ZINC

not another conference

2020 Zooming Innovation in Consumer Technologies Conference (ZINC)

Online, 26-27 May 2020

ISBN: 978-1-7281-8259-9

IEEE Catalog Number: CFP20ZIN-ART

www.GoZinc.org

Design and Implementation of Cluster Based Parallel System for Software Testing

Jovan Milojković
Department of Computer Science
Faculty of Electronic Engineering
Nis, Serbia
jovan.milojkovic@elfak.ni.ac.rs

Vladimir Ćirić
Department of Computer Science
Faculty of Electronic Engineering
Nis, Serbia
vladimir.ciric@elfak.ni.ac.rs

Dejan Rančić
Department of Computer Science
Faculty of Electronic Engineering
Nis, Serbia
dejan.rancic@elfak.ni.ac.rs

***Abstract*— In a new era of technology, reducing process execution time using multiple hardware components is a trend. In order to reduce cycle time in software delivery smarter deployment pipeline strategies should be considered, as well as parallel builds and parallel testing strategies. In this paper we propose a system that parallelizes software testing within the deployment pipeline. The proposed system increases hardware used for the process execution and reduces time needed for testing of a developed software product. Docker will be used for the system implementation. Parallelization will be achieved at process and container levels. Furthermore, in this paper we show how parallelization using containers and processes affects time needed for test execution of a developed software product.**

Keywords*— container, Docker, parallel systems, software testing, deployment pipeline, continuous integration

I. INTRODUCTION

In this part of the paper some basic concepts and terms will be defined. Cycle time is a unit of time which passes when a decision is made that a change should be introduced in a software product until that change is implemented in the production. The deployment pipeline process tries to reduce this time. Deployment pipeline is automated implementation of build, test, deploy and release of software product [1]. Deployment pipeline presents a set of stages through which the system that is being developed goes through.

Continuous Integration is a methodology which states that after every change to a product, the product should go through Deployment Pipeline process [2]. If there is an error in any of the phases of Deployment Pipeline, team which made the error should resolve it as soon as possible. Continuous Integration can be implemented on a team as a set of rules which every team member should respect.

Product testing is a part of Verification and Validation process [3]. Validation is a process in which the product is checked that it does what customer imagined, while Verification is a process which checks if the product is made according to its requirements specification.

Product testing can be done manually or it can be done automatically. In manual product testing, a person manually goes through checklist from documentation and checks whether or not the point in checklist is implemented and is functioning properly. In automated testing, the test cases are run over a product and their results are gathered at the end.

There are many tools that encourage automated software testing [4]. The most widely used tools include Bamboo, Jenkins, CruiseControl, Sysphus, etc. However, they do not use parallelization techniques while running tests for software products. In the methodology of Continuous Integration and Deployment Pipeline, which mentioned tools use, everything is sequential.

In this paper the parallelization of automated testing in deployment pipeline is proposed. A system for parallel software testing will be implemented on cloud using containers, namely Docker. Parallelization will be achieved at process and container levels. Parallelization techniques used in the proposed system will be described further and their comparison will be given. The architecture of the system will be evaluated. The implementation results that show how parallelization using containers and processes affects time needed for test execution of a developed software product will be given. The price paid for reducing the testing time is increased hardware usage.

II. DEPLOYMENT PIPELINE AND SYSTEM TESTING

In order to be able to design a system for parallel software testing, in this section Deployment Pipeline and some general software testing methodologies and their usage will be discussed.

As mentioned earlier, software testing [3] is a part of Verification and Validation process. The main goal of a test [5] is to find the errors in the system that is being tested. While running tests on some product, we can make descriptions of how a certain part of system, or whole system performs.

In general, there are three testing strategies: White Box, Black Box and Gray Box [6], [7]. White Box testing is a testing in which we write tests knowing the internal structure of the product we are testing. Black Box testing examines the functionalities of the product, without the knowledge of how the product is implemented. Gray Box testing is strategy in which both Black Box and White Box strategies are combined. In proposed system, only functional Black Box tests are written.

While writing automated tests, tests cases should be provided. Each test case should then be written as a program and run to test the product. One run of test case consists of starting a software product, getting it into a desired state (or getting some of its components in some desired state), running a set of commands on the desired state, gathering results and checking

978-1-7281-8259-9/20/$31.00 ©2020 IEEE

the gathered results with the test case results. It can be observed that one test case is fully sequential, so no parallelism should be introduced here.

Deployment Pipeline represents automated manifestation of getting the product from some version control system through build and test phase. At the end the product can be released. It can be said that Deployment Pipeline is one instance of the given process. The Deployment Pipeline process and its phases are shown in Fig. 1. They depend on the organization in which the process is implemented. This means that the process can be divided in more phases or merged in less, but the view in Fig. 1 gives a general picture of it. All phases do not need to be run every time a change happens, but the Continuous Integration methodology states that at every stage whole product should go through Deployment Pipeline. Several Deployment Pipelines can be run in parallel so that several versions of product can be tested.

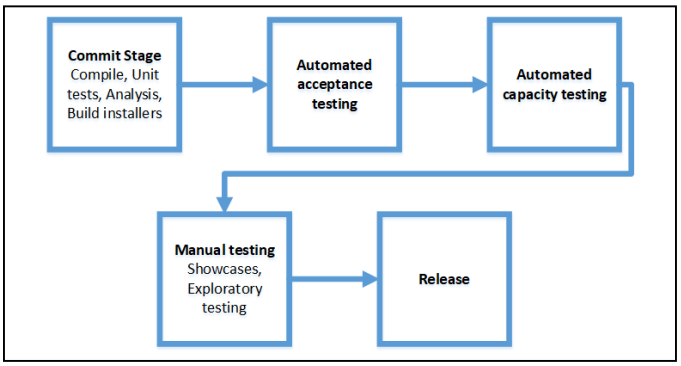

Fig. 1. Deployment Pipeline Process

It can be observed that the process consists of 5 phases. In the first phase, the Commit stage, the change in the product is made, and the process of building the software product is initialized. Code analyzer tools are run. At the end of the stage a set of unit tests are run. Running unit tests can be another phase. However, in Fig. 1 they are all integrated into the Commit Stage. The reason is because they should be run every time a change happens in the underlying code. If the product passes the initial stage (we say that the product passes the stage if and only if all tests did not recover any bugs), then Automated Acceptance Testing starts (Fig. 1). If the product passes this stage then Automated Capacity testing begins, etc.

In this paper we deal with the stages where automated tests are executed, because they can be executed in parallel. This means that testing with automated tests in one deployment pipeline process can be parallelized, and therefore, its run time can be shorter.

There are many tools for software testing, like Bamboo, Jenkins, CruiseControl, Sysphus, etc [4]. All this tools can run several deployment pipeline processes from Fig. 1 in parallel, but as a whole. However, they do not use parallelization techniques while running tests in one deployment pipeline instance. When a branch gets through the process, the process of automated testing of that branch is linear. Furthermore, in many cases branch is always build from the begging. The system proposed here parallelizes the testing of a product inside one branch.

In order to parallelize a deployment pipeline instance, in the following section we will examine the possible parallelization strategies.

III. Parallelization Techniques

We propose the processes and containers as parallelization techniques. Both techniques ensure that several test cases can be run in parallel. In both techniques whole test suit will be divided into disjoint test suits, which can be run in parallel.

A. *Processes*

Parallelization using processes consists of dividing the test cases into groups and running them in parallel throughout different processes. One process can execute a group of given tests in sequence, which enables execution of one test group per process, where several processes can be executed in parallel.

For this case a scheduler of tests is required, who will distribute the test cases to different parallel processes. One scheduler should control several processes. Processes can be run in parallel on one node (multicore) or on several nodes. Depending on number of nodes in the system, the scheduler architecture and its responsibilities varies.

Involving containers for process control in one node only the scheduler can be simplified.

B. *Containers*

Docker Containers can be used to encapsulate several processes along with accompanied scheduler and executed there on one node. Involving containers simplifies parallelization on cluster of nodes and inherently gives another level of parallelization. In order to be able to distribute tests to containers a master scheduler should be involved. The master scheduler can pass environment variables, so that every container in its configuration knows the total number of containers in the system and which tests it should run.

The tests cases that are run on containers represent the disjoint group of tests. Every container has its own test cases. Furthermore, in each container parallelization technique with processes is used, so the group of tests that are run inside a container are further parallelized with processes, giving us two levels of parallelization: containers and processes.

IV. Design and Implementation of The Proposed System

In this section the architecture of the proposed system and the design of Docker container image will be discussed.

The system consists of several components as shown in Fig. 2. The system is divided in 3 parts: Application that is being tested, Test Framework (library of classes and functions for tests) and the Scheduler (scheduler for process level parallel testing). For the sake of illustration, we choose to test the standard calculator application (Fig. 2). Architecture shown in Fig. 2 enables implementation using previously mentioned process parallelism technique. With the containerization of whole architecture from Fig. 2, the container techniques parallelization is applied.

In order to evaluate the system every component of the system has a logger component which logs the time of the

system when the action is performed. Containers have their start and end time, and components inside the containers log their time as well. After the container finishes its job, the logs are gathered. All logs display time in nanoseconds.

A. *Application that is being tested*

Application that is being tested for is standard calculator application. It has some of the functionalities of the calculator applications we see broad wide. The TCP connection is used to communicate with the calculator, and to receive the feedback (Fig. 2).

B. *Test Framework*

The Test framework consists of several components. We will show this part of the system by using Bottom-Up approach.

In the bottom there is a component that is used to encapsulate calculator application for the testing purposes (Fig. 2). It can send and receive data from calculator application, start it and end it. Command class represents a generalization of commands that can be executed on calculator.

Next is a test generalization and abstraction (Fig. 2). *Test* consist of calculator application that will be started and commands that can be sent to it. Furthermore, it contains an abstraction of test workflow: start application, send commands, gather results and check results. *TestCase* specializes *Test* and contains one test case that should be run. In Fig. 2 *TestCase* is shown as one component, and there is a component called *Test Cases* which consists of all *TestCases* written. The *Test Cases* represent all tests that will be run while testing the product. A configuration of all test cases is in configuration file *tests.config*. This file contains all the test cases that will be run in one deployment pipeline.

C. *Python Scheduler*

The Python scheduler makes parallelization possible. This component splits tests depending on the environment configuration, and run test cases in parallel.

It consists of the *Server* which splits the tests and knows which processes should run which group of the tests (Fig. 2). The *Server* knows about the tests because it has access to *tests.config* file. *Server* can be additionally fine-tuned by using the environment configuration. Server has its handlers, where each handler is a thread, and one handler handles one process. One process is implemented by one client (Fig. 2), which connects to the server to gather the configuration and group of tests that it should run. One client run in sequence test cases that it got from server, as described in the previous section.

Clients and server are started with the script. Only one server exists, but the number of clients dictates the number of parallel processes which are testing the software product in parallel.

D. *Containerisation*

The image that will be run in the Docker container is the image of the whole system from Fig. 2 giving us another level of parallelism. Through the environment configuration the parameters of the process parallelization are being communicated to the container. Here the tests cases are split to groups and each group of tests goes to one container. Furthermore, each split group is divided to processes in containers so that the process parallelization is obtained inside each container. When a container starts, it gets the configuration from environment and runs the *Server* and *Clients* from the *Python Scheduler*. In each container the branch that is given to it can be built, but if it is already built, build files can be used for testing the product. The current implementation uses *Docker Swarm* [8] scheduler for the scheduling of the containers to cluster nodes.

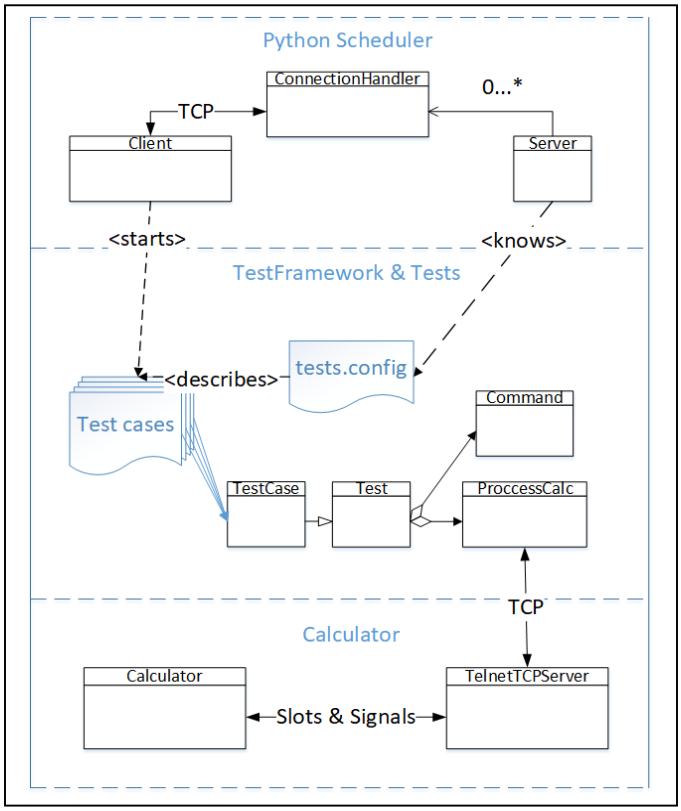

Fig. 2. Architecture of the proposed system for parallel software product testing

V. Implementation Results

The proposed system was implemented and evaluated on a cluster of several nodes. The cluster consisted of 3 physical servers with 6 physical processors (Intel Xeon CPU_E5603 @ 1.6 GHz). 24 logical processors, and 334 GB of RAM memory in total. The operating system run on physical machines is VMware ESXi 5.5.0. Docker and Swarm are installed on 10 virtual machines. All virtual machines were the same. They were running Ubuntu 16.04. They had 2 logical CPUs and 8GB of RAM each.

On all virtual machines NTP [9] was installed so that the time measurement could be done properly. Regarding containers, no resource constraints were used on them, so they can use the resources as much as they need.

Three types of measurements were made. The measurements are labeled using the following pattern: "X_N,

X_K, X_C, X_T". X_N is the number of nodes in the measurement, X_K is the number of containers, X_C is the number of client processes inside the container, while X_T stands for the number of tests run by a container. "X" in the measurement means that that parameter varies throughout the evaluation. The *y* axis in all given results stands for execution time in nanoseconds [*ns*], while *x* axis represents different measurement patterns [*p*].

The first performed evaluation for a measurement pattern "10N, 1K, X_C, 72T" and it is shown in Fig. 3. Here the parallelization technique using processes is shown. It can be observed that the time needed to test the whole test cases of 72 tests gradually declines while the number of processes in system increases. In this measurement pattern the number of clients (X_C) in measurements goes: 1C, 2C, 3C, 4C, 6C, 8C, 9C.

The second evaluation is a measurement pattern "10N, X_K, 1C, X_T". This evaluation can be seen in Fig. 4. Here the total number of test cases in every measurement is constant 72. In this measurement the parallelization using containers only is displayed. The measurement parameters regarding containers and tests "K/T" go: 1K/72T, 2K/36T, 3K/24T, 4K/18T, 6K/12T, 8K/3T and 9K/8T.

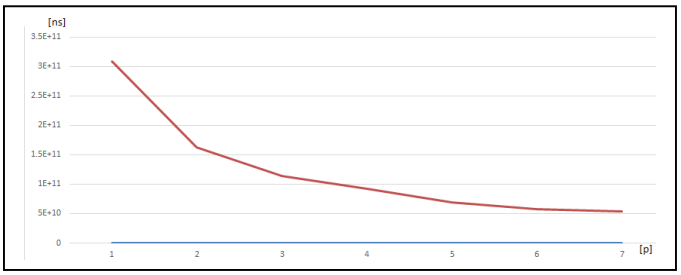

Fig. 3. Measurement "10N, 1K, X_C, 72T"

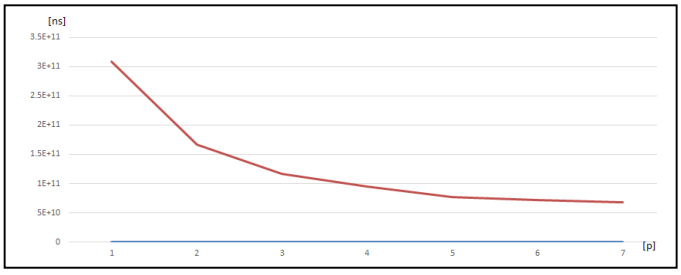

Fig. 4. Measurement "10N, X_K, 1C, X_T"

The third evaluation is a measurement pattern "10N, X_K, X_C, X_T" and it is shown in Fig. 5. In this measurement the number of test cases is constant 72. The parallelization with clients throughout this measurement is gradually replaced with the parallelization by containers. The measurement parameters "K/C/T" go: 1K/24C/72T, 2K/12C/36T, 3K/8C/24T, 4K/6C/24T, 6K/4C/12T, 8K/3C/9T, 12K/2C/6T and 24K/1C/3T. It can be observed that in this measurement one client always runs 3 test cases in sequence.

In the first evaluation from Fig. 3 it can be noticed that saturation is reached when more than 4 clients are run in one container. This is related with the number of cores per processor. In the third evaluation from Fig. 5, it can be observed that it is better to use more clients than more containers, but only to some point. The first and the last measurement are significant because they show that it is better to have less containers and more processes inside them. Furthermore, in this evaluation the best result was the measurement with id 3, which shows that parallelization is the most productive when both parallelization techniques are being used, with the respect of the cluster configuration.

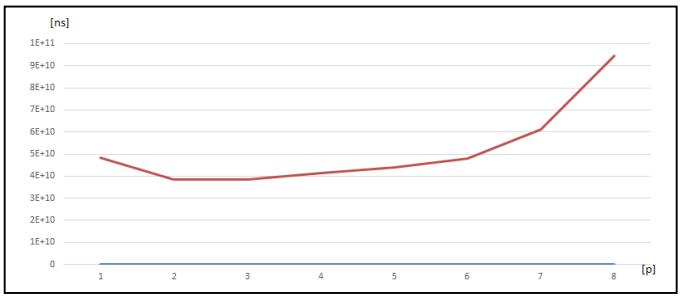

Fig. 5. Measurement "10N, X_K, X_C, X_T"

VI. Conclusion

In this paper the concepts of Deployment pipeline were presented and their improvements regarding parallelization of testing was proposed. Techniques used for parallelization were presented in detail. The architecture of proposed cluster based parallel system for software testing is proposed and evaluated. The evaluation show that the best results were achieved with the compromise between parallelization using processes and parallelization using containers, with the respect to cluster configuration.

References

[1] Humble, Jez, and David Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Pearson Education 2020

[2] Paul M. Duvall, Steve Matyas, Andrew Glover, Continuous Integration: Improving Software Quality and Reducing Risk, Pearson Education, 2007

[3] Sommerville Ian, Software engineering 9th Edition, ISBN -10 137035152, 2011

[4] Mojtaba Shahina, Muhammad Ali Babara, Liming Zhub, Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices, CREST – The Centre for Research on Engineering Software Technologies, The University of Adelaide, Australia, Data61, Commonwealth Scientific and Industrial Research Organisation, Sydney, NSW 2015, Australia

[5] Myers, Glenford J., Corey Sandler, and Tom Badgett, The art of software testing, John Wiley & Sons, 2011

[6] Jovanovic Irena, Software testing methods and techniques, The IPSI BgD Transactions on Internet Research 30, 2006

[7] Jan, Syed Roohullah, et al., An innovative approach to investigate various software testing techniques and strategies, International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN (2006): 2395-1990

[8] Swarm mode overview, https://docs.docker.com/engine/swarm/, Docker Documentation, Date Accessed: May 7 2020, Date Published: May 4 2020

[9] NTP: The Network Time Protocol, http://www.ntp.org/, Website Title: ntp.org: Home of the Network Time Protocol, Date Accessed: May 07, 2020