, we shortly describe what data are measured and how in subchapter 5.2. Third, in subchapter 5.3 we describe how and why we simulate task duration. Finally, we describe how the experiment is set up in subchapter 5.4.

5.1 Scheduling algorithm implementation

In order to determine the best scheduling algorithm for our case environment we compare three scheduling algorithms. We decided to compare three different types of scheduling algorithms. To set a baseline we implement an algorithm which uses as little information about the tasks as possible: the naive scheduling algorithm. We then implement an algorithm which intuitively seems to be the best fit for the case scenario: the intuitive scheduling algorithm. As a requirement the intuitive algorithm should use some of the data available in the Validation Coordinator. Finally we implement an algorithm that shows the Validation System can support complex scheduling algorithms: the complex scheduling algorithm. This algorithm should use as much information and functions as can realistically be expected within the case environment.

5.1.1 Naive scheduling: Random

Several common scheduling algorithms can be classified as naive scheduling algorithms based on our criteria. Possible candidates are First In First Out (FIFO), First In Last Out (FILO), and purely random scheduling. Of these three, random scheduling requires the least known data. The order in which the workflows/tasks enter the queue is required for FIFO and FILO. Random scheduling only requires to know how many workflows/tasks are in the queue.

Random scheduling is implemented in its simplest form. For each available Builder the algorithm randomly picks a workflow from the queue and assigns it to the Builder. Randomization is achieved by using the `random.choice()` function in *Python 2.7.6*.

5.1.2 Intuitive scheduling: Highest Response Ratio Next

Users expect a long task to take longer than a short task. Waiting ten minutes in a queue is less frustrating when the task will approximately take an hour compared to a task that takes only thirty seconds. The shortest workflows/tasks should be at the front of the queue to avoid long waits for short jobs. The Shortest Job First (SJF) algorithm is an optimal fit for this problem. However, this could theoretically lead to starvation of jobs with a long estimated running time. To avoid starvation we use the Highest Repsonse Ratio Next algorithm.

HRRN prioritizes workflows/tasks based on two variables. First, their estimated execution duration and second the time the workflow/task has spent in the queue. This way longer workflows/tasks gain priority as they remain in the queue longer. This ensures their eventual execution. HRRN does not interrupt running processes in our implementation.

5.1.3 Complex scheduling: "Not" Heterogeneous Earliest Finish Time

Finding complex scheduling algorithms is not a difficult task. Finding a complex scheduling algorithm that applies well to our case environment presented more of a challenge. Many grid-based and cloud-based algorithms use very short processes using small amounts of data. These processes must execute independent from each other in most algorithms. An algorithm that takes into account these process dependencies is HEFT. HEFT relies on a heterogeneous set of processors. This is useful in our case environment because not all build machines have equal specifications. HEFT allows processes to be paused and transferred to a more suitable processor when necessary. The amount of time this transfer takes is integrated in the algorithm.

HEFT is an algorithm that makes full use of the capabilities of the Validation System. However, its implementation in the experiment is limited such that we can not call it HEFT. This is why we name the algorithm that is currently implemented in the Validation System: Not HEFT (NHEFT).

NHEFT uses the Python implementation of HEFT created by Matthew Rocklin [15]. However, there are several important differences between HEFT and NHEFT. First, NHEFT does not allow for transfers between Builders. Estimated transfer time between Builders is required to run the HEFT algorithm. To ensure no transfer is ever considered *CommunicationCost* returns a high value. Second, the estimated time to complete a task is equal for all Builders because the build machines they run on have equal specifications.

Finally NHEFT has an issue in the code. The two shortest tasks in the queue are performed latest because of a bug in the *ComputationCost*. HEFT calculates which task in the queue would finish the fastest on which machine and schedules all active Builders accordingly. Builders that are currently running a task are taken into account even though they are not available. When a Builder is not available *ComputationCost* returns an extremely large negative number. For example: in the case that three Builders are active and one Builder becomes available the scheduling algorithm will assign the two shortest tasks to the two unavailable Builders. The third shortest task is assigned to the available Builder. NHEFT does not allow transfers and thus assigns the third shortest task to the available Builder. The two shortest tasks are processed last. This results in very long *queue* times. This is reflected in the results which can be found in chapter 6.

5.2 Measurements

We compare the different scheduling algorithms based on the time that workflows are in the queue and how long it takes to complete them. The time it takes for workflows to progress through the Validation System is measured from the moment that a workflow enters the Validation Coordinator until the moment its last task is completed. This information is stored within the Validation Coordinator. The data are stored in an XML file.

First we store events from external factors. This includes Builders assigning themselves to the pool as well as workflows being requested. Second we store workflow events: when the workflow enters the queue, when it gets assigned to a Builder, and when its last task finishes. Finally we store events for the tasks: the moment a task is run and the moment a task finishes. All data are collected and stored by the Validation Coordinator. Comparing Coordinator measurements with Builder measurements is not relevant because transfer and

function call times between the Validation scripts are near zero. Measurements made on the Builders are discarded because of this.

Measurements are taken with microsecond precision for all experiments as described in subsection 5.4.

5.3 Simulation of build behavior

Measuring the time needed for workflows as they would be tested in full production is not feasible. To achieve a realistic representation of actual use of the system a varied amount of programmers, artists, and designers would need to provide changelists with non-trivial changes. The type and amount of changes in those changelists have great influence on the execution times of each task. E.g.: A programmer's changes will have different effects on different tasks when compared to an artist's changes. At the time of testing the Validation System was not sufficiently finished for professional use in the case environment. This presents a challenge when testing the Validation System. A solution needed to be found to test the Validation System other than user-based testing.

One solution would be to create non-trivial changelists. However, this would not be realistic within the scope of the experiment. The tasks themselves do not necessarily need to be executed because we measure time spent on workflows and tasks. We therefore simulate the tasks by running processes with a simulated duration instead of the actual tasks. This presents the question how to simulate the build behavior.

5.3.1 Distribution data

No data on personal build tasks executions have been collected at Guerrilla Games. No systems were in place to do so. However, the Continuous Integration system does collect the duration of every task performed. These data go back for many months with a multitude of tasks performed each day. The build tasks are performed on build machines similar to the build machines in the Validation System.

The main difference between the build farm and a build in the Validation System is the amount of changelists. Before a build task is performed by the build farm many changes are applied. These changes come from many changelists submitted by different users. The amount of changes is usually higher than the amount of changes in a single user's changes. There is a lesser variety of changes in the Validation System. We choose to disregard these differences based on the following:

A Continuous Integration task has an execution time T, where $T = T0 + Tc$. Each build task has a base execution time of $T0$. The execution time increases with more changes thus increasing Tc. The length of both $T0$ and Tc can vary. Unless a previous change reverts a certain change it may be assumed that an increase in changes will always increase Tc. The Validation System user can add any amount of changes to a single changelist. Perforce allows the user to combine multiple changelists into a single changelist to be tested using the Validation System. The amount of users submitting changes and the amount of changelists containing changes do not define an amount of changes. Changelists have a minimum amount of one change. The amount of changelists has no direct influence on Tc. Only the amount of changes do.

Because we miss data for the number of changes submitted in a personal test we resort to the data we do have at hand. We want to be able to simulate use of the Validation System based on the Continuous Integration data. To do so we assume the distribution of task execution duration between these two systems is congruent.

5.3.2 Simulation basis

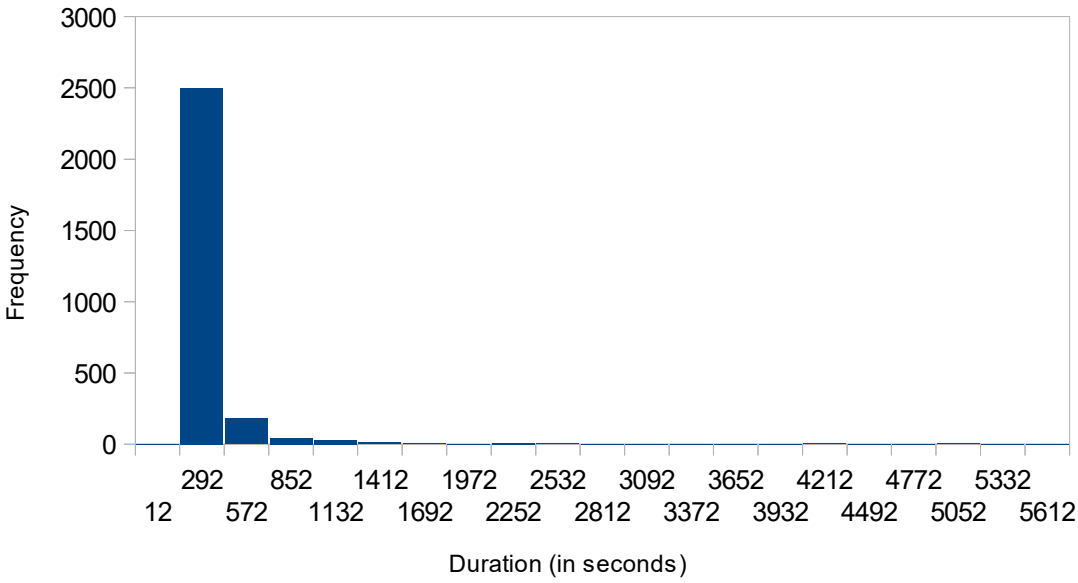

Figure 6: Conversion

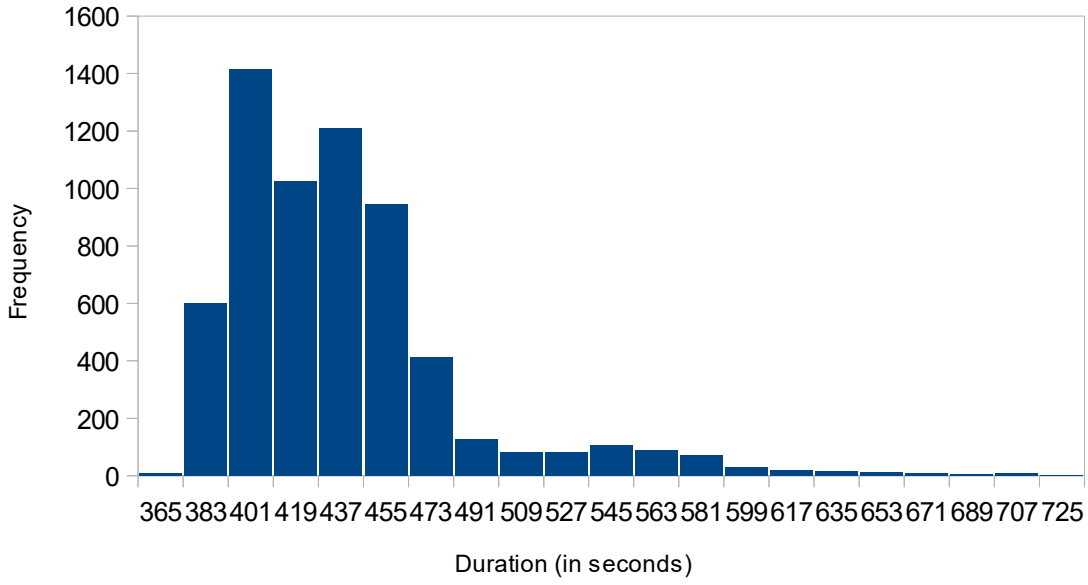

Figure 7: LinkChecker

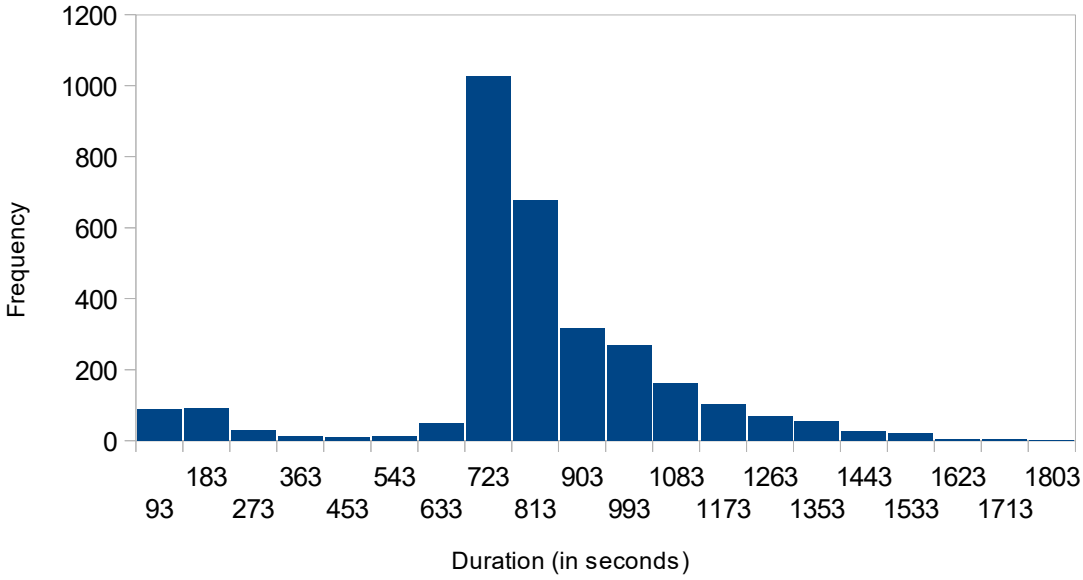

Figure 8: Game Main

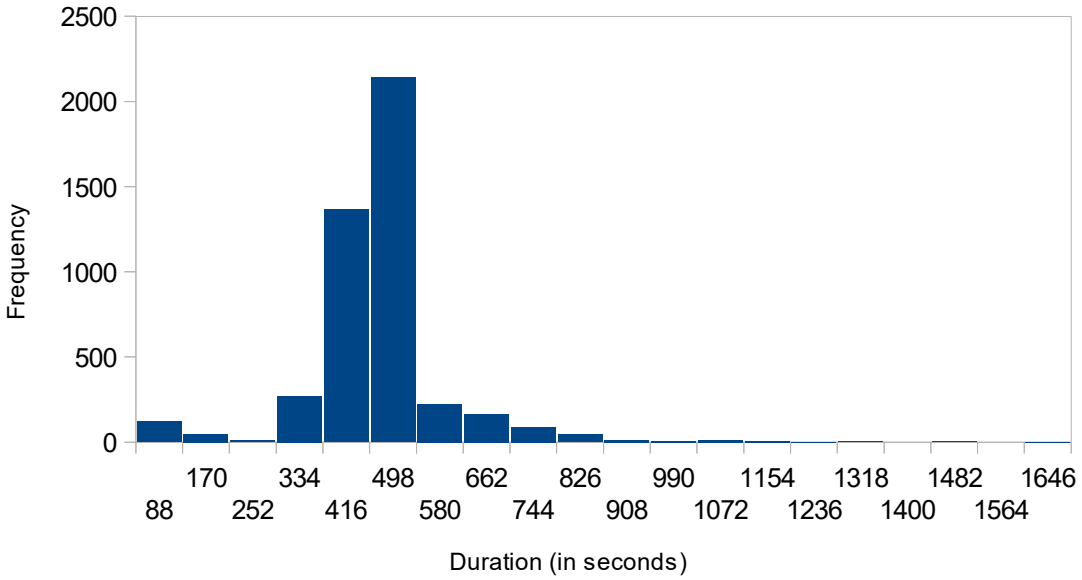

Figure 9: Test Singleplayer

For our simulation we've chosen four distinguishable tasks to simulate. A short, a medium, and two long tasks. The latter two are grouped in one workflow. The short task is based on Conversion, the medium length task on LinkChecker, and the longer tasks are based on Game.Main and Tools.Verify. The task execution time of these tasks has been parsed for a period of two months. These two months contain periods of lower and higher workload. The data are divided into a histogram with 20 bins. Figures 6 to 9 show these histograms. Data points that show clear signs of outside influences, for example build farm failure resulting in extremely long or extremely short builds, have been removed as these data points do not reflect valid task execution.

We base the distribution model for our simulation on the histograms. To make the

experiment shorter we divide the simulated data by a factor of six. We use algorithm 1 to determine how long a simulated task should run. Bins in the histogram represent a certain length of time. The height of the bin represents how many times the measured task execution time falls within the range of that bin. All bins are weighted by their amount of occurrences and a bin is then randomly chosen. The simulated time is randomly generated from within the range of the chosen bin.

Algorithm 1 Randomized task execution time simulation

Input: TaskType
Output: Time

```
1: Histogram = GetHistogram(TaskType)
2: TotalValue = 0
3: for Bin in Histogram do
4:     TotalValue += Bin.Value
5: RandomValue = Random.Choose(0, TotalValue)
6: CurrentBinIndex = 0
7: CurrentValue = Histogram[CurrentBinIndex].Value
8: while CurrentValue < RandomValue do
9:     CurrentBinIndex++
10:    CurrentValue += Histogram[CurrentBinIndex].Value
11: Time = OffsetTime + TimePerBin * CurrentBinIndex + Random.Range(0,
    TimePerBin)
12: return Time
```

5.3.3 Verifying simulation credibility

We use the Kolmogorov-Smirnov Z (KSZ) test to verify the simulation. The KSZ test is used to verify empirical distribution functions. See subchapter 3.5 for further details.

5.4 Experiment set-up

Our research consists of two experiments. A set of simulated experiments and a simulated experiment with an increased number of tasks. The first set of experiments compares the three scheduling algorithms on different amounts of Builders and different orders of workflows. The amount of workflows is predetermined. The second experiment is performed in the same manner but with five times the amount of workflows. With this experiment we try to determine whether the limited amount of workflows in the first experiment has an influence on the results.

5.4.1 Physical setup

The Validation Builder scripts are run on three identical build machines at Guerrilla Games' build farm. A fourth identical machine is used to run the Validation Coordinator. The RPC service used for the experiment is run on the same machine as the Validation Coordinator. The simulation script runs on a work machine inside the office.

The average task execution time for the selected tasks is added to the settings storage of the Validation Coordinator. When the simulation script is run the experiment starts. The simulation script requests a set of workflows. One mock client process is started for each workflow. This client acts as if it is TLocalBuild in such that it listens to the feedback

it receives from the Validation Builders. Instead of running a task the Builders start a mock process. This mock process generates a randomized duration based on our simulation distribution and executes for that amount of time.

5.4.2 Broad experiment

The first and largest of our experiments consist of 49 test runs. The tests are a combination of scheduling algorithms, the amount of Builders, and the order of workflows.

The three scheduling algorithms are random scheduling, HRRN, and NHEFT. The three Builder set-ups are 1, 2, and 3 Validation Builders on seperate machines. We also have five different sets of 20 workflows. These sets consist of three types of workflows: a workflow with one short task (S), a workflow with one medium task (M), and a workflow with two longer tasks (L). Each set of workflows contains two long, four medium, and fourteen short workflows. This division is based on expert knowledge on the expected amount of tests performed company-wide.

Table 1: Workflow request ordered sets

Even distribution	SMSSLSSMSSSSMSSLSSMS
HRRN specific distribution	MMLSSSSMMLSSSSSSSSSS
HEFT specific distribution	SSLSMSLMSSSSSSSSMMSS
Long tasks in the front	LMMLMMSSSSSSSSSSSSSS
Long tasks in the back	SSSSSSSSSSSSSSMMLMML

Table 1 shows the order of the workflows in the ordered sets. The HEFT and HRRN distribution are based on the behavior of both algorithms in an environment where three Builders are available. For HRRN we first fill all three Builders with longer processes so the queue can fill with a set of shorter and longer workflows. All shorter workflows that come after the first three should then get priority over the medium and long workflows. For HEFT we ordered the workflows in such a way that interruption and transferring can be considered. NHEFT does not take this int account so both interrupting a workflow/task and transferring it to a different Builder is not possible in the current implementation.

Each experiment session starts with an empty Coordinator with clean log files. The Coordinator is set to use the required scheduling algorithm for that session. The required number of Builders are then started. The Builders automatically assign themselves to the Coordinator. After the Validation System is fully running, the simulation script is started and then requests a workflow every 5 seconds. The script does this for an entire ordered set of workflows and then waits for that ordered set to fully finish. After an ordered set has finished it starts the next ordered set until all five above mentioned sets finish. After each experiment session the logs are stored and then cleaned manually. Next the Coordinator and Builders are reset and the next session is run.

The 5 second delay between workflows allows workflows to enter the queue at different times and thus have shorter or longer *queue* times. This also allows longer workflows to be assigned to Builders earlier on.

5.4.3 Longer experiment

The second experiment is a continuation of the first experiment. We anticipate the possibility that the relatively low amount of 20 workflows in each ordered set is too low to find significance in the data. To verify this problem and possibly remedy it we change the amount of workflows in each ordered set. The simulation script now requests each ordered set 5 times. This results in 100 workflows for each ordered set. The order of the set is maintained and repeated. For example: the first ordered set is run five times, so [Even][Even][Even][Even][Even] is requested during 495 seconds. Only after all workflows in that multi-set is completed will five of the next distribution ordered set be requested.

For the second experiment we choose to only run on three Builders because we focus on comparing the scheduling algorithms and not on scalability of the Builder pool. We also exclude NHEFT. NHEFT misses important features from HEFT like task transfer and heterogeneous build machines. The results from the broad test run experiment show no significant differences between HRRN and random scheduling. NHEFT is significantly slower than both HRRN and Random. NHEFT will not be tested in the second experiment because NHEFT is a custom scheduling algorithm, because it is already shows significant data in the broad tests, and because we have limited time to perform the longer test runs.

6 Results

We show the results of the KSZ test in subchapter 6.1. We then show the results gathered from our broad experiment in subchapter 6.2. Finally we show the results of the long experiment in subchapter 6.3.

6.1 Verifying the simulation

The simulated task durations are validated against the durations collected from the Guerrilla Games build farm. We input both data sets into the KSZ test. The test is performed using R. The original data set is put in as is. The simulated data are changed to account for two discrepancies. First the simulation results are six times smaller than the original data. Second there is a bug in the simulation code. This bug is described in paragraph 7.1.3. Adjustments are made to account for both these discrepancies. These adjustments make the simulated data comparable to the original data. Figure 10 through 13 show the distribution of the durations in seconds. The results of the KSZ test can be found in table 2. Only LONG_A appears to originate from its original distribution.

Table 2: KSZ test results

Original Simulation	Conversion SHORT	LinkChecker MEDIUM	Game.Main LONG_A	Tests Singleplayer LONG_B
D value	0.491	0.139	0.063	0.189
p value	< 0.01	< 0.01	0.466	< 0.01

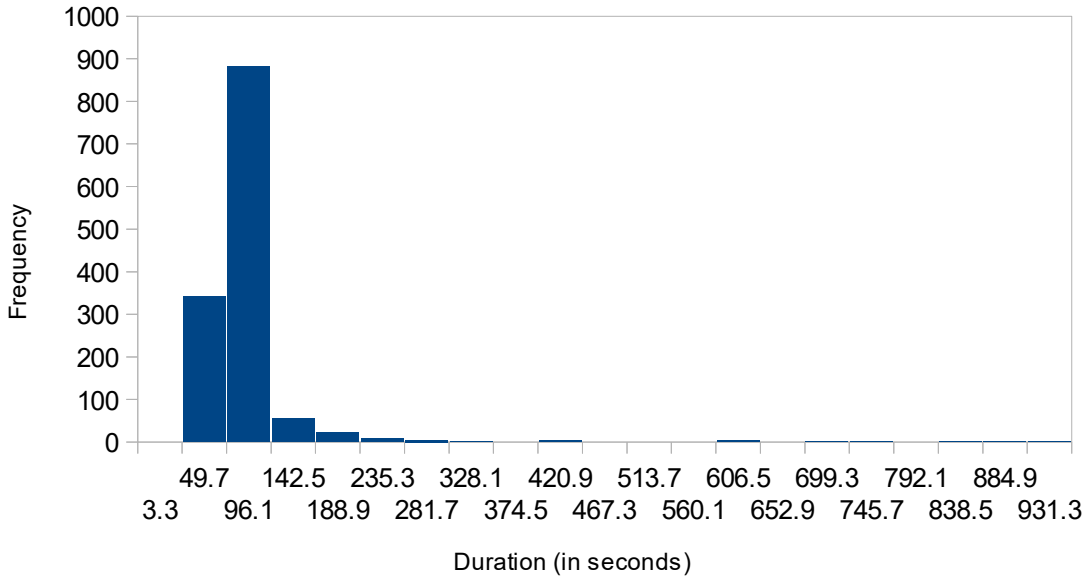

Figure 10: Short (based on Conversion)

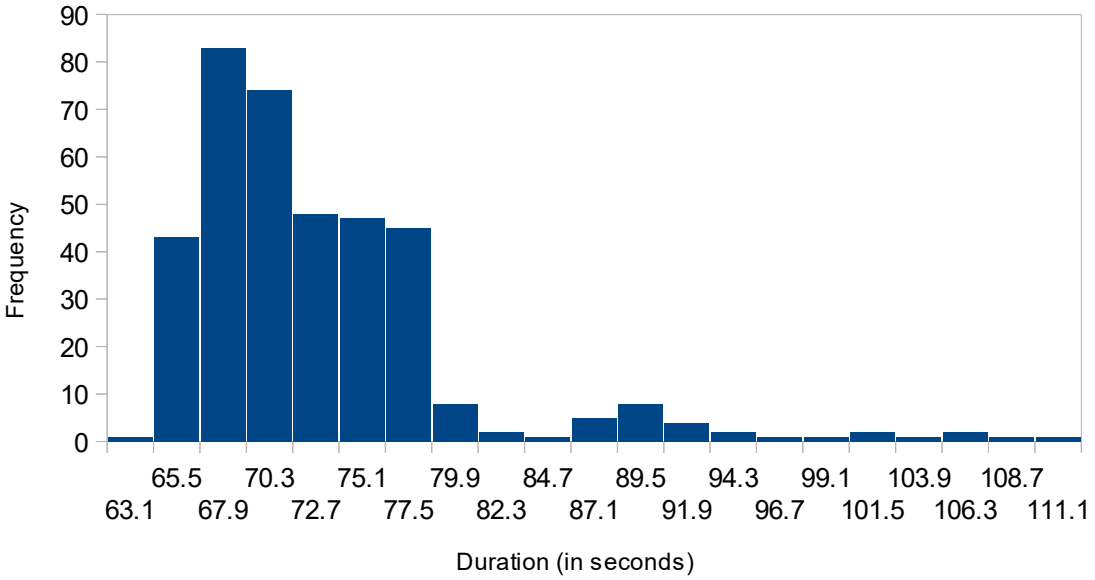

Figure 11: Medium (based on LinkChecker)

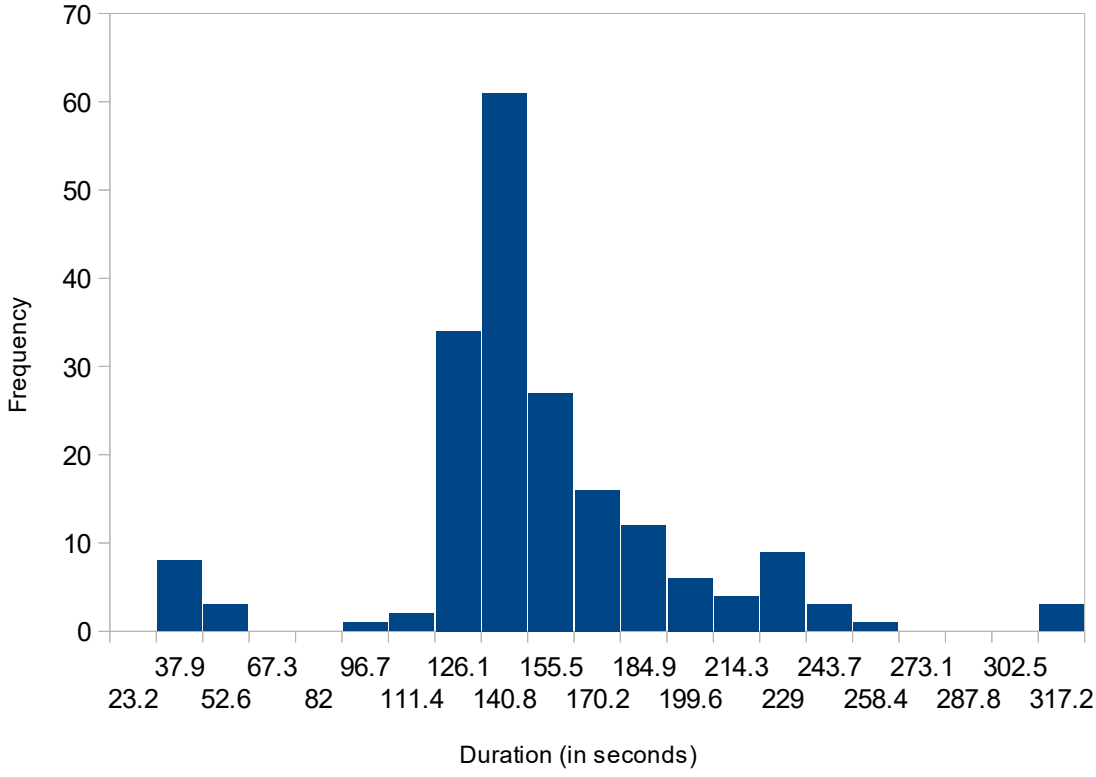

Figure 12: Long A (based on Game.Main)

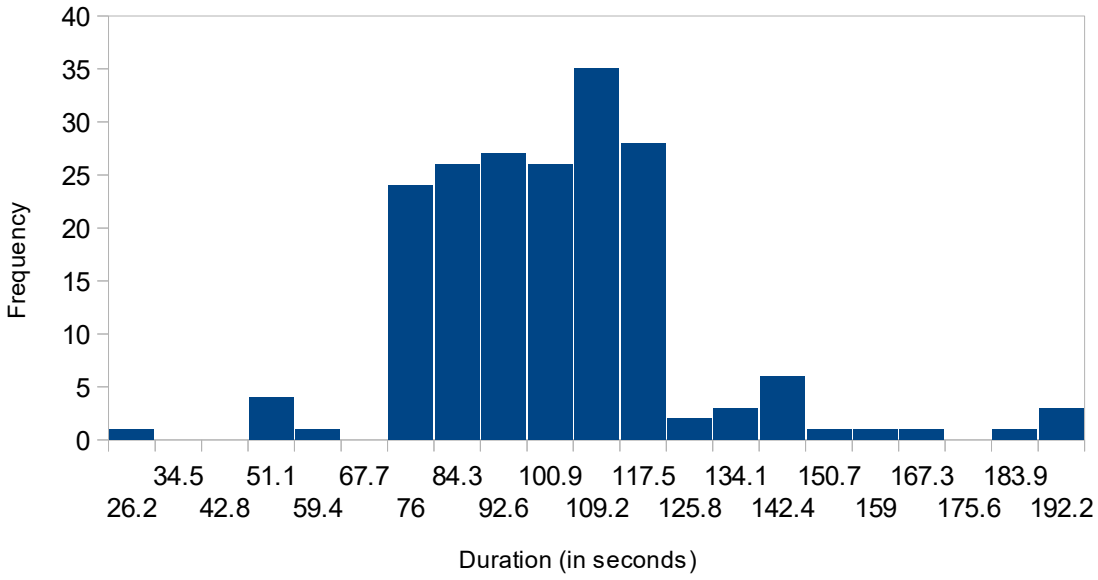

Figure 13: Long B (based on Tests Singleplayer)

6.2 Experiment 1: Broad experiment

In this experiment we measure workflow duration and the time a workflow resides in the queue without being executed.

We expect an environment in which the framework is used to have a multitude of Builders. The use of 3 Builders most closely resembles such an environment. Therefore, we focus on the results of the 3 Builder set-up. Results for 1 and 2 Builders can be found in Appendix A.

It is important to note that the average [run] time should not differ significantly between algorithms for any set of data. This must not be because the [run] time of running workflows increases the [queue] time of all other workflows in the queue. Average [queue] time increases when average [run] time is higher. When [run] time between algorithms differs significantly that data set should be dismissed.

Table 3 shows the mean and standard deviation of our results. The results are grouped by algorithm and by order of workflows. The description of each order of workflows can be found in section 5.4.2. Measurements are taken for the workflow's total running time (*run*) and the time the workflow spent in the queue while not running (*queue*). The *total* time is an addition of *run* and *queue* (*total*). Table 4 shows the measurements normalized by *run* time. To show significant differences between the various measurements we use Student's t-test. Tables 5 through 7 show these comparisons. Each table compares two algorithms. Significant differences have been highlighted. Differences with $p < 0.05$ have been made italic while differences with $p < 0.01$ have been made bold.

Table 3: Results for 3 Builders

Order	type	NHEFT mean	NHEFT std dev	HRRN mean	HRRN std dev	Random mean	Random std dev
Even	run	84.871	57.982	82.115	50.584	83.978	68.393
	queue	208.013	173.166	143.454	117.387	146.009	120.853
	total	292.885	172.574	225.569	154.347	229.987	158.820
HRRN	run	123.138	186.376	77.871	51.255	76.013	51.970
	queue	296.937	244.079	174.011	117.260	182.648	145.595
	total	420.075	254.339	251.881	132.172	258.661	144.613
HEFT	run	78.776	52.554	76.722	60.580	80.575	39.742
	queue	218.254	146.026	154.330	117.090	188.284	159.275
	total	297.030	129.065	231.051	144.223	268.859	163.056
Front	run	88.193	70.539	76.636	62.870	71.890	51.734
	queue	212.386	177.004	167.374	117.149	185.528	140.634
	total	300.579	181.641	244.009	139.319	257.418	135.357
Back	run	94.449	70.020	84.397	61.564	74.091	45.782
	queue	204.727	166.158	156.815	105.420	160.529	140.833
	total	299.175	177.700	241.212	139.024	234.620	142.021
Overall	run	93.886	99.898	79.548	56.529	77.309	51.523
	queue	228.063	183.878	159.196	113.116	172.600	140.066
	total	321.949	190.153	238.745	139.427	249.909	146.871

Table 4: Normalized results for 3 Builders

Order	type	NHEFT mean	HRRN mean	Random mean
Even	run	1.000	1.000	1.000
	queue	2.451	1.747	1.739
	total	3.451	2.747	2.739
HRRN	run	1.000	1.000	1.000
	queue	2.411	2.235	2.403
	total	3.411	3.235	3.403
HEFT	run	1.000	1.000	1.000
	queue	2.771	2.012	2.337
	total	3.771	3.012	3.337
Front	run	1.000	1.000	1.000
	queue	2.408	2.184	2.581
	total	3.408	3.184	3.581
Back	run	1.000	1.000	1.000
	queue	2.168	1.858	2.167
	total	3.168	2.858	3.167
Overall	run	1.000	1.000	1.000
	queue	2.429	2.001	2.233
	total	3.429	3.001	3.233

Table 5: t values for 3 Builders: NHEFT vs HRRN

Order	run	queue	total
Even	0.160	1.380	1.300
HRRN	1.047	*2.030*	*2.624*
HEFT	0.115	1.527	1.525
Front	0.547	0.948	1.105
Back	0.482	1.089	1.149
Overall	1.249	**3.190**	**3.529**

Table 6: t values for 3 Builders: NHEFT vs Random

Order	run	queue	total
Even	0.045	1.313	1.199
HRRN	1.089	1.798	*2.467*
HEFT	-0.122	0.620	0.606
Front	0.833	0.531	0.852
Back	1.088	0.907	1.269
Overall	1.475	*2.399*	**2.998**

Table 7: t values for 3 Builders: HRRN vs Random

Order	run	queue	total
Even	-0.098	-0.068	-0.089
HRRN	0.114	-0.207	-0.155
HEFT	-0.238	-0.768	-0.777
Front	0.261	-0.444	-0.309
Back	0.601	-0.094	0.148
Overall	0.293	-0.744	-0.551

6.3 Experiment 2: Long experiment

In the second experiment we attempt to find more differences between the algorithms by entering more workflows into the request queue. With this experiment we focus on a more specific subset of the previous experiment. We compare the HRRN and random algorithms using 3 Builders. The results are presented in the same manner as the first experiment found in subsection 6.2. Because the difference in *run* time of both algorithms is almost none we present them without normalizing.

Table 8: Long experiment results for 3 Builders

Order	type	HRRN mean	HRRN std dev	Random mean	Random std dev
Even	run	85.149	68.358	85.891	88.516
	queue	843.502	643.018	1015.629	710.868
	total	928.652	683.668	1101.520	730.444
HRRN	run	79.000	49.751	86.189	60.674
	queue	866.466	577.054	1193.805	804.131
	total	945.466	603.991	1279.994	801.512
HEFT	run	95.875	102.080	92.509	87.679
	queue	1167.155	760.327	1182.726	903.345
	total	1263.029	783.450	1275.235	914.204
Front	run	92.350	96.768	85.267	60.419
	queue	1038.052	728.640	1264.364	812.514
	total	1130.402	758.743	1349.632	802.895
Back	run	86.050	62.000	91.062	82.926
	queue	873.583	634.732	1158.438	884.011
	total	959.633	673.404	1249.500	891.009
Overall	run	87.685	78.362	88.184	76.861
	queue	957.752	681.056	1162.992	826.527
	total	1045.436	712.889	1251.176	831.407

Table 9: Normalized long experiment results for 3 Builders

Order	type	HRRN mean	Random mean
Even	run	1.000	1.000
	queue	9.906	11.825
	total	10.906	12.825
HRRN	run	1.000	1.000
	queue	10.968	13.851
	total	11.968	14.851
HEFT	run	1.000	1.000
	queue	12.174	12.785
	total	13.174	13.785
Front	run	1.000	1.000
	queue	11.240	14.828
	total	12.240	15.828
Back	run	1.000	1.000
	queue	10.152	12.721
	total	11.152	13.721
Overall	run	1.000	1.000
	queue	10.923	13.188
	total	11.923	14.188

Table 10: t values for long

Order	HRRN vs Random		
	run	queue	total
Even	-0.066	-1.796	-1.728
HRRN	-0.916	**-3.307**	**-3.333**
HEFT	0.250	-0.132	-0.101
Front	0.621	*-2.074*	*-1.985*
Back	-0.484	**-2.617**	*-2.595*
Overall	-0.102	**-4.285**	**-4.201**

7 Discussion

Before reaching conclusions based on the results we address several points of interest in our research. In this chapter we discuss possibly incorrect assumptions. We address mistakes that have been made during the experiments. We also discuss the presumed effects of these assumptions and mistakes. We explain what impact these points of interest have on our results and their scientific credibility.

7.1 Simulation

The framework's run time simulation has several flaws. First, the simulation is based on data that may not be representative of actual use. Second, the implemented distribution of workflow durations differs significantly from the data we use for the simulation input. Finally, there is a bug in the simulation code resulting in an incorrect offset in the data.

7.1.1 Distribution base

In chapter 3.5 we explain why we use data from the build farm as the basis for our simulation. In short, we can not test our framework in a fully operational professional environment. Therefore we simulate the duration of the workflow's tasks. The simulated task durations are based on the distribution of process duration measurements. Each task in our experiment is based on one of these measured processes. These processes are part of the build farm used for continuous integration at Guerrilla Games. In subchapter 5.3.1 we elaborate on why it is possible to assume the build farm process duration distribution can be used to simulate workflow duration.

The source of the distribution is not based on usage data of the local build tool but on measurements from the build farm. Therefore we can not say with certainty that the base distribution is representative of the framework's usage when the framework is being used in a professional environment. We hypothesize that the usage of the local test tool will change significantly when users have the opportunity to test remotely as well as locally. Because the actual use of the framework is uncertain there is no distribution to base the simulation on with great certainty. Using build farm data limits the conclusions we can make in our research. However, it is an efficient solution to the given problem.

7.1.2 Distribution Validity

The KSZ test is used to verify whether a set of data samples is taken from a given distribution. This base distribution is represented by a separate set of data samples. The distribution used in the simulation differs significantly from the base data. The KSZ tests were performed after the experiments. Verification of the simulation should have been performed before the experiments, not afterwards. Because of this it was not possible to create a more suitable distribution for the simulation. Although we can not scientifically state that the simulated distribution is equal to its base, both distributions do resemble each other. Figures 6 to 9 strongly resemble their distributed counterparts in figures 10 to 13 when looking at them.

The difference between the original and the simulated distribution should have little influence on the final results. The difference seems to be small when viewed as a histogram.

The uncertainty surrounding the base distribution, as described in paragraph 7.1.1, has greater influence than the small yet significant difference between the base distribution and the distribution of the simulated data. Performing the KSZ test before the experiment would have made it possible to rectify this problem. However, the impact of this mistake is not greater than the impact of the distribution base in paragraph 7.1.1.

7.1.3 Distribution bug

We compared the base and simulated data before entering it into the KSZ test. The simulated data appeared to be consistently greater than the base data. This was indeed the case. Inspection of the simulation code revealed a bug. Because of this bug the simulation generates a duration which is exactly the size of one bin too great. The bug is located in line 11 of algorithm 1. The algorithm is correct. However, in the implementation of `TimePerBin * CurrentBinIndex` we incorrectly used `TimePerBin * CurrentBinIndex + 1`.

This mistake has an impact on the total running time of the experiment which becomes longer resulting in less experiments performed. However, it was possible to account for the bug when putting the data into the KSZ test. The data samples have all increased according to their respective simulation. Because of this global increase the relative distribution between the data samples has not changed. This is why these differences have no effect on the results and any conclusions that can be taken from them.

7.2 Variable workflow durations

The simulation of workflow durations results in an extra variable in the experiments, the workflow duration. Running the experiment using predetermined and non-variable durations for each ordered set is also an option. These set durations may perfectly conform to the base distribution and would be equal for each ordered set. We chose not to use preset workflow durations because we wanted to test the framework in a realistic as possible environment. This approach can make it harder to find significant differences.

7.3 Framework completeness

The framework as used in the experiment does not make use of the specific functionality the HEFT algorithm offers, e.g.: taking performance into account when dealing with heterogeneous build machines. The case environment does not contain any heterogeneous servers. Other HEFT specific functions like transferring process data from one build machine to another were not feasible either. In a different experimental environment HEFT may have performed better than it did in our experiments.

We believe the current experimental environment, using NHEFT, is relevant to our research questions. The conclusions that can be made are specific to our case environment and should be presented as such. To determine which algorithm works best in a different case scenario this experiment should be performed in that specific environment.

7.4 NHEFT erroneous implementation

In 5.1.3 we describe the bug that resulted in high average *queue* times for the NHEFT algorithm. The bug occurs when multiple Builders are added to the Validation System. It causes

the shortest workflows to remain in the queue until all other workflows have completed. This leads to very long *queue* times for those workflows and raises the average *queue* time significantly. NHEFT has significant longer *run* times compared to the other algorithms. These differences are likely caused by this bug. No conclusions about NHEFT can be made based on the experiments. Consequently we can not make any assumptions about the effectiveness of the HEFT algorithm.

8 Conclusion

In subchapter 8.1 we answer the first research question. We answer the second research question in subchapter 8.2.

8.1 Framework

With the first research question we ask how we can make a framework which can facilitate different scheduling algorithms whilst adhering to the requirements in chapter 1.2. Here we present how the Validation System contains the answer to that question.

8.1.1 Accessibility to algorithms

The implementation of the Validation System contains three different scheduling algorithms. These algorithms require a different type and variety of data which are available in the Validation Coordinator. HEFT especially requires more functions to for example transfer a task from one Builder to another. The functions required to do so are available in the framework although some of this functionality has not been implemented and tested. Figure 5 in chapter 4 shows all the required functions available for both the NHEFT and HEFT algorithms to run. The data and functions required for HRRN and random scheduling to run are all present in the data and functions required for NHEFT/HEFT. Requirement 1 has been met for the implemented scheduling algorithms.

8.1.2 Scalable Builder pool

The Validation Coordinator adheres to Requirement 2 by assigning available Builders to a Builder pool. Builders can be added to and removed from the Builder pool during runtime. When a Builder is removed from the Builder pool it is necessary to reschedule. If the removed Builder was actively running a task this task may need to be restarted on or transferred to another Builder. The task that was last run on a removed Builder may have produced changes required by other tasks in its workflow. These changes need to be transferred to the newly assigned Builder as well. This is a very costly process and should be avoided within our case environment. As long as at least one Builder remains assigned to the Validation Coordinator the system can stay active and functioning. Although Requirement 2 is met it is advisable to not remove Builders during runtime unless the entire workflow is run again from the start on a different Builder.

8.1.3 Peak usage

It is difficult to show that Requirement 3 has been met. Stressing the framework to a breaking point is not feasible during our experiments. Neither is proving that this requirement is met for all possible usage scenarios. We approach Requirement 3 by investigating the bottlenecks in our current implementation. Creating a peak usage environment would be difficult given the limited amount of build machines available for the experiment. We can safely say the requirement is met if the framework can clearly deal with an exaggerated worst case scenario for the identified bottlenecks.

TLocalBuild can cause a potential bottleneck. When TLocalBuild crashes it may leave the TCP connection it uses to receive feedback open. When this happens while a Builder is

performing a task using that TCP connection, the Builder may enter an infinite loop. We prevent this using TCP timeout functionality in Python.

Peak load will be caught by the Validation Coordinator before it reaches the Builders. All superfluous requests are stored in the RequestQueue. A high amount of requests will result in long *queue* times. This will not affect the Builders however.

The network at Guerrilla Games is state-of-the-art so we assume that this will not be a bottleneck. We also assume the RPC system is able to deal with many connection calls over a short amount of time. These assumptions have held up during the experiments but should be regarded with caution.

The size of the request queue might be a bottleneck. We consider a realistic worst case scenario. A large game studio realistically does not consist of more than 500 on-site developers. If all these developers would simultaneously request several workflows within a few minutes this would account for no more than 2000 requests. The data stored in the request queue are very small. Even if the Validation Coordinator would run on a work machine with only 16GB of RAM the machine would not run out of memory.

Another possible bottleneck could arise when using an inefficient scheduling algorithm. This is not the case with NHEFT, HRRN, nor random scheduling. No bottlenecks were encountered during the experiments and the analysis of possible bottlenecks indicates high robustness. We consider Requirement 3 met within our implementation and within our case environment given the network and RPC assumptions. However, we can not conclude that the framework will continue to perform when used on a larger scale.

8.1.4 Reliability of data

Requirement 4 is met by using Perforce. It is possible to use other version control software to achieve the same goals. The local source repository on a build machine is cleaned using Perforce commands. The clean is performed before the newly assigned task is executed. When the new task is part of the same workflow as the previous task, no cleaning is done. Perforce reverts any changes that were made since the last clean and, after cleaning, synchronizes the local repository to the latest stable version. It is very important to set up all tasks such that all files that can be changed are checked out in Perforce. Checked out means that the files are known and monitored by Perforce. This results in the version control software knowing which files to revert. This is set up correctly in our case environment.

8.1.5 Front-end tool

Paragraph 4.2 shows the technical description of TLocalBuild. This front-end tool presents output for each task in a separate tab. TLocalBuild fully fulfills Requirement 5.

8.1.6 Requirements conclusion

We have shown that all Requirements have been met within our current implementation of the framework given the assumptions for Requirement 3. In the technical description in chapter 4 we explain how the framework is built. This answers the first research question.

However, there are scenarios in which the Validation System can be used where these Requirements may not be met. We give several examples: Requirement 3 may not be met when the framework receives extremely high amounts of requests. Requirement 1 is not

met when algorithms require information on changes in the workflow request's changelist. The Validation System does not force the use of a front-end tool. Therefore, not every implementation of the framework may meet Requirement 5.

8.2 Algorithms

In the second research question we ask what scheduling algorithm gives the average user the shortest average waiting time. We discuss the overall performance of the algorithms and the results of the longer tests to see whether a difference can be found between the three scheduling algorithms.

We first discuss the experiments set-ups with 1 and 2 Builders. The NHEFT bug does not occur in the 1 Builder set-up. As expected, HRRN is significantly faster than random scheduling ($p < 0.01$). We expect NHEFT to show similar results. However, this is not the case. No significant differences can be found between NHEFT and both HRRN and random scheduling. HRRN does seem to show lower average queue times. For the 2 Builder set-up NHEFT performs almost equal to random scheduling while HRRN has significantly shorter average waiting times ($p < 0.05$). The lower performance by NHEFT can be explained by the NHEFT bug. We base our conclusions upon results that more strongly resemble actual use of the framework. Therefore we base the conclusions on the 3 Builder set-up.

The results for the broad experiment with 3 Builders show that NHEFT has significantly longer *queue* times than the HRRN ($p < 0.01$) and random scheduling ($p < 0.05$) algorithms. There is an indication of longer running times for NHEFT. However, after normalizing the data for *run* duration, table 4 clearly shows longer *queue* times for NHEFT. The difference between HRRN and random scheduling is minimal.

The results for the longer experiment show that HRRN has significantly shorter ($p < 0.01$) *queue* times than random scheduling. From this data we make the preliminary conclusion that HRRN produces shorter average wait times than random scheduling. We also conclude that NHEFT produces longer average wait times than random scheduling.

Our preliminary conclusion is rejected for the NHEFT algorithm. We can only conclude that the algorithm was improperly implemented. Consequently we can not make any assumptions about the HEFT algorithm on which NHEFT is based. However, the difference between the HRRN and random scheduling algorithms is confirmed. In our case environment the HRRN scheduling algorithm gives the user the shortest average waiting time. These conclusions must be considered with caution as described in chapter 7. Different environments and implementations may produce different results.

9 Future Work

9.1 Ordered sets

Of the three scheduling algorithms we compare, HRRN produces the shortest average waiting time overall. However, the results show that this is not the case for all ordered sets. For the *HEFT* specific ordered set HRRN performs equally to random scheduling in the long experiment. It may be important to analyze the relevance of the order in which requests enter the RequestQueue.

The data collected in the experiments show both the order in which requests are added to the RequestQueue and the order in which they are assigned. However, they do not show based on what data the algorithms choose which request to assign to a Builder. The weights on which prioritization is based are not stored. These data may provide important information of the inner workings of the scheduling algorithms.

Knowledge of the internal data of scheduling algorithms may provide insight into the reasons why certain algorithms perform better than others. It may help in finding more suitable scheduling algorithms. It may also help in improving business processes such that the algorithms can perform better. E.g.: Enforcing a rule that long requests can only be submitted after 17:00.

9.2 Framework expansion

The framework in its current state does not have all the features it could potentially have. In subchapter 4.7 we discuss several of these features. Implementation of these features could make the Validation System more effective and better suited to different environments. Implementing these features makes further research possible.

The framework could be implemented and tested in different environments. Other AAA game companies would be a good starting point. Especially if these other companies have a heterogeneous set of build machines available. The similar testing goals in those companies would be a good match for our framework. To test wider use of the framework it can be interesting to test it in large non game related technology companies, e.g.: ASML. The size of this type of organization can be multitudes larger than AAA game companies. This can present new technical issues.

The framework requires very little resources and it is possible to run it using only one build machine. Because of this the framework can be implemented at smaller game companies or for example teams of developers who are located all over the world.

9.3 Further algorithm comparison

The different case environments and the possible additions to the framework discussed in 9.2 can present good opportunities for the implementation of other scheduling algorithms. There are many algorithms available which could be better suited for the technical- and business structure of the environment the framework is implemented in. Comparing the average waiting time for those algorithms can give a clear picture of company-wide efficiency of the algorithm.

9.4 Integration in external tools

In the previous subchapters we discussed expanding the Validation System and testing different scheduling algorithms. A different approach could be integrating the Validation System in other tools to increase their performance. We illustrate this possibility with an example.

T3 is a random testing tool for Java created by Prasetya [14]. T3 entered in the 2013 testing tool contest FITTEST [17]. T3 randomly generates test cases for a given Java class. When the class to be tested increases in complexity, so does the amount of possible tests T3 can perform. T3 could perform its required tests more quickly if it distributes its tasks to remote machines. It could perform much more tests in the same amount of time it currently takes to execute. The Validation System is a light-weight solution which makes it more viable to implement in such a tool as T3 than heavier tools like Jenkins.

References

[1] D. Anderson. Boinc: A system for public-resource computing and storage. *Fifth IEEE/ACM International Workshop on Grid Computing*, January 2004.

[2] D. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. *First International Conference on e-Science and Grid Computing (e-Science'05)*, 2005.

[3] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *Softw: Pract. Exper. Software: Practice and Experience*, pages 1437–1466, 2002.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, pages 599–616, 2008.

[5] F. Costa, L. Silva, G. Fedak, and I. Kelley. Optimizing the data distribution layer of boinc with bittorrent. *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[6] R. Figueiredo, P. Dinda, and J. Fortes. A case for grid computing on virtual machines. *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, 2003.

[7] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. *2008 Grid Computing Environments Workshop*, 2008.

[8] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works http://www.thoughtworks. com/Continuous Integration. pdf*, 2006.

[9] S. Imai, T. Chestna, and C. A. Varela. Elastic scalable cloud computing using application-level migration. *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, 2012.

[10] B. Jacob. *Introduction to Grid Computing.* IBM, International Technical Support Organization, 2005.

[11] A. Law. *Simulation modeling and analysis*, chapter 5. McGraw-Hill Higher Education, 2006.

[12] G. Marsaglia, W. W. Tsang, and J. Wang. Evaluating kolmogorv's distribution. *Journal of Statistical Software*, 2003.

[13] Perforce. Perforce case studies and testimonials. `https://www.perforce.com/our-customers`, 2016.

[14] W. B. Prasetya. T3, a combinator-based random testing tool for java: Benchmarking. *Future Internet Testing*, pages 101–110, 2013.

[15] M. Rocklin. Heterogeneous earliest finish time. `https://github.com/mrocklin/heft`, 2015.

[16] S. Stolberg. Enabling agile testing through continuous integration. *2009 Agile Conference*, 2009.

[17] T. Vos, P. Tonella, W. Prasetya, P. M. Kruse, A. Bagnato, M. Harman, and O. Shehory. Fittest: A new continuous and automated testing process for future internet applications. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 407–410. IEEE, 2014.

[18] Z. Xiao, W. Song, and Q. Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Trans. Parallel Distrib. Syst. IEEE Transactions on Parallel and Distributed Systems*, 2013.

[19] J. Yu, R. Buyya, and K. Ramamohanarao. *Metaheuristics for Scheduling in Distributed Computing Environments*, chapter Workflow Scheduling Algorithms for Grid Computing, pages 173–214. Springer, 2008.

[20] P.-F. Zhang, X.-Y. Li, and L. Ma. Grid computing based on game optimization theory for networks scheduling. *Journal of Networks JNW*, 2014.

A Builder tables

Table 11: results for 1 Builder

Order	type	NHEFT mean	NHEFT std dev	HRRN mean	HRRN std dev	Random mean	Random std dev
Even	run	104.850	128.882	78.602	61.317	81.501	62.305
	queue	1122.319	733.145	517.995	356.538	674.200	498.772
	total	1227.169	706.576	596.597	400.237	755.701	510.581
HRRN	run	83.955	77.895	92.458	57.504	113.709	129.486
	queue	674.514	532.408	764.483	511.427	1272.819	649.364
	total	758.468	548.114	856.941	526.868	1386.528	605.295
HEFT	run	81.164	62.846	88.768	52.978	83.017	51.220
	queue	845.111	513.859	687.848	433.112	896.721	487.311
	total	926.274	496.229	776.616	460.736	979.738	461.523
Front	run	86.919	66.805	94.974	75.656	108.323	152.054
	queue	857.904	423.179	851.350	442.553	1176.885	686.547
	total	944.823	409.901	946.324	448.989	1285.208	656.246
Back	run	97.745	74.210	97.578	80.322	83.640	53.521
	queue	714.274	511.141	673.695	463.237	722.674	511.845
	total	812.019	545.457	771.273	507.893	806.314	516.655
Overall	run	90.927	84.290	90.476	65.389	94.038	98.239
	queue	842.824	563.591	699.074	449.106	948.660	610.588
	total	933.751	562.768	789.550	475.878	1042.698	599.667

Table 12: Normalized results for 1 Builder

Order	type	NHEFT mean	HRRN mean	Random mean
Even	run	1.000	1.000	1.000
	queue	10.704	6.590	8.272
	total	11.704	7.590	9.272
HRRN	run	1.000	1.000	1.000
	queue	8.034	8.268	11.194
	total	9.034	9.268	12.194
HEFT	run	1.000	1.000	1.000
	queue	10.412	7.749	10.802
	total	11.412	8.749	11.802
Front	run	1.000	1.000	1.000
	queue	9.870	8.964	10.865
	total	10.870	9.964	11.865
Back	run	1.000	1.000	1.000
	queue	7.308	6.904	8.640
	total	8.308	7.904	9.640
Overall	run	1.000	1.000	1.000
	queue	9.269	7.727	10.088
	total	10.269	8.727	11.088

Table 13: t values for 1 Builder: NHEFT vs HRRN

Order	run	queue	total
Even	0.822	**3.315**	**3.473**
HRRN	-0.393	-0.545	-0.579
HEFT	-0.414	1.047	0.988
Front	-0.357	0.048	-0.011
Back	0.007	0.263	0.244
Overall	0.042	1.995	1.957

Table 14: t values for 1 Builder: NHEFT vs Random

Order	run	queue	total
Even	0.729	*2.260*	*2.419*
HRRN	-0.881	**-3.186**	**-3.440**
HEFT	-0.102	-0.326	-0.353
Front	-0.576	-1.769	-1.967
Back	0.689	-0.052	0.034
Overall	-0.240	-1.274	-1.325

Table 15: t values for 1 Builder: HRRN vs Random

Order	run	queue	total
Even	-0.148	-1.139	-1.097
HRRN	-0.671	**-2.750**	**-2.951**
HEFT	0.349	-1.433	-1.393
Front	-0.352	-1.782	-1.906
Back	0.646	-0.317	-0.216
Overall	-0.302	**-3.293**	**-3.307**

Table 16: results for 2 Builders

		NHEFT		HRRN		Random	
Order	type	mean	std dev	mean	std dev	mean	std dev
Even	run	81.201	52.260	82.705	51.683	112.455	104.090
	queue	331.908	266.345	269.303	184.833	382.741	357.395
	total	413.109	264.365	352.008	214.591	495.197	383.756
HRRN	run	90.355	83.738	84.855	87.589	75.304	50.664
	queue	361.431	293.620	222.500	160.388	268.542	199.360
	total	451.786	298.365	307.356	224.680	343.846	217.256
HEFT	run	75.911	53.762	75.512	36.134	75.611	51.819
	queue	307.441	228.238	251.542	180.546	271.922	232.984
	total	383.352	229.448	327.054	203.948	347.533	245.129
Front	run	125.423	190.191	85.770	66.978	92.822	71.504
	queue	455.632	300.585	335.950	205.257	413.612	276.563
	total	581.054	334.401	421.720	218.351	506.434	268.023
Back	run	80.276	47.323	94.035	90.216	91.180	55.579
	queue	303.192	239.562	249.471	171.317	347.333	282.741
	total	383.468	244.323	343.506	232.903	438.513	288.445
Overall	run	90.633	100.621	84.575	68.515	89.474	69.645
	queue	351.921	267.702	265.753	181.489	336.830	275.706
	total	442.554	281.070	350.329	218.155	426.305	289.002

Table 17: Normalized results for 2 Builders

Order	type	NHEFT mean	HRRN mean	Random mean
Even	run	1.000	1.000	1.000
	queue	4.088	3.256	3.404
	total	5.088	4.256	4.404
HRRN	run	1.000	1.000	1.000
	queue	4.000	2.622	3.566
	total	5.000	3.622	4.566
HEFT	run	1.000	1.000	1.000
	queue	4.050	3.331	3.596
	total	5.050	4.331	4.596
Front	run	1.000	1.000	1.000
	queue	3.633	3.917	4.456
	total	4.633	4.917	5.456
Back	run	1.000	1.000	1.000
	queue	3.777	2.653	3.809
	total	4.777	3.653	4.809
Overall	run	1.000	1.000	1.000
	queue	3.883	3.142	3.765
	total	4.883	4.142	4.765

Table 18: t values for 2 Builders: NHEFT vs HRRN

Even	-0.092	0.864	0.803
HRRN	0.203	1.857	1.729
HEFT	0.028	0.859	0.820
Front	0.879	1.471	1.784
Back	-0.604	0.816	0.529
Overall	0.498	*2.664*	*2.592*

Table 19: t values for 2 Builders: NHEFT vs Random

Order	run	queue	total
Even	-1.200	-0.510	-0.788
HRRN	0.688	1.170	1.308
HEFT	0.018	0.487	0.477
Front	0.718	0.460	0.779
Back	-0.668	-0.533	-0.651
Overall	0.095	0.393	0.403

Table 20: t values for 2 Builders: HRRN vs Random

Order	run	queue	total
Even	-1.145	-1.261	-1.456
HRRN	0.422	-0.805	-0.522
HEFT	-0.007	-0.309	-0.287
Front	-0.322	-1.008	-1.096
Back	0.121	-1.324	-1.146
Overall	-0.501	*-2.153*	*-2.098*

Table 21: Average of queue time normalized by runtime for the broad experiment (seconds)

Algorithm	Count	Even	HRRN	HEFT	Front	Back
HEFT	1	18.460	9.721	14.408	13.342	9.159
HEFT	2	5.066	5.812	5.472	7.159	4.213
HEFT	3	3.343	4.984	3.590	3.188	2.768
HRRN	1	7.219	9.696	8.820	12.040	7.937
HRRN	2	3.720	3.175	3.344	5.069	3.290
HRRN	3	1.821	2.684	2.345	2.802	2.232
Random	1	10.062	17.413	14.755	18.000	10.726
Random	2	5.490	4.319	4.538	6.301	4.297
Random	3	2.056	3.124	2.710	3.314	2.533

Table 22: Average of queue time normalized by runtime for the long experiment (seconds)

Algorithm	Even	HRRN	HEFT	Front	Back
HRRN	11.004	12.141	14.771	13.750	13.972
Random	15.429	17.691	17.313	19.530	17.051