PERANCANGAN KLASTER SISTEM PENGUJI KODE PROGRAM MENGGUNAKAN DOCKER

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Ferdi Cezano Santosa
NIM: 145150200111025

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2018

PENGESAHAN

PERANCANGAN KLASTER SISTEM PENGUJI KODE PROGRAM MENGGUNAKAN DOCKER

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Ferdi Cezano Santosa
NIM: 145150200111025

Skripsi ini telah diuji dan dinyatakan lulus pada
18 Juli 2018
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Mahendra Data, S.Kom., M.Kom	Ir.Heru Nurwarsito, M.Kom
NIK: 201503 861117 1 001	NIP: 19650402 199002 1 001

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T, M.T, Ph.D
NIP: 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 18 Juli 2018

Ferdi Cezano Santosa

NIM: 145150200111025

KATA PENGANTAR

Puji syukur penulis panjatkan kehadirat Allah SWT, karena berkat rahmat serta bimbingan-Nya, penulis dapat menyelesaikan penulisan skripsi dengan baik. Judul yang penulis ajukan adalah Klaster Sistem Penguji Kode Program Menggunakan Docker.

Dalam penyusunan dan penulisan skripsi ini tidak terlepas dari bantuan, bimbingan serta dukungan dari berbagai pihak. Oleh karena itu dalam kesempatan ini penulis dengan senang hati menyampaikan terima kasih kepada:

1. Bapak Ir. Slamet Santosa dan Ibu Ima Fitrianita S.H selaku orang tua penulis dan saudara penulis yang selalu memberikan dukungan dan doa untuk penulis agar dapat menyelesaikan skripsi ini,
2. Bapak Mahendra Data, S.Kom., M.Kom dan Bapak Ir.Heru Nurwarsito, M.Kom., selaku dosen pembimbing I dan pembimbing II yang telah dengan sabar membimbing, mengarahkan, serta meluangkan waktu untuk penulis sehingga dapat menyelesaikan skripsi ini,
3. Bapak Tri Astoto Kurniawan, S.T, M.T, Ph.D, Bapak Agus Wahyu Widodo, S.T, M.Sc dan Bapak M. Tanzil Furqon, S.Kom, M.CompSc selaku Ketua Jurusan Teknik Informatika, Ketua Program Studi Teknik Informatika dan Sekertaris jurusan Teknik Informatika,
4. Seluruh Dosen Program Studi Informatika Universitas Brawijaya atas kesediaan membagi ilmunya kepada penulis,
5. Seluruh Civitas Akademik Fakultas Ilmu Komputer Universitas Brawijaya yang telah banyak memberi bantuan dan dukungan selama penulis menempuh studi di Informatika Universitas Brawijaya dan selama penyelesaian skripsi ini,
6. Teman–teman angkatan 2014 Informatika, terima kasih atas segala bantuan selama menempuh studi di Teknik Informatika Fakultas Ilmu Komputer Universitas Brawijaya,
7. Seluruh pihak yang telah membantu kelancaran penulisan skripsi yang tidak dapat penulis sebutkan satu persatu.

Penulis menyadari bahwa dalam penyusunan skripsi ini masih banyak kekurangan baik format penulisan maupun isinya. Oleh karena itu, saran dan kritik membangun dari para pembaca senantiasa penulis harapkan guna perbaikan bagi skripsi selanjutnya. Semoga skripsi ini dapat memberikan manfaat bagi semua pihak.

Malang, 18 Juli 2018

Penulis

ferdicezano@gmail.com

ABSTRAK

Sistem evaluasi kode program otomatis merupakan sistem untuk menguji kebenaran program yang sering dipakai, bahkan penelitian tentan sistem ini sudah ada sejak tahun 1965. Seiring berkembangnya teknologi, maka semakin banyak yang menggunakan Docker untuk membuat sistem ini. Dengan menggunakan Docker, sistem akan terisolasi sehingga tidak ada masalah pada keamanan komputer. Karena itulah sistem evaluasi otomatis pada penelitian ini menggunakan Docker untuk mengamankan komputer. Meskipun sudah dapat mengevaluasi program dengan aman, tetapi masih ada masalah pada sistem ini. Masalah tersebut adalah bagaimana agar proses evaluasi menjadi efisien dan tidak terlalu membebani komputer serta bagaimana mempercepat proses evaluasi jika ada program yang membutuhkan waktu yang lama untuk dievaluasi. Karena itu pada penelitian ini dibuatlah klaster sistem penguji kode program menggunakan docker. Pada klaster ini terdapat *master* yang membagi tugas dan slave yang menjalankan evaluasi. *Master* dan *slave* akan dihubungkan dengan RPC karena *master* hanya akan memanggil fungsi pada *slave*. Proses pembagian tugas akan menggunakan algoritme round-robin, dimana *slave* akan dipilih secara bergantian. Berdasarkan hasil pengujian didapatkan bahwa klaster mampu berjalan dan mampu melakukan sampai dengan 104 pengujian sekaligus. Selain itu klaster ini juga dapat membagi tugas dengan efisien sehingga mengurangi beban pada komputer dan mempercepat jalannya evaluasi jika program membutuhkan waktu yang lama untuk dievaluasi.

Kata kunci: docker, evaluasi, pengujian, kode program, RPC, klaster

ABSTRACT

Automated source code evaluation system is a system used to examine a program, in fact its research has been around since 1965. As technology grows, more and more people use Docker to create this system. By using Docker, system will be isolated so it can make computer safe. That's why the automated evaluation system in this study uses Docker to make sure the computer is safe. Even though the system can run safely, there are still some problems occurred on the automated system. The problem is how to make the evaluation process efficient and less burdening the computer and how to speed up the evaluation process if there is a program that takes a long time to be evaluated. Therefore, this research will make a cluster of automated source code evaluation system. In this cluster, there will be two instances, a master that divides the task and slaves that run the evaluation. Master and slaves will be connected with RPC because master will only call a function on the slave. The division of tasks will use round-robin algorithm, where slaves will be selected interchangeably. The test results show that this cluster capable of evaluating programs and able to evaluate up to 104 programs at once. In addition, this cluster can also divide the task efficiently so it can reduce the burdens on the computer and speed up the evaluation if the program takes a long time to be evaluated.

Keyword: docker, evaluation, testing, source code, RPC, cluster

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar belakang

Sistem evaluasi kode program otomatis merupakan sistem yang sering sekali dipakai. Bahkan, penelitian tentang sistem ini sudah ada sejak tahun 1965. Tugas utama dari sistem ini adalah pengecekan kode program. Pengecekan dilakukan dengan cara membandingkan hasil keluaran sistem dengan nilai yang diharapkan dari keluaran program. Pengecekan tersebut biasanya akan mengubah-ubah nilai masukan dari program, dimana pengecekan setiap inputan disebut *test case*. Sistem evaluasi otomatis akan memberi keluaran berapa test case yang berjalan dan hasilnya sesuai dan berapa *test case* yang hasilnya tidak sesuai. Dengan adanya sistem ini setiap orang bisa mengetahui apakah kode program yang sudah dibuatnya benar atau salah. Selain itu sistem ini bisa digunakan untuk berlatih kemampuan untuk menuliskan kode program.

Meskipun sudah banyak aplikasi sistem evaluasi otomatis, tetapi banyak yang belum memikirkan keamanan. Pada penelitian sebelumnya mengenai sistem evaluasi untuk kode siswa (Saraf, et al., 2015), walaupun sudah membuat sistem evaluasi yang dapat berjalan, tetapi sistem tersebut belum memikirkan masalah keamanan. Banyak masalah yang dapat terjadi, apalagi jika sistem dijalankan langsung di komputer yang digunakan, maka akan menimbulkan celah keamanan. Misalkan program dijalankan pada user yang memiliki akses terhadap *file* penting pada komputer. Jika pada salah satu program terdapat kode untuk menghapus *file* system dari sistem operasi di komputer, maka sistem operasi bisa tidak berjalan normal. Selain itu, jika terdapat kode yang akan mengunduh *malware*/program yang bisa merusak seperti virus, maka komputer yang digunakan akan langsung terinfeksi. Untuk menghindari hal-hal yang disebutkan diatas, maka salah satu caranya adalah dengan menjalankan sistem evaluasi otomatis pada *sandbox*. *Sandbox* adalah sebuah lingkungan yang terisolasi sehingga sebuah program bisa dijalankan tanpa mengganggu aplikasi lain (Rouse,2018). Ada berbagai macam cara yang dapat dilakukan untuk menciptakan *sandbox*. Salah satunya adalah dengan menggunakan *Container* pada *Docker*. *Container* merupakan abstraksi dari aplikasi layer yang membungkus perangkat lunak, kode, dan semua dependensi yang dibutuhkan menjadi satu (Sîrbu,2017). *Container* berjalan sebagai proses yang terisolasi sehingga tidak akan mengganggu aplikasi lain. Sudah banya penelitian yang telah menguji keamanan dari *Docker* dan memakai Docker untuk memastikan keamanan dari sistem. Contoh pertama terdapat pada penelitian sebelumya mengenai sistem penguji menggunakan docker (Špaček, Sohlich & Dulík, 2015), docker telah digunakan dan diuji keamanannya. Selain itu pada penelitian *Security Testing as a Service with Docker Containerization* (Pathirathna, et al., 2017), docker digunakan sebagai sandbox dan telah diuji saat terdapat program yang berbahaya docker mampu mengisolasi program tersebut sehingga komputer tetap aman. Selanjutnya tinggal dipikirkan bagaimana untuk mengimplementasikan docker pada sistem penguji kode program.

Sistem evaluasi otomatis memang dapat dengan cepat memberikan evaluasi untuk kode program. Tetapi akan ada masalah yang muncul jika pada saat hampir bersamaan ada banyak program yang harus dievaluasi. Tidak akan efisien dan akan memakan waktu yang lama jika hanya terdapat satu penguji. Selain itu jika hanya ada satu penguji komputer akan terbebani dari segi penggunaan *resource*. Jika komputer memang memiliki spesifikasi yang bagus mungkin tidak akan terlalu bermasalah, tetapi bagaimana jika komputer yang dimiliki tidak memilki spesifikasi yang bagus. Karena itulah dibutuhkan beberapa penguji sekaligus agar tidak memakan waktu. Misalkan kita memiliki beberapa komputer, meskipun bukan dengan spesfikasi yang bagus jika dibuat klaster maka komputer tidak perlu bekerja terlalu keras untuk melakukan semua pengujian dan menampilkan hasilnya. Selain itu jika ada program dengan waktu berjalan yang lama, jika hanya ada satu sistem penguji waktu yang dibutuhkan untuk menyelesaikan pengujian tidak sedikit. Pada penelitian sebelumya (Špaček, Sohlich & Dulík, 2015), meskipun sudah menggunakan *Docker* untuk isolasi sistem tetapi belum dipikirkan bagaimana mengatasi masalah efisiensi dan lamanya waktu pengujian. Salah satu cara untuk menyelesaikan masalah adalah dengan membuat banyak sistem penguji. Dengan banyak sistem penguji, maka pengujian dapat dibagi-bagi antara sistem penguji. Dengan begitu, pengujian dapat berjalan lebih cepat.

Berdasarkan masalah yang ada, maka akan akan digunakan arsitektur *master-slave. Slave* akan melakukan pengujian sementara *master* akan membagikan tugas antara slave. Docker akan diimplementasikan pada slave untuk melakukan pengujian kode program. *Master* dan *slave* akan berkomunikasi menggunakan *Remote Procedure Call*. Pada *master* terdapat pengaturan untuk membagi tugas antara *slave* menggunakan algoritme *round-robin*. *Master* akan menerima *file* program dari klien lalu memerintahkan *slave* untuk menjalankan menguji program tersebut. Setelah itu *slave* akan mengirim hasil pengujian ke *master* lalu *master* menampilkannya ke klien.

1.2 Rumusan masalah

1. Bagaimana pengujian kode program menggunakan docker?
2. Bagaimana komunikasi antara *master* dan *slave*?
3. Bagaimanaka distribusi beban antara *slave*?

1.3 Tujuan

4. Menguji kode program menggunakan docker sehingga terisolasi.
5. Menghubungkan *master* dengan *slave* sehingga dapat berkomunikasi. Dari komunikasi tersebut, *master* dapat menugaskan *slave* untuk melakukan pengujian dan mengambil hasil dari pengujian tersebut.
6. Mendistribusikan beban antara *slave*.

1.4 Manfaat

Manfaat yang diperoleh dari penelitian ini adalah agar kendala dalam mengevaluasi kode program dapat teratasi. Selain itu hasil penelitian ini dapat

digunakan bagi pengajar maupun pembuat lomba/tugas membuat kode program untuk memeriksa hasil pekerjaan. Sedangkan bagi penulis, penelitian ini dapat menjadi wadah untuk menerapkan ilmu yang didapatkan.

1.5 Batasan masalah

Dalam penyusunan skripsi yang baik diberikan beberapa batasan masalah yang akan menjadi batasan dalam melakukan penelitian dan juga dapat memudahkan penyusunan laporan yang sistematis sehingga mudah dipahami. Batasan-batasan yang digunakan dalam skripsi ini antara lain:

1. Pada skripsi ini untuk komunikasi antara *master* dengan *slave* akan menggunakan RPC dengan algortime *round-robin*.
2. Proses evaluasi akan dijalankan di dalam *docker*.
3. Bahasa yang digunakan untuk kode dan program penguji adalah *Python, Java* dan *C++*.
4. Penelitian berfokus kepada komunikasi antara *master* dengan *slave* dan bagaimana menjalankan proses pengujian.

1.6 Sistematika pembahasan

Sistematika penulisan laporan yang ditujukan untuk memberikan informasi dan uraian dari laporan penelitian secara garis besar adalah sebagai berikut :

1. BAB 1 : PENDAHULUAN

   Pada bab ini diuraikan mengenai latar belakang penelitian, rumusan masalah, tujuan, manfaat, batasan masalah, dan sistematika pembahasan.

2. BAB 2 : LANDASAN KEPUSTAKAAN

   Bab ini berisi kajian pustaka mengenai referensi terkait dengan penelitian ini. Selain itu bab ini menjelaskan dan menguraikan tentang pengertian atau teori-teori yang digunakan sebagai penjelasan yang mendasari penelitian ini.

3. BAB 3 : METODOLOGI

   Bab ini menjelaskan mengenai metode dan langkah kerja yang dilakukan dalam proses analisis kebutuhan, perancangan implementasi dan pengujian pada pelaksanaan penelitian yang menjadi objek studi kasus penelitian.

4. BAB 4 : PERANCANGAN

   Bab ini menjelaskan analisis kebutuhan dan tahap perancangan sesuai kebutuhan yang akan diterapkan kedalam sistem.

5. BAB 5 : IMPLEMENTASI

Bab ini membahas tentang implementasi dari sistem sesuai perancangan yang telah dibuat sebelumnya secara rinci beserta langkah-langkah pengerjaannya.

6. BAB 6: PENGUJIAN DAN ANALISIS

Bab ini menjelaskan tentang bagaimana sistem akan diuji. Setelah itu hasil-hasil yang diperoleh akan dijabarkan pada bab ini. Hasil pengujian didapatkan dari pengujian yang telah dilakukan sesuai dengan skenario pengujian yang telah ditentukan. Setelah itu akan dilakukan analisis sesuai dengan hasil pengujian.

7. BAB 7 : PENUTUP

Bab ini berisi kesimpulan dan saran yang diperoleh selama proses penelitian.

BAB 2 LANDASAN KEPUSTAKAAN

Bab ini membahas tinjauan pustaka yang meliputi kajian pustaka dan dasar teori yang nantinya digunakan untuk mendukung serta menjadi dasar untuk penelitian ini. Kajian pustaka membahas penelitian sebelumnya yang telah dilakukan dan berhubungan dengan penelitian ini. Dasar teori membahas tentang teori yang berhubungan dan diperlukan untuk menyusun penelitian.

2.1 Kajian Pustaka

Terdapat beberapa penelitian yang telah dilakukan sebelumya yang berhubungan dengan penelitian yang akan dilakukan ini. Tabel dibawah menunjukkan kajian pustaka yang dilakukan.

Tabel 2.1 Kajian Pustaka

No	Judul	Tahun	Penulis	Hasil Penelitian	Penelitian Penulis
1	Automatic Evaluation System for Student Code	2015	Pratik Saraf, Shankar Ramesh, Sachin Patel dan Prof. Sujata Pathak	Sistem evaluasi kode bahasa Java berbasis web	Menambahkan penggunaan *docker* untuk menjalankan kode program dan menambah bahasa yang bisa diuji.
2	Docker as Platform for Assignments Evaluation	2015	František Špaček, Radomír Sohlich dan Tomáš Dulík	Sistem evaluasi kode dengan menggunakan Docker	Menambahkan beberapa sistem penguji untuk menambah efisiensi dan mempercepat pengujian dan menggunakan

					load balancing untuk membagi tugas antar sistem serta menggunakan RPC.
	Security Testing as a Service with Docker Containerization	2017	P.P.W. Pathirathna, V.A.I. Ayesha, W.A.T. Imihira, W.M.J.C. Wasala, Nuwan Kodagoda, E. A. T. D. Edirisinghe	Sistem untuk Menguji Keamanan Aplikasi Berbasis Web menggunakan Docker Containerization	Menambahkan beberapa sistem penguji untuk menambah efisiensi dan mempercepat pengujian dan menggunakan *load balancing* untuk membagi tugas antar sistem serta menggunakan RPC.

2.2 Dasar Teori

Untuk melaksanakan penelitian ini dibutuhkan referensi dan dasar teori untuk melakukan penelitian. Dengan adanya referensi dan penelitian maka akan memperlancar proses perancangan dan penelitian. Berikut adalah beberapa teori yang akan digunakan untuk menjadi dasar penelitian:

2.2.1 *Unit Testing*

Unit Testing merupakan praktek pengujian fungsi atau unit tertentu pada sebuah kode tertentu(McFarlin,2012). Dengan melakukan unit testing maka penulis kode program bisa menguji apakah fungsi yang terdapat pada kode program memberikan keluaran yang sesuai. Melakukan unit testing sangat membantu dalam meningkatkan kualitas sebuah kode. Selain itu jika pembuat program sering melakukan unit testing, maka akan mempermudahnya untuk menuliskan kode program lain lebih baik lagi.

Untuk melakukan *unit testing* dapat dilakukan dengan cara membuat *test case*. *Test case* dapat dibuat dengan mempersiapkan input yang akan diujicobakan ke kode program dan *file* jawaban. *File* jawaban akan dicocokkan dengan keluaran dari kode program untuk menentukan benar atau tidaknya keluaran kode program. Dari beberapa test case yang dijalankan hasilnya akan dirangkum sehingga mengetahui apakah kode program sudah benar-benar berjalan sesuai dengan yang diinginkan.

2.2.2 Python

Python merupakan bahasa pemrograman tingkat tinggi yang berorientasi objek. *Python* memiliki sintaks yang mudah dibaca, sehingga cocok untuk orang yang baru belajar memahami program. *Python* dapat digunakan untuk berbagai macam jenis pemrograman seperti pengembangan web, pemrograman jaringan, dan komputasi matematika. *Programmer python* merupakan salah satu dari 5 *programmer* dengan bahasa lain yang paling dicari dan memiliki gaji tinggi(Heath,2017). Itulah mengapa pada penelitian ini akan melakukan evaluasi dari kode bahasa Python.

Untuk menjalankan *master*, akan dibuat *server* berbasis bahasa python. Python digunakan karena mendukung pemrograman jaringan. Pada python, terdapat dua level akses ke servis jaringan. Pada *low level*, kita bisa mengakses *socket* pada sistem operasi sehingga bisa membuat koneksi. Python juga menyediakan *library* yang bisa mengakses level yang lebih tinggi pada application layer seperti HTTP,FTP, dan lain-lain. Pada penelitian ini akan digunakan salah satu *library* yang ada yaitu RPyC untuk melakukan RPC yang nanti akan dijelaskan lebih lanjut pada subbab RPC.

2.2.3 Java

Bahasa pemrograman Java merupakan bahasa pemrograman yang berbasis pemrograman berorientasi objek. Bahasa Java telah banyak digunakan untuk mengembangkan aplikasi pada berbagai *environment*. Karena itulah *programmer* Java juga merupakan *programmer* yang paling banyak dicari dengan gaji tinggi. Java merupakan turunan dari bahasa C, sehingga sintaksis dari Java mirip dengan bahasa C. Karena alasan yang telah disebutkan, maka pemrograman dengan bahasa java juga digunakan dalam penelitan ini.

2.2.4 C++

Bahasa pemrograman C++ juga bisa diuji pada sistem yang akan dibuat. Bahasa C++ juga seringkali dipakai untuk mengembangkan program. Bahasa ini sejak tahun 1998 telah distandarisasi oleh ISO. Bahasa ini juga disusun langsung ke bahasa mesin, sehingga jika dioptimalkan bisa menjadi bahasa tercepat. Bahasa C++ memiliki banyak *library* yang dapat digunakan. Selain itu bahasa ini bisa untuk berbagai paradigma, salah satunya pemrograman berorientasi objek.

2.2.5 Remote Procedure Call

RPC merupakan protokol dimana sebuah program dapat meminta servis dari program lain di sebuah jaringan. RPC termasuk jenis model arsitektural jaringan yang melihat dari sisi berkomunikasi/*Communication Paradigm* yaitu Remote Invocation. RPC menggunakan model klien-*server*. Program yang melakukan request adalah klien sedangkan yang memberikan servis adalah *server*. RPC merupakan operasi yang tersinkronisasi sehingga jika terdapat operasi berjalan, program lain yang meminta servis akan ditunda terlebih dahulu. Tetapi dengan menggunakan thread, maka operasi dapat dilakukan secara parallel.

2.2.5.1 RPyC

RPyC merupakan *library* python untuk RPC, klastering dan komputasi terdistribusi. RPyC menggunakan teknik object-proxying, sehingga remote object dapat dimanipulasi seolah-olah objek tersebut lokal. Untuk menginstall RPyC dapat menggunakan perintah "pip install rpyc". Untuk memakainya, harus terlebih dahulu memasukkan *library* pada kode program dengan perintah "import rpyc". *Library* ini akan digunakan untuk komunikasi antara *master* dan *slave*.

2.2.6 Docker

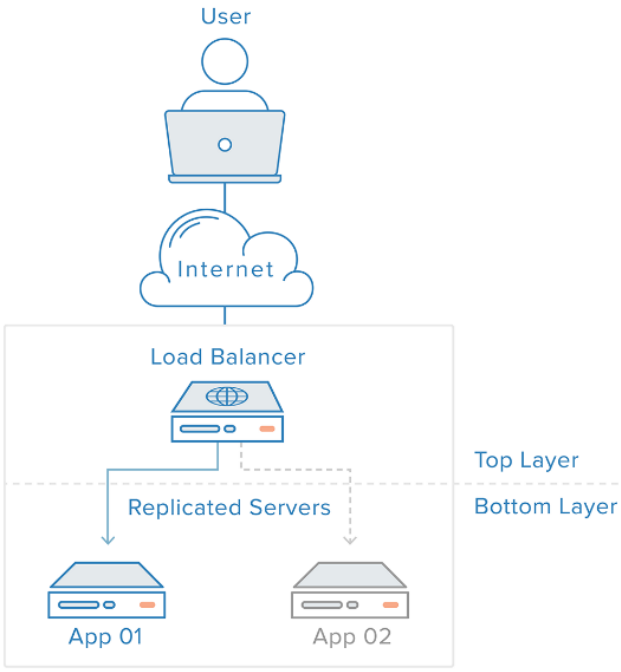
Docker merupakan aplikasi *open source* yang berfungsi untuk membungkus perangkat lunak dan semua dependensinya(Sîrbu, 2017). Pengaturan dan hal pendukung lainnya akan dibangun menjadi sebuah *image*. Hasil instansiasi dari image tersebut dinamakan *container*. *Container* inilah yang nantinya bisa digunakan untuk menjalankan perangkat lunak.

Docker memiliki kelebihan untuk dapat mengisolasi container dari *host* maupun *container* lain(Sîrbu, 2017). Ini karena docker menggunakan *kernel namespace*. Pada saat *container* dijalankan, docker akan membuat satu set *namespace* dan *control group* sehingga proses dari container tidak akan mengganggu *host* dan container lain. *Container* juga memiliki pengaturan jaringan tersendiri sehingga sebuah container tidak akan memiliki hak akses dari *socket* atau *interface* container lain.

2.2.7 *Load balancing*

Load balancing digunakan untuk mengarahkan pengguna ke salah satu *server* yang telah direplika sebelumnya. Dengan adanya *load balancing*, maka

beban *server* dalam menangani permintaan user akan terbagi. Pada gambar berikut merupakan ilustrasi dari *load balancing*.



Gambar 2.1 Ilustrasi *Load balancing*(Anderson,2017)

Pada *load balancing* terdapat beberapa algoritme yang dapat digunakan untuk menentukan *server* mana yang dipilih. Salah satunya adalah *round robin*. *Round robin* akan memilih *server* secara bergantian. Salah satu cara untuk mengimplementasikan algoritme ini adalah dengan menggunakan rumus modulo. *Load balancing* akan diimplementasikan pada *server* yang terdapat pada *master*.

2.2.8 Apache JMeter

Aplikasi ini termasuk aplikasi open-source. Aplikasi ini dibuat menggunakan Java dan digunakan untuk melihat sifat dan mengukut performa dari suatu program. Aplikasi ini lebih sering digunakan untuk melakukan tes pada aplikasi web, meskipun sekarang bisa digunakan untuk jenis aplikasi lainnya. Cara kerja dari aplikasi ini adalah dengan mensimulasikan *load* yang berat pada *server* untuk mengetahui kemampuan *server* atau menganalisis performa *server* pada tipe *load* yang berbeda-beda. Pada penelitian ini akan digunakan salah satu plugin yang terdapat pada JMeter yang digunakan untuk mengetes *websocket server* yaitu *websocket request-response sampler.*

2.2.9 SYSSTAT

SYSSTAT adalah paket dari banyak *tool* yang terdapat pada unix. Alat-alat pada SYSSTAT digunakan untuk memantau penggunaan sistem dan memantau performa dari suatu sistem. Terdapat dua alat pada SYSSTAT yang nantinya akan digunakan pada komputer ini yaitu mpstat dan sar.

Mpstat merupakan alat khusus untuk memantau CPU. Untuk menggunakan alat ini cukup menuliskan perintah "mpstat (*interval*) (*count*)". Perintah interval digunakan untuk mengatur setiap berapa detik alat ini akan menangkap penggunaan CPU dan menampilkannya. Perintah *count* digunakan untuk mengatur berapa kali alat ini akan memperlihatkan penggunaan cpu dari komputer. Pada salah satu kolom di keluaran mpstat akan menunjukkan berapa persen penggunaan CPU saat itu. Pada baris paling akhir semua nilai yang sudah dicatat akan dihitung rata-ratanya. Rata-rata dari persentase penggunaan CPU inilah yang nanti akan dicatat dan digunakan untuk mendapatkan hasil pengujian.

Sar adalah alat yang dapat digunakan untuk memantau penggunaan memori dari komputer. Untuk menggunakan sar untuk memantau penggunaan memori dapat menggunakan perintah "sar –r (*interval*) (*count*)". Perintah interval digunakan untuk mengatur setiap berapa detik alat ini akan menangkap penggunaan memori dan menampilkannya. Perintah *count* digunakan untuk mengatur berapa kali alat ini akan memperlihatkan penggunaan memori dari komputer. Salah satu kolom pada hasil keluaran sar akan menampilkan persentase penggunaan memori. Lalu semua hasil akan dihitung rata-ratanya. Hasil penghitungan rata-rata tersebut dapat digunakan untuk menentukan berapa persen memori yang digunakan pada saat menjalankan suatu program.

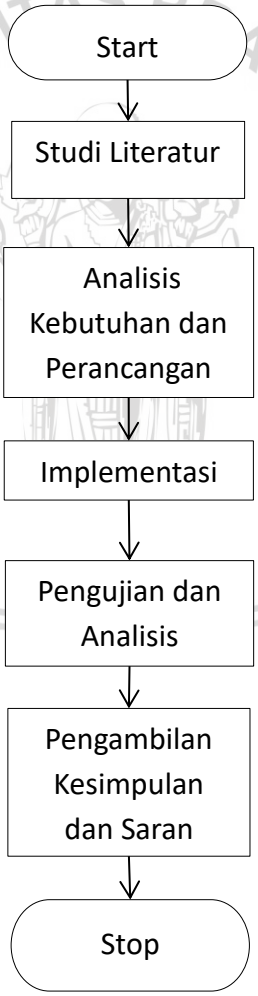
BAB 3 METODOLOGI

3.1 Jenis Penelitian

Jenis penelitian yang dilakukan adalah implementatif yang berfokus pada pengembangan. Penelitian ini nantinya akan menghasilkan jaringan komputer melalui proses perancangan hingga analisis kebutuhan.

3.2 Metodologi Penelitian

Bagian ini membahas tentang metode yang digunakan untuk membangun lingkungan pengujian dengan menggunakan *load balancer* dan *docker*. Selain itu pada bagian ini akan menjelaskan tentang langkah-langkah yang dilakukan untuk melaksanakan penelitian. Berikut adalah gambar dari diagram alir dari penelitian ini



Gambar 3.1 Metodologi Penelitian

3.2.1 Studi Literatur

Studi literatur digunakan sebagai acuan untuk melakukan penelitian yang bisa berasal dari berbagai sumber. Dengan menggunakan studi literatur maka akan diperoleh dasar teori untuk menyelesaikan permasalahan dengan baik dan benar. Berikut adalah teori atau referensi yang digunakan untuk penelitian ini:

- Unit Testing
- *Python*
- *Java*
- *C++*
- *RPC*
- *Docker*
- *Load balancing*
- Apache JMeter
- SYSSTAT

Literatur diperoleh dari berbagai sumber seperti ebook, jurnal, website dan lain-lain.

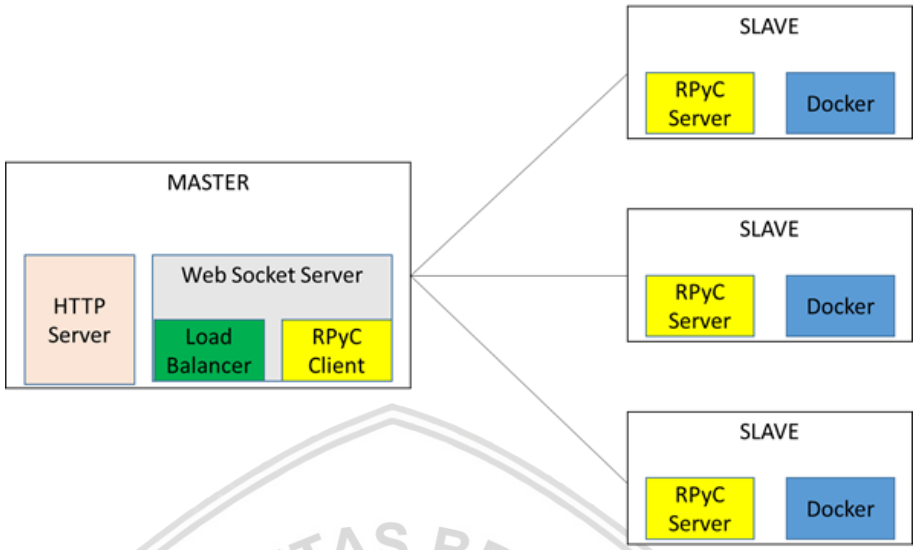
3.2.2 Analisis Kebutuhan dan Perancangan

Pada tahap ini akan dilakukan perancangan terhadap sistem yang akan dibuat. Sebelumnya akan dilakukan analisis kebutuhan untuk membuat klaster ini. Selanjutnya dilakukan perancangan fungsi dengan kebutuhan. Nantinya akan terdapat dua jenis sistem yang akan dirancang fungsinya. Jenis sistem pertama adalah *master*. *Master* didalamnya akan terdapat http *server* dan *websocket server*. Pada *websocket server* akan terdapat *load balancer* dan juga RPyC *client*. Jenis sistem kedua adalah *slave*. *Slave* akan digunakan untuk menguji kode program. Perancangan akan dimulai dari container yang akan digunakan untuk menjalankan program. Nantinya akan terdapat satu container untuk masing-masing bahasa pemrograman. Setelah itu akan dilakukan perancangan *master* dan *slave*. Terakhir akan dirancang bagaimana perangkat lunak berjalan pada *master* dan *slave*. Setelah itu akan dirancang juga bagaimana proses pengujian dilakukan. Rancangan umum ditunjukkan pada gambar 3.2.

3.2.3 Implementasi

Implementasi akan hasil rancangan yang telah dirancang pada bab perancangan. Pada tahap implementasi akan digunakan *virtual machine* sebagai simulasi komputer. *Virtual machine* yang berfungsi sebagai *slave* akan dipasang *docker* untuk menjalankan program evaluasi. Tiap *slave* dipasang 3 container berbeda untuk setiap bahasa. Salah satu virtual machine lain akan digunakan sebagai *master*. Pada *master* ini akan dijalankan *server* yang akan mendistribusikan beban dan berkomunikasi dengan *slave*. Terdapat pemrograman jaringan dimana di

dalamnya terdapat algoritme round-robin. Dengan begitu *master* akan mendistribusikan tugas menjalankan dan menguji program ke salah satu *slave*.



Gambar 3.2 Rancangan Klaster

Untuk menghubungkan *master* dan *slave*, pada semua virtual machine akan dipasang RPyC untuk komunikasi.Dengan RPyC, *master* mendapatkan hasil pengujian dari *slave* dan dikembalikan ke *client*. Nantinya *client* akan mengakses *master* dan meng*upload* program.

3.2.4 Pengujian dan Analisis

Pengujian sistem dilakukan untuk mengetahui apakah implementasi pada klaster sistem evaluasi kode berbasis *docker* berhasil atau tidak. Pengujian sistem dilakukan dengan cara menguji tiap fungsi. Cara menguji ialah dengan mengakses *master*. Setelah itu pengujian dilakukan dengan cara mengunggah *file* yang akan diuji ke *master* dan melihat keluaran dari sistem. Pengujian akan dilakukan dengan mencoba melakukan pengujian kepada beberapa program yang benar dan salah untuk melihat apakah sistem sudah bisa menguji dengan benar. Selain itu juga akan dilakukan pengujian performa. Pengujian performa akan menguji apakah sistem sudah berhasil melakukan *load balancing* dan menguji bagaimana penggunaan *resource* serta kecepatan pengujian yang dijalankan. Pengujian ini berfungsi sebagai tolak ukur keberhasilan dari penelitian. Nantinya dari hasil pengujian ini akan dianalisa.

3.2.5 Pengambilan Kesimpulan dan Saran

Pengambilan kesimpulan dilakukan setelah melakukan pengujian dan melihat dari hasil pengujian. Setelah itu akan dituliskan saran untuk pengembangan sistem dan penelitian berikutnya.

BAB 4 ANALISIS KEBUTUHAN DAN PERANCANGAN

4.1 Analisis Kebutuhan

4.1.1 Pendahuluan

4.1.1.1 Tujuan

Analisis kebutuhan ini menjelaskan tujuan dan fitur dari sistem, antarmuka sistem, apa yang akan dilakukan sistem, serta ruang lingkup kerja sistem.

4.1.1.2 Ruang Lingkup

Ruang lingkup dari sistem ini yaitu melakukan penerapan konsep komunikasi sistem terdistribusi. Pengguna akan berkomunikasi dengan *master* menggunakan koneksi TCP. Setelah itu *master* akan berkomunikasi dengan *slave* menggunakan RPC untuk menguji sistem yang hasilnya akan ditampilkan pengguna. Pembuatan sistem ini ditujukan kepada penguji, developer, user sehingga kedepannya sistem ini dapat dikembangkan lebih lanjut dan dapat digunakan guna keperluan pembelajaran atau penelitian.

4.1.1.3 Istilah

Tabel 4.1 Tabel Istilah

Term	Definition
Protokol RPC	Protokol komunikasi berbasis arsitektur komunikasi *client server*. Biasanya digunakan oleh program untuk memanggil fungsi pada komputer lain pada sebuah jaringan.
RPyC Client	Berfungsi untuk memanggil fungsi yang ada pada RPyC *Server*.
RPyC Server	*RPYC server* sebagai penyedia fungsi yang akan dipanggil / dijalankan oleh RPyC *Client*
Master	Merupakan komputer yang akan berfungsi untuk menerima data dari pengguna, menentukan *slave* tujuan dan memanggil fungsi pada slave.
Slave	Merupakan komputer yang akan berfungsi untuk menjalankan pengujian kode program.
Developer	Seseorang yang meneliti artikel atau penelitian dan memiliki kemampuan untuk merekomendasikan persetujuan dari artikel untuk publikasi atau meminta perubahan harus dibuat dalam artikel
Software Requirements Specification (SRS)	Sebuah dokumen yang benar-benar menggambarkan sebuah fungsi dari sistem yang

	diusulkan dan bagaimana kebutuhan agar sistem dapat beroperasi
*User/*pengguna	*Reviewer* atau penulis.
Docker	Perangkat lunak yang digunakan untuk menguji kode program

4.1.1.4 Sistematika

Untuk mempermudah dalam meninjau pembahasan yang ada, maka diperlukan sistematika yang digunakan sebagai pedoman penulisan *Software Requirements Specification* (SRS) ini. Bagian ini dibagi menjadi 3 bagian sebagai berikut:

1. Pendahuluan

   Pada bagian ini terdiri dari tujuan, ruang lingkup, istilah dan sistematika penulisan.

2. Deskripsi Umum

   Bagian ini terdiri dari perspektif produk, fungsi produk, karakteristik pengguna, batasan-batasan, asumsi dan ketergantungan,

3. Spesifikasi Kebutuhan

   Bagian ini terdiri dari kebutuhan fungsional, kebutuhan antarmuka eksternal, kebutuhan non-fungsional.

4.1.2 Deskripsi Umum

4.1.2.1 Perspektif Produk / Sistem

Sistem ini akan dibagi menjadi dua yaitu *master* dan *slave*. *Master* akan menerima data dari pengguna lalu melakukan perhitungan untuk menentukan *slave* dan memanggil fungsi yang terdapat pada *slave* tersebut. Untuk berkomunikasi dengan pengguna sistem akan menggunakan *interface web application* sedangkan untuk berkomunikasi dengan *slave* akan menggunakan RPC. Pada *slave*, terdapat fungsi yang digunakan untuk menjalankan pengujian program. Sistem ini dapat menangani proses pengiriman data sebagai penerapan konsep komunikasi sistem terdistribusi yang dimulai dari pengguna hingga ditampilkan lagi pada pengguna menggunakan *interface* web*server*.

4.1.2.2 Kegunaan

Dengan adanya sistem ini diharapkan pengujian program dengan cepat karena terdapat tiga penguji dimana masing-masing penguji bisa menguji tiga bahasa pemrograman. Selain itu karena pengujian dilakukan menggunakan docker, maka proses pengujian akan aman dan tidak mengganggu lingkungan sistem.

4.1.2.3 Karakteristik Pengguna

Pengguna hanya bertindak sebagai pemilik kode program yang akan diujikan pada sistem ini. Setelah mengunggah *file*, pengguna akan menunggu proses pengujian yang hasilnya akan ditampilkan pada web *server*.

4.1.2.4 Lingkungan Operasi

Persyaratan kebutuhan lingkungan yang mendukung kebutuhan sistem adalah *Master* harus dapat berkomunikasi dengan tiap *Slave.*

4.1.2.5 Batasan Perancangan dan Implementasi

Beberapa batasan akan dijelaskan sebagai berikut:

1. Sistem akan dijalankan pada virtual komputer.
2. Pengujian program akan berjalan jika pengguna telah berhasil mengunggah *file*.
3. Program yang diujikan ada dua macam, yang satu merupakan program yang sesuai/benar sedangakan satunya adalah program yang dibuat salah dengan sengaja.

4.1.2.6 Asumsi dan Ketergantungan

Ketergantungan sebagai persyaratan sistem adalah program yang akan diuji harus sesuai nama, atribut, klas, dan fungsinya dengan program penguji, jika tidak pengujian tidak akan berjalan.

4.1.3 Kebutuhan Antarmuka Eksternal

4.1.3.1 Antarmuka Pengguna

Pengguna dapat mengunggah *file* yang akan diuji dan melihat hasil pengujian. Untuk melakukan proses ini pengguna harus memiliki koneksi dengan *Master* dan menghubungi *master* lewat browser.

4.1.3.2 Antarmuka Perangkat Keras

1. *Master*
    a. Virtual komputer
2. *Slave*
    a. Virtual komputer

4.1.3.3 Antarmuka Perangkat Lunak

1. *Master*
    a. Python
    b. RPyC
    c. Web *Server* (Apache)

2. *Slave*
    a. Python
    b. RPyC
    c. Docker

4.1.3.4 Antarmuka Komunikasi

Komunikasi yang terjadi pada sistem berada pada satu jaringan lokal yang sama sehingga dapat menghubungkan *master* dan *slave* menggunakan RPC. Pada *master* juga terdapat web *server* yang akan digunakan untuk berkomunikasi dengan pengguna.

4.1.4 Kebutuhan Fungsional

Kebutuhan fungsional merupakan kebutuhan yang harus dipenuhi agar suatu sistem dapat bekerja sesuai keinginan, beberapa kebutuhan fungsional yang harus ada pada sistem ini dijelaskan pada Tabel 4.2 sebagai berikut:

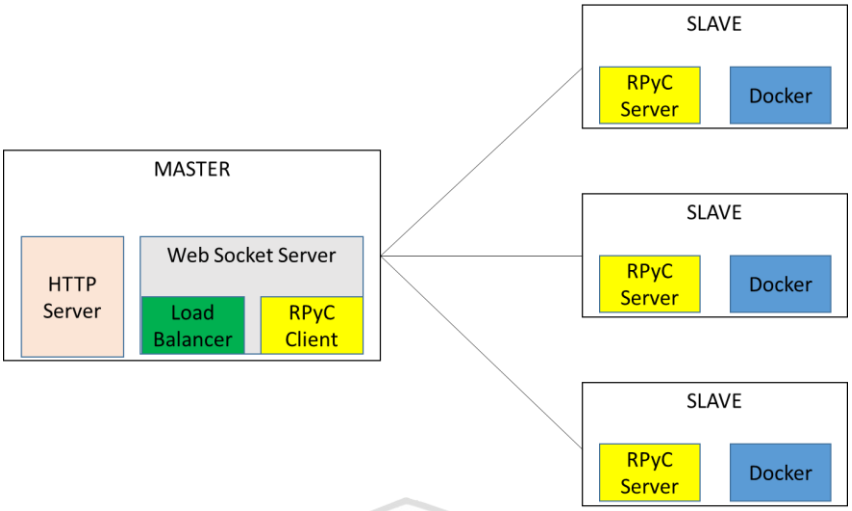
Tabel 4.2 Kebutuhan Fungsional

No	Kebutuhan Fungsional
1	*Master* dapat menerima data/*file* dari pengguna
2	*Master* dapat melakukan *load balancing* untuk memilih *slave*.
3	*Master* dapat memanggil fungsi yang terdapat pada *slave*.
4	*Master* dapat menerima hasil pengujian dari *slave* dan menampilkannya ke pengguna
5	*Slave* dapat menerima koneksi dari *master*.
6	*Slave* dapat menjalankan pengujian saat *master* memanggil fungsi yang ada.
7	*Slave* dapat mengirimkan hasil pengujian kepada *master*

4.2 Perancangan

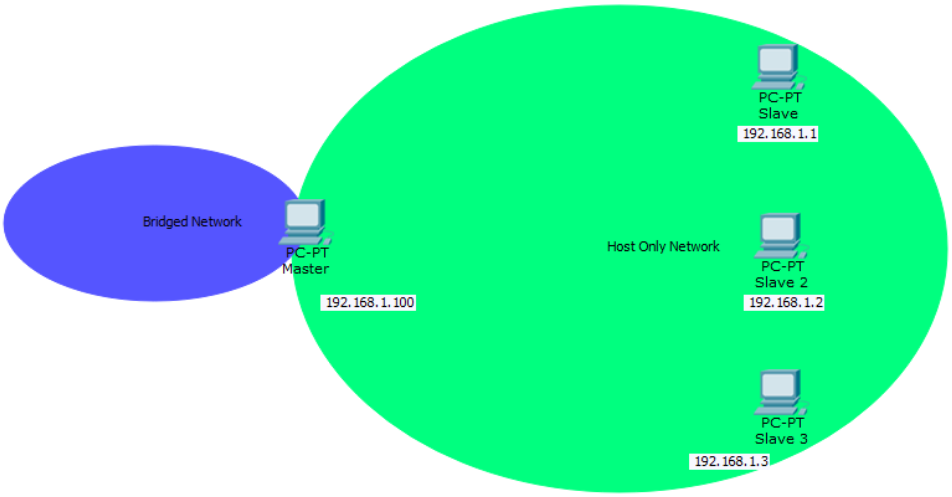
4.2.1 Gambaran umum

Terdapat ada beberapa komponen yang terdapat pada klaster, yaitu *master*, *server*, *load balancer*, RPyC *Client*, *slave*, RPYC *server* dan docker. *Master* akan berfungsi untuk menerima *file* dari klien lalu menentukan pada *slave* mana akan dilakukan pengujian. Selanjutnya *master* akan berkomunikasi dengan *slave* lewat RPC. Melalui RPC ini nanti *file* akan dijalankan di docker setelah itu hasilnya akan dicocokkan dengan *file* berisi jawaban. Setelah itu hasilnya akan dikembalikan pada klien melalui *server* yang ada pada *master*. Gambaran umum klaster sistem penguji kode dapat dilihat pada gambar 4.1.



Gambar 4.1 Klaster Sistem Penguji Kode Program

Komunikasi antara *master* dan *slave* akan terdapat pada jaringan *host only network* yang terdapat pada *virtual box*. Jaringan ini digunakan karena *slave* tidak akan terlihat oleh komputer selain *host* dan *master* serta komputer *host* bisa dengan mudah melakukan pengaturan pada semua komputer. Lalu *master* juga memiliki interface yang akan terkoneksi ke *bridged network* sehingga dapat diakses oleh seluruh komputer yang terdapat pada jaringan dimana *host* virtual box terkoneksi. Pada gambar 4.2 akan ditunjukkan bagaimana topologi dari klaster ini.



Gambar 4.2 Topologi Klaster Sistem Penguji Kode Program

4.2.2 Perancangan Alur Kerja

Pertama klien akan meng*upload file* program dengan mengakses web *server*. Setelah itu web *server* akan mengirim ke *server* yang dibuat dengan menggunakan Bahasa python pada *master*. Selanjutnya *master* akan menentukan *slave* mana yang akan digunakan. Proses pengujian akan dimulai pada saat *master* menjalankan fungsi yang terdapat pada RPyC *slave*. *Master* akan menjalankan

program secara remote pada docker di *slave* dengan memanggil fungsi yang ada pada RPyC *server*. Setelah itu pada *slave* hasil keluaran program akan dicocokkan dengan kunci jawaban yang telah disimpan pada setiap *slave*. Setelah selesai, fungsi yang ada pada *slave* tadi akan mengembalikan nilai dan hasil pengujian yang telah dilakukan. Pada gambar 4.3 digambarkan proses pengujian program dengan sequence diagram.

4.2.3 Perancangan Container

Docker Container nantinya akan digunakan sebagai wadah untuk menguji kode program. Karena itu diperlukan rancangan yang baik agar nantinya proses pengujian bisa berjalan lancar. Kebutuhan untuk menjalankan docker ini adalah menggunakan dasar *image* python, openjdk dan gcc. Karena pada penelitian ini akan berfokus untuk menguji kode bahasa python, Java dan C++ maka diperlukan container yang sudah memiliki *compiler* dari ketiga bahasa tersebut. Nantinya akan digunakan image dari *repository* docker yaitu *python*, *openjdk* dan *gcc*.

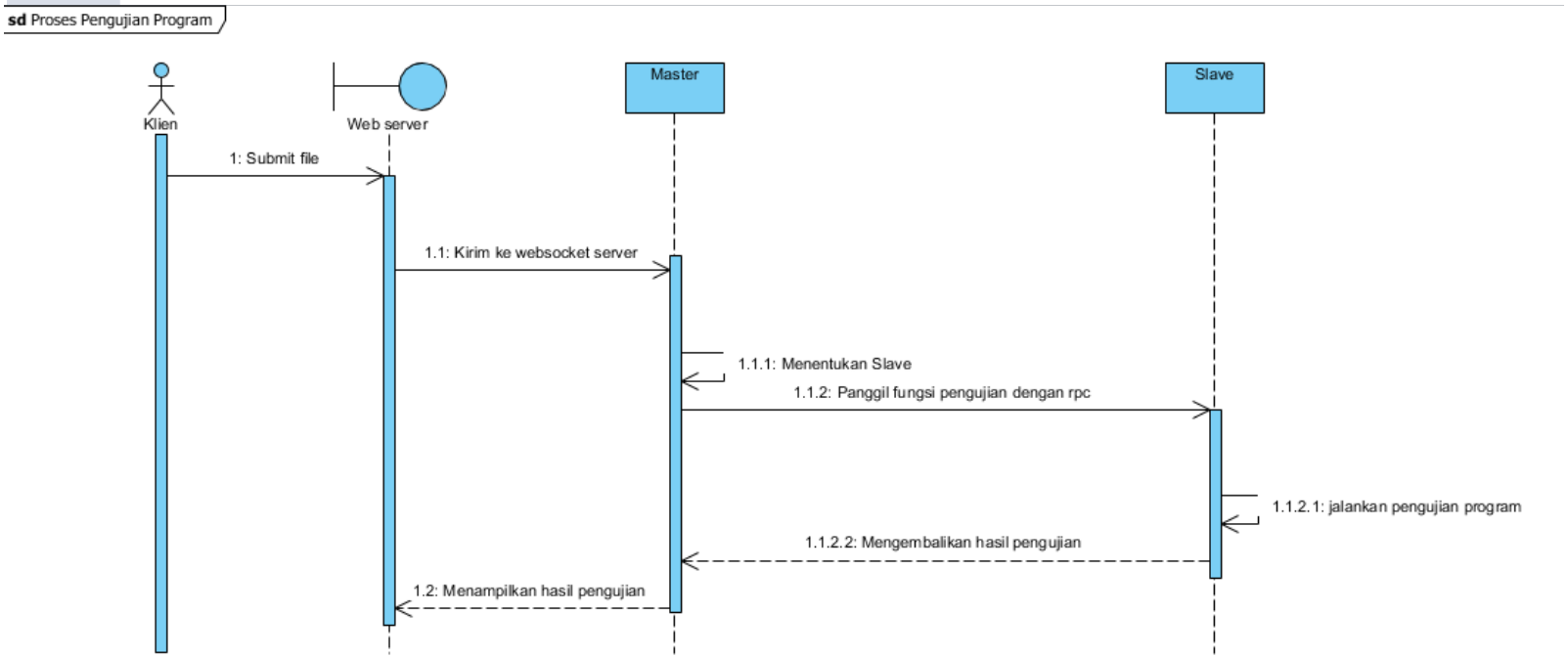
4.2.4 Perancangan Perangkat Lunak

4.2.4.1 Perancangan HTTP Server

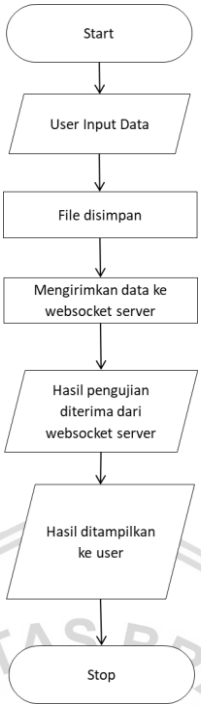
HTTP *Server* akan berjalan menggunakan apache. Lalu kode program akan ditulis dengan menggunakan php dan html. *Server* akan berjalan pada *master*. Pada halaman pertama akan terdapat form untuk *user* memasukkan nama dan *file*. Jika *user* klik *upload*, maka program akan disimpan pada komputer. Setelah itu pada halaman selanjutnya web akan mengirim nama melalui *websocket* untuk memulai pengujian. Saat data hasil pengujian dari *websocket* diterima, maka data tersebut akan langsung ditampilkan ke user. Pada gambar 4.3 akan ditunjukkan bagaimana HTTP *server* berjalan dengan menggunakan flowchart.

4.2.4.2 Perancangan Websocket Server

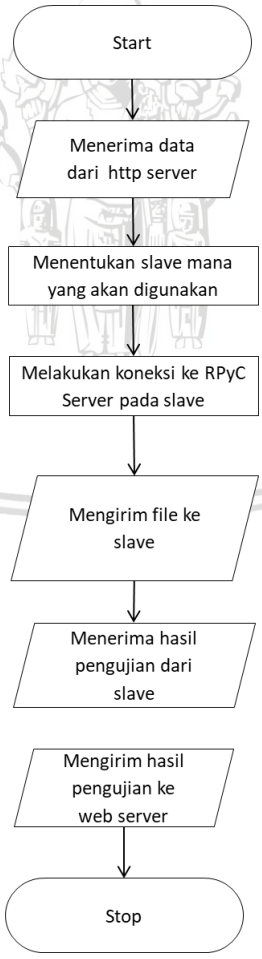
Websocket server akan dibuat menggunakan python. Modul yang digunakan adalah twisted, txws, json dan rpyc. Setelah itu program akan dijalankan di *master*. Program akan mulai berjalan saat HTTP *server* membuka koneksi dan mengirim data ke *server* ini. *Websocket server* akan menentukan *slave* mana yang akan digunakan dengan mengakses variabel global. Variabel tersebut akan dihitung menggunakan modulo tiga. Setelah ditentukan, maka *server* ini akan membuka koneksi dengan *slave* dengan menggunakan RPyC. Selanjutnya data akan dikirim melalui rpc tadi dan mengunggu hasil pengujian dari *slave*. Saat hasil pengujian diterima, hasil tadi akan diteruskan ke HTTP *server*. Pada gambar 4.4 akan ditunjukkan bagaimana *Websocket server* berjalan dengan menggunakan flowchart



Gambar 4.3 Proses Pengujian Program



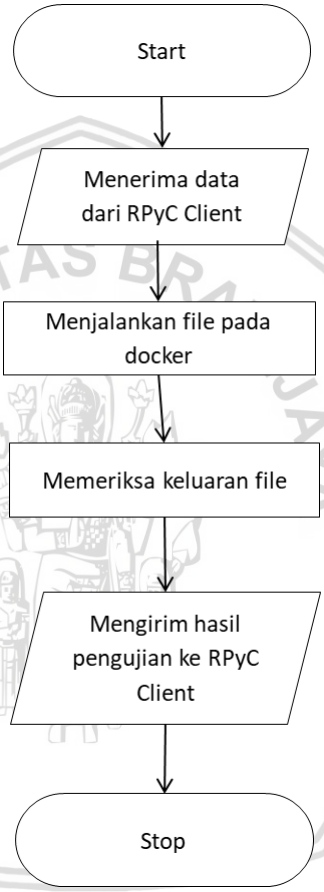
Gambar 4.4 HTTP *Server*



Gambar 4.5 *Websocket Server*

4.2.4.3 Perancangan Slave

Slave juga akan dibuat menggunakan python. Modul yang digunakan antara lain rpyc. RPyC akan dijalankan pada port tertentu dari *slave*. Saat menerima data dari *master*, *slave* akan menjalankan program pada docker. Untuk menjalankan program maka akan digunakan modul *os*. Selanjutnya keluaran dari program akan dicek dengan kunci jawaban yang telah disimpan pada *slave* sebelumnya. Dari proses tadi maka akan menghasilkan *file* berisi hasil pengujian. *File* ini yang akan dikirim kembali ke *master*. Pada gambar 4.5 akan ditunjukkan bagaimana *Slave* berjalan dengan menggunakan flowchart



Gambar 4.6 *Slave*

4.2.5 Perancangan Pengujian

Perancangan pengujian dibutuhkan untuk menentukan batasan-batasan serta skenario pengujian yang harus dipenuhi oleh sistem yang dikembangkan.

4.2.5.1 Pengujian Fungsional

Pengujian fungsional merupakan pengujian secara *black-box* yang dilakukan untuk menguji fungsional sistem apakah sudah berjalan dengan baik dan benar. Pengujian dilakukan dengan menjalankan interaksi antara *master* dan *slave*. Pengujian ini dianggap berhasil jika semua fungsi yang dipaparkan tidak

menunjukkan adanya kegagalan dan kesalahan dalam pengujian. Pengujian akan dilakukan dengan mencoba menguji program sederhana untuk menghitung permutasi. Program akan mengacu pada rumus nPr. Nilai n yang diujikan dimulai dari -1 sampai lima. Untuk setiap nilai n, akan dicoba nilai r mulai dari -1 sampai n+1.

4.2.5.2 Pengujian Performa

Pengujian performa dilakukan untuk menguji performa dari sistem. Pengujian berfokus pada pembagian kerja yang dilakukan oleh *master* sudah benar atau belum. Pengujian akan dilakukan dengan menambahkan sebuah kalimat pada hasil yang menunjukkan dimana pengujian dilkukan. Dengan begitu dapat dihitung proses pengujian yang dilakukan setiap *slave*. Pengujian ini akan dillakukan dengan jumlah klien yang divariasi. Pada pengujian pembagian kerja akan diuji jika terdapat 6, 12, 24, 48, 60, 96 dan 120 pengguna.. Pengguna merupakan kelipatan 6 karena *file* yang bisa diuji ada 6. Pengujian akan dilakukan dengan menggunakan aplikasi jMeter.

Selain itu juga terdapat pengujian pengunaan CPU dan memori untuk melihat apakah dengan klaster membagi rata pengujian dapat menurunkan penggunaan *resource*. Untuk membandingkan hasil pengujian dibuat satu sistem baru dimana sistem tersebut akan langsung melakukan pengujian tanpa membagi tugas kepada sistem lain. Jumlah pengguna yang akan diuji adalah 6, 12, 24, 48, 60 dan 96 pengguna. Untuk melihat penggunaan CPU dan memori digunakan tool mpstat dan sar. Yang dilihat adalah rata-rata penggunaan *resource* selama pengujian dilakukan.

Pengujian terakhir adalah dengan membandingkan kecepatan pengujian dari klaster sistem yang telah dibuat dengan dimana sistem tersebut akan langsung melakukan pengujian tanpa membagi tugas kepada sistem lain. Pengujian ini dilakukan dengan menambahkan waktu tunda pada setiap pengujian untuk mengetahui jika satu pengujian semakin lama, manakah yang bisa menyelesaikan pengujian lebih cepat. Waktu delay yang divariasi adalah 0, 5, 10, 15, 30 dan 45. Pengujian ini akan dilakukan dengan mensimulasikan ada enam pengguna yang melakukan pengujian.

BAB 5 IMPLEMENTASI

5.1 HTTP *Server*

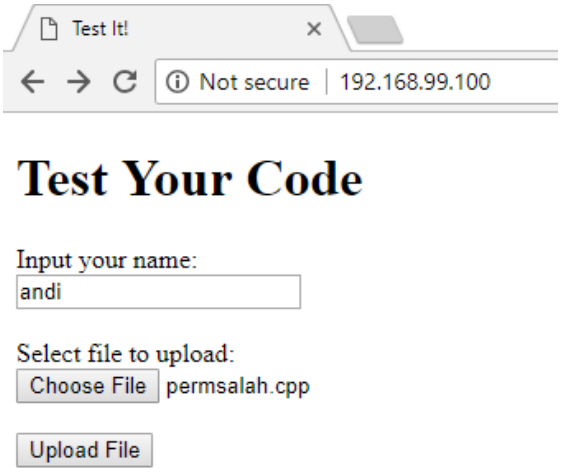
Pada HTTP *server* akan digunakan 3 kode program. Kode program pertama akan digunakan untuk masukan dari pengguna. Pada baris 1-6 digunakan untuk mendefinisikan *title* dari halaman dan juga mengatur halaman web. Lalu baris 7-15 akan digunakan untuk membuat form. Pada form akan terdapat dua masukan dari pengguna yaitu nama dan *file* program yang akan diuji yang terlihat pada baris 10 dan baris 12 sampai 13. Nama akan digunakan untuk membedakan proses pengujian. Untuk masukan akan digunakan form yan satu bertipe text dan satunya lagi adalah *file*. Pada baris 7 akan ditentukan halaman selanjutnya, dimanan saat pengguna menekan tombol *Upload File*, maka inputan akan dikirim menggunakan method post ke kode program selanjutnya untuk menyimpan *file*. *File* akan dicek terlebih dahulu apakah terdapat *file* yang sama, ukurannya cukup dan tipe *file*. Jika semua sudah benar, *file* akan disimpan lalu halaman ini juga akan menyimpan cookie yang akan digunakan untuk kode selanjutnya. Implementasi dapat dilihat pada algoritme 1. Sedangkan tampilan dapat dilihat pada gambar 5.1

Algoritme 1:Implementasi Halaman Utama

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>Test It!</title>
5  </head>
6  <body>
7  <h1> Test Your Code</h1><form action="upload.php" method="post"
8  enctype="multipart/form-data">
9           Input your name:<br>
10          <input type="text" name="nama" id="nama"><br>
11     <br>Select file to upload:<br>
12     <input type="file" name="fileToUpload"
13 id="fileToUpload"><br>
14     <br><input type="submit" value="Upload File" name="submit">
15 </form>
16 </body>
17 </html>
```

Pada program upload *file* / menyimpan *file*, algoritmenya dapat dilihat pada algoritme 2. Pada baris 2-6 nama dari *file* akan diambil dan nama *file* nantinya akan diganti dengan nama yang ada dari form sebelumnya. Pada baris ke-7 dideklarasikan variabel baru dengan nilai satu. Variabel baru ini nantinya akan berubah nilainya menjadi nol saat tidak lolos pengecekan. Pada baris 9-11 akan dilakukan pengecekan apakah *file* dengan nama yang sama sudah ada sebelumnya atau belum. Ini dilakukan sehingga program *websocket server* tidak akan bingung dalam mengambil *file*. Pada baris 13-15 akan dilakukan pengecekan apakah *file* yang akan diunggah tidak melebihi 500.000 *bytes.* Pada baris 17-20 dilakukan pengecekan sehingga *file* yang diunggah hanya *file* dengan bahasa pemrograman C++, Java dan Python. Pada baris 22 dan 23 akan dilakukan seleksi apabila variabel yang tadi dibuat bernilai 0, maka *file* tidak akan tersimpan dan akan menampilkan tulisan permohonan maaf. Pada baris 24-31 merupakan proses untuk menyimpan

file pada direktori yang sebelumnya telah ditentukan dan menyimpan nama serta jenis *file* pada *cookies* dan selanjutnya akan membuka halaman hasil. Baris 32 dan 33 digunakan pada saat jika terjadi kesalahan pada saat mengunggah *file* maka halaman akan menampilkan pesan adanya kesalahan.



Gambar 5.1 Halaman awal

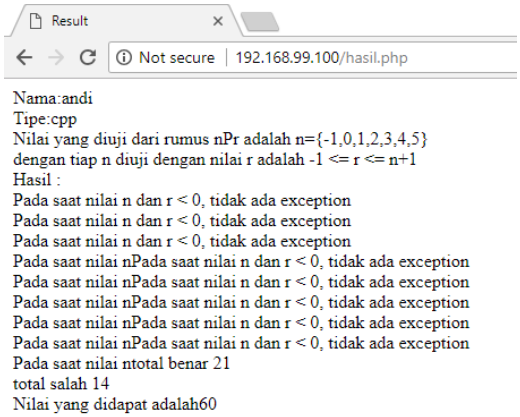
Algoritme 2 : Implementasi menyimpan *file*

```
1   <?php
2   $imageFileType =
3  strtolower(pathinfo($_FILES["fileToUpload"]["name"],PATHINFO_EXTEN
4  SION));
5   $target_dir = "D:\Smt akhir\beneran/";
6   $target_file = $target_dir.$_POST["nama"].".".$imageFileType;
7   $uploadOk = 1;
8
9   if (file_exists($target_file)) {
10      echo "Sorry, file already exists.";
11      $uploadOk = 0;
12  }
13  if ($_FILES["fileToUpload"]["size"] > 500000) {
14      echo "Sorry, your file is too large.";
15      $uploadOk = 0;
16  }
17  if($imageFileType != "py" && $imageFileType != "java" &&
18 $imageFileType != "cpp") {
19      echo "Sorry, ";
20      $uploadOk = 0;
21  }
22  if ($uploadOk == 0) {
23      echo "Sorry, your file was not uploaded.";
24  } else {
25      if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"],
26 $target_file)) {
27          echo "The file ". basename(
28 $_FILES["fileToUpload"]["name"]). " has been uploaded.";
29          setcookie("nama",$_POST['nama']);
30          setcookie("tipe",$imageFileType);
31          header("Location: hasil.php");
32      } else {
33          echo "Sorry, there was an error uploading your file.";
34      }
35  }
36  ?>
```

Pada halaman hasil, pada baris satu dan dua digunakan untuk menentukan tulisan *title*. Selanjutnya antara baris 3-30 merupakan *script* yang akan dijalankan pada saat halaman diakses. Baris 5-8 akan membuka koneksi ke *websocket server* lalu menjalankan fungsi kirim data. Baris 11-13 akan berjalan pada saat ada data masuk, maka fungsi *set* hasil akan berjalan. Baris 15-24 merupakan fungsi kirim data. Fungsi ini akan mengambil data pada *cookies* lalu mengirimkannya ke koneksi yang tadi telah dibuka dalam bentuk *Json*. Baris 26-28 merupakan fungsi untuk menerima data. Pada fungsi ini pada saat terdapat data yang diterima, data tersebut akan ditampilkan pada bagian hasil pada badan halaman. Baris 33 dan 34 akan menampilkan data yang tadi disimpan pada *cookies*. Baris 35-37 merupakan deklarasi dari bagian hasil, dimana pada bagian ini nantinya akan ditampilkan data hasil pengujian kode program yang diterima. Contoh tampilan dapat dilihat pada gambar 5.2. Implementasi dapat dilihat pada algoritme 3.

Algoritme 3: Implementasi Halaman Hasil

```
1     <head>
2         <title>Result</title>
3         <script type="text/javascript">
4
5             ws = new Websocket('ws://server:port/');
6             ws.onopen = function(){
7                 // alert("hello");
8                 kirimData();
9             };
10
11              ws.onmessage = function(evt){
12                      //alert(evt.data);
13                      setHasil(evt.data);
14              };
15            function kirimData(){
16                cook = document.cookie;
17                cookiearray = cook.split('; ');
18                nama = cookiearray[1].split("=")[1];
19                nama = nama.replace(/[+]/g, ' ');
20                tipe = cookiearray[0].split("=")[1];
21                document.getElementById('nama').innerHTML = nama;
22                document.getElementById('tipe').innerHTML = tipe;
23                var nudata= {type:tipe,name:nama};
24                ws.send(JSON.stringify(nudata));
25            }
26              function setHasil(data){
27                      hasil = document.getElementById('hasil');
28                      hasil.innerHTML = data
29              }
30        </script>
31    </head>
32    <body>
33    Nama:<div id="nama" style="display: inline-block"></div><br>
34    Tipe:<div id="tipe" style="display: inline-block"></div><br>
35    Hasil :
36        <div id="hasil" nama="hasil" style="display: contents;">
37        </div>
38 </body>
```



Gambar 5.2 Tampilan Halaman Hasil

5.2 Implementasi Web *Socket Server*

Untuk web *socket server* akan digunakan bahasa python dengan menggunakan beberapa modul. Modul yang digunakan antara lain *twisted*, *txws*, *json*, *os* dan *rpyc*. Pada baris pertama akan dideklarasikan variabel counter untuk *load balancing* dengan nilai awal 0. Lalu pada baris dua akan dibuat fungsi yang akan berjalan saat mendapatkan data dari http *server*. Pada baris tiga dan empat, data dari http *server* akan disimpan pada variabel nama dan bahasa/tipe pengujian. Setelah itu pada baris lima variabel counter akan ditambah satu untuk merepresentasikan bahwa terdapat pengujian baru. Setelah itu pada baris 6-11 nilai dari variabel counter akan dihitung dengan modulo 3 untuk menentukan pada slave yang mana pengujian akan dilakukan yang selanjutnya akan langsung membuka koneksi dengan *slave*. Pada baris dua belas akan membuka *file* sesuai dengan nama dan tipe *file*. Pada baris 13-18 akan menyeleksi sesuai dengan tipe *file* dan akan memanggil fungsi yang sesuai dengan bahasa pemrograman. Setelah itu pada baris 19 hasil yang dikembalikan dari slave akan dikembalikan kepada http *server*. Http *server* nantinya akan menampilkan hasil kepada pengguna. Implementasi dapat dilihat pada algoritme 4.

Algoritme 4: *Websocket Server*

```
1    counter = 0
2     def dataReceived(data):
3            nama = data ["name"]
4            tipe = data ["type"]
5            counter += 1
6            if (counter%3==0):
7                connect(Slave1)
8            elif(counter%3==1):
9                connect(Slave2)
10           else:
11               connect(Slave3)
12           file = open(savedfile)
13           if(tipe == "python"):
14               hasil = python(file)
15           elif(tipe == "java"):
16               hasil java(file)
17           elif(tipe == "cpp"):
18               hasil = cpp(file)
19           return hasil
```

5.3 Implementasi *Slave*

Slave akan dibuat menggunakan bahasa python. Pada program ini terdapat satu fungsi untuk setiap Bahasa yang akan diuji. Sebelumnya telah disimpan terlebih dahulu *file-file* yang akan digunakan untuk menjalankan pengujian. Implementasi dari program dapat dilihat pada algoritme 5.

Pada fungsi untuk menguji bahasa python yang terdapat pada baris 1-6, di baris kedua akan dibuat terlebih dahulu dibuat folder untuk menyimpan hasil dan mengumpulkan semua kode sumber yang dibutuhkan. Setelah itu pada baris 3 dan 4 *file* dari *master* dan *file* yang telah disimpan sebelumya akan dicopy dan disimpan pada folder tersebut. Setelah itu pada baris 5 program penguji akan dijalankan pada docker. Hasilnya nanti akan keluar pada folder tersebut. Pada baris 6 hasil tadi akan dikembalikan ke *websocket server*.

Pada fungsi untuk menguji bahasa Java yang dituliskan pada baris 7-13, sebenarnya hampir sama dengan fungsi untuk bahasa python. Pertama pada baris 8 akan dibuat terlebih dahulu dibuat folder untuk menyimpan hasil dan mengumpulkan semua kode sumber yang dibutuhkan. Setelah itu pada baris 9 dan 10 *file* dari *master* dan *file* yang telah disimpan sebelumya akan dicopy dan disimpan pada folder tersebut. Lalu selanjutnya yang membedakan adalah pada baris 11 program harus dilakukan compile terlebih dahulu. Setelah itu pada baris 12 program penguji baru bisa dijalankan. Terakhir pada baris 13 hasil akan dikembalikan juga kepada *websocket server*.

Pada fungsi untuk menguji bahasa C++ yang terdapat pada baris 14-19, di baris 15 akan dibuat terlebih dahulu dibuat folder untuk menyimpan hasil dan mengumpulkan semua kode sumber yang dibutuhkan. Setelah itu pada baris 16 dan 17 *file* dari *master* dan *file* yang telah disimpan sebelumya akan dicopy dan disimpan pada folder tersebut. Setelah itu pada baris 18 program penguji akan dijalankan pada docker. Hasilnya nanti akan keluar pada folder tersebut. Pada baris 19 hasil tadi akan dikembalikan ke *websocket server*.

Algoritme 5: Slave

```
1     def python(file):
2           system(create new folder)
3           system(copy tester program to folder)
4           system(copy file to folder)
5           system(docker run tester)
6           return hasil
7       def java(file):
8           system(create new folder)
9           system(copy tester program to folder)
10          system(copy file to folder)
11          system(docker compile tester)
12          system(docker run tester)
13          return hasil
14      def cpp(file):
15          system(create new folder)
16          system(copy tester program to folder)
17          system(copy file to folder)
18          system(docker run tester)
19          return hasil
```

Pada docker di tiap *slave*, akan dipasang tiga image yang berbeda menggunakan perintah pull. Image yang dipasang adalah python versi 2-slim dengan perintah pada baris 1, gcc versi 7 dengan perintah pada baris 2 dan openjdk versi 11-slim dengan perintah pada baris 3. Perintah lengkap dapat dilihat pada algoritme 6.

Kode Sumber 5.1 Memasang Image pada Docker

Algoritme 6: Instalasi Docker Image	

1
2
3
	`docker pull python:2-slim`
	`docker pull gcc:7`
	`docker pull openjdk:11-slim`

BAB 6 PENGUJIAN DAN ANALISIS

6.1 Pengujian Fungsional

Pengujian fungsional merupakan pengujian secara *black-box* yang dilakukan untuk menguji fungsional sistem apakah sudah berjalan dengan baik dan benar. Pengujian dilakukan dengan interaksi antara *user-master* dan *master-slave*. Pengujian ini dianggap berhasil jika semua fungsi yang dipaparkan tidak menunjukkan adanya kegagalan dan kesalahan dalam pengujian. Skenario pengujian dibuat berdasarkan kebutuhan yang ada. Skenario pengujian dapat dilihat pada tabel 6.1.

Pengujian ini dilakukan dengan cara menguji kode program langsung. Kode program yang akan diujikan adalah untuk menghitung total permutasi. Ada dua kode program yang akan digunakan, dimana satu kode program benar sedangkan kode satunya salah. Kode program ini akan diuji menggunakan program uji dimana didalamnya terdapat beberapa inputan yang akan diujikan. Inputannya adalah nilai n dari -1 sampai dengan 5 dan masing masing nilai n diuji dengan nilai r, dimana nilai r adalah -1 sampai n+1. Setelah semua masukan dijalankan dan diuji, hasil dari setiap masukan akan ditulis pada sebuah *file*. *File* ini nantinya akan dibaca dan ditampilkan. Untuk kode program permutasi dapat dilihat pada algoritme 7 dan 8, sedangkan untuk kode program penguji dapat dilihat pada algoritme 9.

Algoritme 7: Permutasi yang benar

```
1     def factorial(a):
2         fact = 1
3         if a<0:raise ValueError("Nilai harus >=0")
4         elif a == 0:return 1
5         else:
6             for i in range (1,a+1):
7                 fact = fact * i
8             return fact
9
10    def permutation(n,r):
11        if n<r:raise ValueError("Masukkan nilai n > r")
12        elif n<0 or r<0:raise ValueError("Nilai harus >=0")
13        else:answer = factorial(n) / factorial(n-r)
14         return answer
```

Algoritme 8: Permutasi yang salah

```
1     def factorial(a):
2         fact = 1
3         for i in range (1,a+1):
4             fact = fact * i
5         return fact
6
7     def permutation(n,r):
8         answer = factorial(n) / factorial(n-r)
9      return answer
```

Algoritme 9: Penguji Kode

```
1       import permutation
2       benar = 0
3       salah = 0
4       kesalahan = ""
5       n = [-1,0,1,2,3,4,5]
6       r  = {}
7       r [-1]=[0,0]
8       r [0]=[0,1,0]
9       r [1]=[0,1,1,0]
10      r [2]=[0,1,2,2,0]
11      r [3]=[0,1,3,6,6,0]
12      r [4]=[0,1,4,12,24,24,0]
13      r [5]=[0,1,5,20,60,120,120,0]
14      for i in n:
15          for j in range(-1,i+2):
16              try:
17                  perm = permutation.permutation(i,j)
18                  if i == -1 or j == -1:
19                      salah += 1
20      kesalahan += "pada n= "+str(i)+" dan r= "+str(j)+" Harus lebih
21  dari 0\n"
22                  elif i < j:
23                      salah += 1
24          kesalahan += "pada n= "+str(i)+" dan r= "+str(j)+ " n>r\n"
25                  elif perm == r [i][j+1]:
26                      benar += 1
27                  else:
28                      salah+=1
29          kesalahan += "pada n= " + str(i) + " dan r= " + str(j) + "
30  Nilai seharusnya "+str(r [i][j+1])+" tetapi hasilnya"+str(perm)
31              except ValueError as e:
32                  benar += 1
33      tulis = open("hasil.txt","w+")
34      tulis.write(kesalahan)
35      tulis.write("total benar "+str(benar)+" total salah
36  "+str(salah)+" dan nilai adalah "+str((benar/35.0)*100))
37  tulis.close()
38
```

Tabel 6.1 Skenario Pengujian

No	Fungsi	Skenario
1	Menerima data dari pengguna	1. Pengguna mengisi data pada form di halaman awal dan mengunggah *file*. 2. Data tersimpan pada *master*
2	*Master* melakukan *load balancing*	1. Pengguna melakukan proses pengujian berulang kali 2. Seleksi kondisi untuk *load balancing* berjalan
3	*Master* melakukan koneksi ke *slave*	1. *Master* membuka koneksi ke *slave* 2. Tidak terjadi error

4	*Master* meneruskan hasil ke pengguna	1. *Master* menerima data dari *slave* 2. *Master* mengirimkan data pada halaman hasil
5	*Slave* menerima koneksi dari *master*	1. *Master* mencoba koneksi ke *slave* 2. Tidak terjadi error
6	*Slave* menjalankan fungsi yang dipanggil oleh *master*	1. *Master* memanggil fungsi pada *slave* 2. Proses pengujian dapat berjalan
7	*Slave* mengirim hasil pengujian kepada	1. *Slave* membaca hasil pengujian 2. Pada akhir fungsi *slave* mengembalikan hasil

6.1.1 Hasil Pengujian Fungsional

Dari skenario pengujian yang dilakukan didapatkan hasil seperti pada tabel 6.2.

Tabel 6.2 Hasil Pengujian Fungsional

No	Kebutuhan Fungsional	Hasil
1	*Master* dapat menerima data/*file* dari pengguna	Berhasil
2	*Master* dapat melakukan *load balancing* untuk memilih *slave*.	Berhasil
3	*Master* dapat memanggil fungsi yang terdapat pada *slave*.	Berhasil
4	*Master* dapat menerima hasil pengujian dari *slave* dan menampilkannya ke pengguna	Berhasil
5	*Slave* dapat menerima koneksi dari *master*.	Berhasil
6	*Slave* dapat menjalankan pengujian saat *master* memanggil fungsi yang ada.	Berhasil
7	*Slave* dapat mengirimkan hasil pengujian kepada *master*	Berhasil

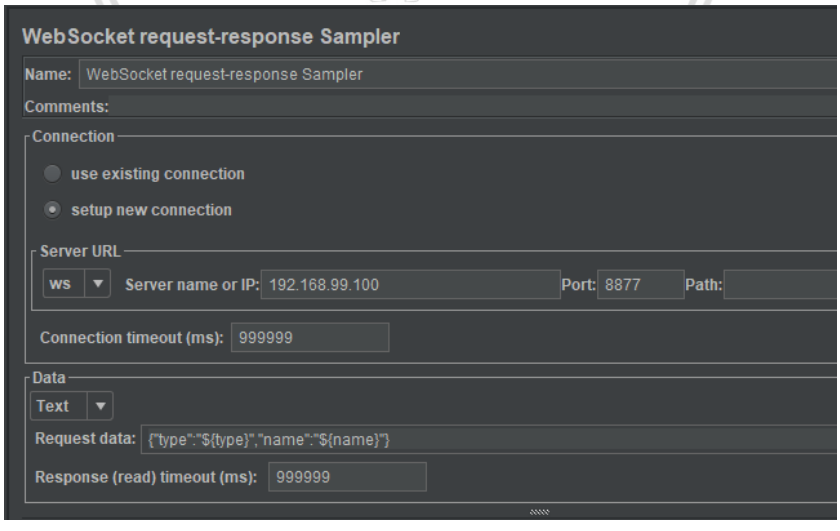
6.1.2 Analisis Pengujian Fungsional

Berdasarkan hasil, setiap fungsi yang ada dapat berfungsi dengan benar dan proses pengujian kode program dapat berjalan. Pengguna sudah dapat melakukan proses mengunggah *file* dan *master* dapat menerima *file* tersebut dapat dibuktikan pada pengujian pertama. Setelah itu proses *load balancing* juga dapat dilakukan dilihat dari hasil pengujian nomor 2. Proses selanjutnya yaitu *slave*

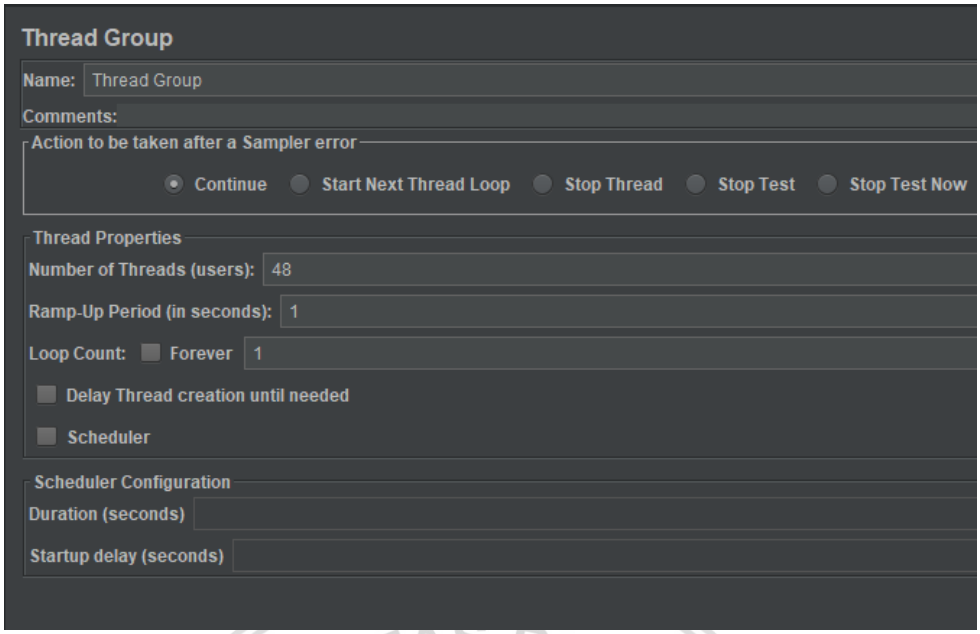
menerima koneksi juga dapat berjalan sesuai dengan pengujian nomor 5. Pada pengujian nomor tiga proses selanjutnya yaitu *master* memanggil fungsi dapat berjalan dengan baik. Setelahnya, proses pengujian yang dilakukan oleh *slave* juga dapat berjalan, ini dibuktikan dengan pengujian nomor 6. Pengembalian hasil kepada pengguna juga sudah dapat berjalan, ini dibuktikan dengan pengujian nomor tujuh dan nomor empat. Dengan hasil ini maka dapat dinyatakan bahwa seluruh proses dapat berjalan dengan baik sesuai dengan urutan prosesnya yang telah dijabarkan barusan, termasuk pengujian kode program menggunakan docker yang dimulai saat *master* memanggil fungsi pada *slave*, kemudian *slave* menjalankan pengujian. Selain itu dengan berhasilnya pengujian pada nomor 3, 4, 5 dan 7 maka menandakan bahwa komunikasi antara *master* dengan *slave* yang dibangun menggunakan RPC dapat berjalan sehingga *master* dapat terhubung dengan *slave* dan memanggil fungsi yang telah disediakan oleh *slave*.

6.2 Pengujian *Load balancing*

Pengujian *Load balancing* digunakan untuk mengetahui apakah sistem dapat membagi proses pengujian dengan menggunakan algoritme *round robin*. Pengujian dilakukan dengan cara melakukan variasi pada jumlah pengguna yang akan melakukan pengujian. Pengujian ini akan membutuhkan plugin tambahan pada jMeter yaitu *websocket request-response sampler*. Program yang akan diuji adalah program permutasi yang telah diberi nama angka 1-120 dengan bahasa pemrograman yang berbeda-beda. Untuk konfigurasi plugin, akan disesuaikan alamat ip dan port sesuai dengan alamat dan port dari *server* yang berjalan. Contoh konfigurasi terdapat pada gambar 6.1. Untuk konfigurasi thread, pada bagian jumlah thread akan diganti sesuai dengan jumlah pengguna yang akan divariasi. Contoh konfigurasi terdapat pada gambar 6.2. Pada akhir dari hasil pengujian akan ditambah kalimat dari *server* mana pengujian berlangsung. Kalimat tersebut yang akan diamati dan dihitung berapa kali kalimat tersebut muncul.



Gambar 6.1 Konfigurasi Plugin *Websocket*



Gambar 6.2 Konfigurasi Thread

6.2.1 Hasil Pengujian *Load balancing*

Hasil pengujian dapat dilihat pada tabel 6.3

Tabel 6.3 Hasil Pengujian *Load balancing*

No	Jumlah User	Jumlah Pengujian yang Dilakukan			Error
		Slave 1	*Slave* 2	*Slave* 3	
1	6	2	2	2	0
2	12	4	4	4	0
4	24	8	8	8	0
5	48	16	16	16	0
6	60	20	20	20	0
7	96	32	32	32	0
8	120	35	34	35	16

6.2.2 Analisis Pengujian *Load balancing*

Berdasarkan hasil pengujian pada tabel 6.3, dapat dilihat bahwa jumlah pengujian antar *slave* hampir sama. Pada saat terdapat enam pengujian, pada tiap *slave* mendapatkan dua pengujian, pada saat terdapat 12 pengujian, pada tiap *slave* mendapatkan empat pengujian, begitu seterusnya sampai pada jumlah pengujian ada 96 dengan tiap *slave* menguji 32 kode program. Hal ini membuktikan bahwa *master* mampu mendistribusi proses pengujian antar *slave* dengan baik berdasarkan algoritme *round robin.* Sedangkan pada saat pengguna

mencapai 120, sistem hanya mampu menerima sekitar 104 pengguna dan tetap dibagi hampir rata, 35 untuk *slave* satu, 34 untuk *slave* dua serta 35 untuk *slave* tiga. Pengguna lainnya yang berjumlah 16 mengalami timeout. Setelah dicoba dengan menambahkan *core* dari *processor* menjadi dua *core*, jumlah error berkurang. Berarti dapat disimpulkan untuk jumlah *core* satu dan memori 1024 MB, jumlah pengguna dalam satu waktu yang dapat melakukan pengujian adalah 104. Jika terdapat lebih dari 104 pengguna maka *resource* perlu ditambah.

6.3 Pengujian Penggunaan *Resource*

Pengujian penggunaan *resource* dilakukan untuk mengetahui seberapa besar *resource* yang digunakan pada saat ada pengguna yang ingin melakukan pengujian. Pengujian dilakukan dengan cara melakukan variasi pada pengguna. Hal yang diamati adalah hasil monitoring *resource*. Data yang diambil merupakan nilai rata-rata penggunaan selama proses pengujian berlangsung. Nilai rata-rata ini nantinya akan dikurangi dengan nilai awal dari sebelum pengujian, untuk mengetahui pada saat melakukan pengujian, berapa banyak *resource* yang dibutuhkan. Untuk mengambil hasil, digunakan tools mpstat dan sar. Perintah untuk mpstat dan sar akan disesuaikan dengan lamanya hasil pengujian. Pada mpstat nilai yang diambil merupakan rata-rata dari kolom "%usr". Contoh dapat dilihat pada gambar 6.3. Sedangkan pada tools sar, nilai yang diambil merupakan rata-rata dari kolom "memused". Contoh dapat dilihat pada gambar 6.4.

```
ferdicezano@santosa:~$ mpstat 1 3
Linux 4.4.0-87-generic (santosa)        07/21/2018      _x86_64_        (1 CPU)

11:26:43 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
11:26:44 AM  all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
11:26:45 AM  all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
11:26:46 AM  all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
Average:     all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
```

Gambar 6.3 Mpstat

```
ferdicezano@santosa:~$ sar -r 1 3
Linux 4.4.0-87-generic (santosa)        07/21/2018      _x86_64_        (1 CPU)

11:26:51 AM kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
11:26:52 AM    132248    883848     86.98     97648    379052   1709112     82.86    437692    287308         0
11:26:53 AM    132248    883848     86.98     97648    379052   1709112     82.86    437748    287308         0
11:26:54 AM    132248    883848     86.98     97648    379052   1709112     82.86    437748    287308         0
Average:       132248    883848     86.98     97648    379052   1709112     82.86    437729    287308         0
```

Gambar 6.4 Sar

Untuk membandingkan penggunaan *resource* akan dibuat satu sistem, dimana sistem tersebut akan menerima data dari pengguna dan langsung diuji pada sistem tersebut tanpa dibagi ke sistem lain. Setelah itu hasilnya akan langsung dikembalikan. Pembuatan sistem ini digunakan untuk mengetahui lebih efisien mana penggunaan klaster sistem atau penggunaan sistem biasa. Untuk algoritme dapat dilihat pada algoritme 10 dibawah. Pada pengujian ini akan diacak bahasa yang akan diuji karena beban pengujian untuk tiap bahasa tidaklah sama. Pengujian paling berat adalah Java karena java harus melakukan *compile* terlebih dahulu. Pengujian yang lebih ringan selanjutnya dengan urutan berat-ringan berturut-turut adalah C++ dan python.

```
Algoritme 10: Permutasi yang benar
1       def dataReceived(data):
2           nama = data ["name"]
3           tipe = data ["type"]
4           file = openfile()
5           if(tipe==py):
6               system(create new folder)
7               system(copy tester program to folder)
8               system(copy file to folder)
9               system(docker run tester)
10              return hasil
11          elif(tipe==java):
12              system(create new folder)
13              system(copy tester program to folder)
14              system(copy file to folder)
15              system(docker compile tester)
16              system(docker run tester)
17              return hasil
18          elif(tipe==cpp):
19              system(create new folder)
20              system(copy tester program to folder)
21              system(copy file to folder)
22              system(docker run tester)
23              return hasil
24          file.close()
```

6.3.1 Hasil Pengujian Penggunaan *Resource*

Berdasarkan skenario pengujian, maka didapatkan hasil pengujian yang dapat dilihat pada tabel 6.4 untuk hasil pengujian pada klaster sistem dan tabel 6.5 untuk hasil pengujian pada sistem biasa.

Tabel 6.4 Hasil Pengujian Kehandalan Klaster Sistem

No	Jumlah User	CPU *Usage* (%)				*Memory Usage* (%)			
		Slave 1	*Slave* 2	*Slave* 3	*Master*	*Slave* 1	*Slave* 2	*Slave* 3	*Master*
1	6	41	54	48	0,3	2	2	2	0,02
2	12	63	57	50	0,5	3	3	2	0,03
3	24	51	54	50	0,3	2	2	3	0,06
4	48	54	55	55	0,2	2	3	2	0,09
5	60	53	50	56	0,2	2	2	2	0,1
6	96	47	47	49	0,2	3	3	4	0,22

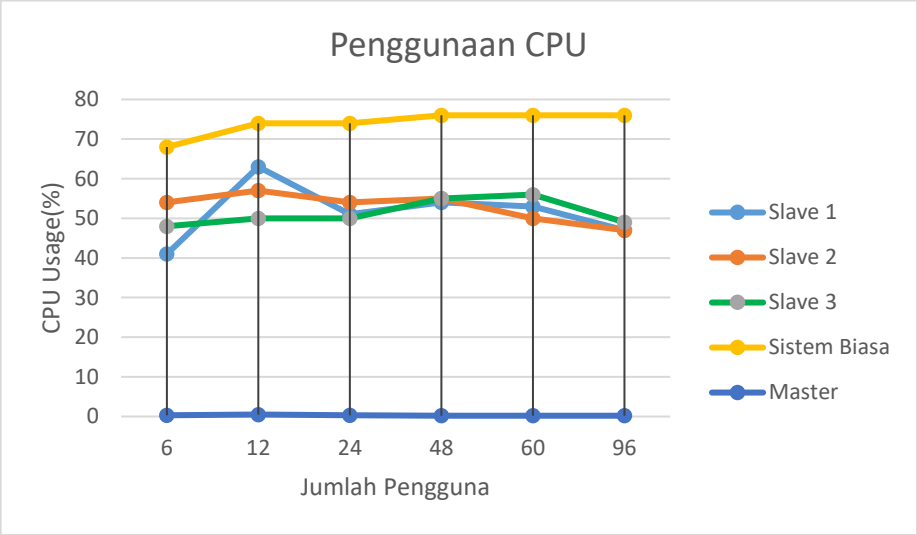
Tabel 6.5 Hasil Pengujian Kehandalan pada Sistem Biasa

NO	Jumlah user	CPU *Usage* (%)	*Memory Usage* (%)
1	6	68	2
2	12	74	3

3	24	74	3
4	48	76	3
5	60	76	4
6	96	76	4,5

6.3.2 Analisis Pengujian Penggunaan *Resource*

Hasil penggunaan CPU setelah digambarkan dengan menggunakan grafik dapat dilihat pada gambar 6.5.



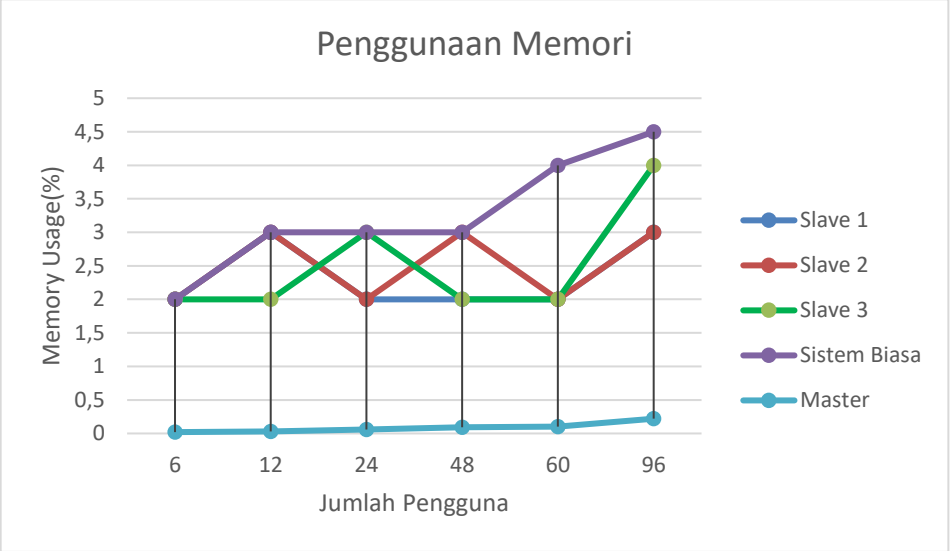
Gambar 6.5 Grafik Penggunaan CPU

Berdasarkan hasil pengujian, dapat dilihat bahwa sistem biasa selalu memiliki penggunaan yang lebih tinggi daripada klaster sistem yang menggunakan tiga penguji. Ini terjadi karena pada klaster sistem membagi pengujian sehingga masing-masing slave tidak terlalu banyak menguji program yang lebih berat seperti menguji java yang perlu dilakukan *compile* terlebih dahulu untuk mendapatkan hasil. Sedangkan pada sistem biasa karena sistem ini melakukan semua pengujian sekaligus, maka rata-rata penggunaan CPU otomatis akan lebih tinggi karena sistem ini harus menjalankan semua program yang berat. *Master* memiliki penggunaan CPU yang rendah karena *master* hanya bertugas menyalurkan kode dan hasil yang tidak memerlukan penggunaan CPU yang tinggi.

Selain itu dari hasil dapat dilihat bahwa pada awalnya antara slave terdapat jarak yang lumayan besar. Tetapi dengan semakin bertambahnya pengujian, perbedaan penggunaan CPU pada tiap slave berkurang. Ini dikarenakan pada saat pengguna masih sedikit, beban pengujian yang dilakukan oleh tiap slave tidak sama. Ada slave yang hanya menguji bahasa yang berat seperti java dan ada juga slave yang hanya menguji bahasa yang ringan seperti python dan C++. Pada saat pengguna semakin banyak, maka semakin bervariasi pengujian yang dilakukan oleh tiap *slave*. Inilah yang menyebabkan akhirnya penggunaan CPU antara slave

hampir sama. Selain itu karena bervariasinya pengujian yang dilakukan, maka semakin banyak pengujian semakin berkurang dan semakin stabil juga rata-rata penggunaan CPU pada tiap *slave*.

Hasil penggunaan memori setelah digambarkan dengan menggunakan grafik dapat dilihat pada gambar 6.6.



Gambar 6.6 Grafik Penggunaan Memori

Berdasarkan pengujian yang dilakukan, pada sistem biasa penggunaan memori terus naik saat semakin banyak pengguna. Pada klaster penggunaan memori cenderung stabil antara 2 dan 3, hanya pada saat pengguna mencapai 96 salah satu *slave* mengalami kenaikan. Ini dikarenakan setiap bahasa pemrograman memiliki penggunaan memori yang hampir sama. Untuk *master* penggunaan memori memang kecil karena *master* tidak melakukan pengujian yang memerlukan penggunaan memori yang banyak.

Berdasarkan kedua pengujiaan, dapat disimpulkan bahwa dengan menggunakan klaster, pengujian akan lebih menghemat *resource*. Selain itu penggunaan *resource* dari klaster akan stabil dan hanya bertambah sedikit, bahkan penggunaan CPU justru berkurang jika pengguna semakin banyak. Ini membuktikan bahwa dengan menggunakan klaster pengujian akan menjadi lebih efisien.

6.4 Pengujian Kecepatan

Pengujian ini dilakukan untuk mengetahui bagaimana performa klaster yang dibuat pada saat menguji program yang butuh berjalan dengan waktu yang lama. Pengujian dilakukan dengan membandingkan kecepatan pengujian dari klaster dengan sistem biasa dimana sistem tersebut akan menerima data dari pengguna dan langsung diuji pada sistem tersebut tanpa dibagi ke sistem lain. Sistem biasa yang digunakan pada pengujian ini sama dengan sistem biasa yang sudah dibuat pada pengujian sebelumnya. Pengujian ini akan disimulasikan dengan menggunakan enam pengguna. Variabel bebas yang digunakan adalah lama waktu

program pengujian berjalan. Untuk membuat variabel bebas ini pada kode program akan ditambahkan kode untuk menunda program. Penundaan ini akan divariasi antara 0, 5, 15, 30 dan 45 detik.

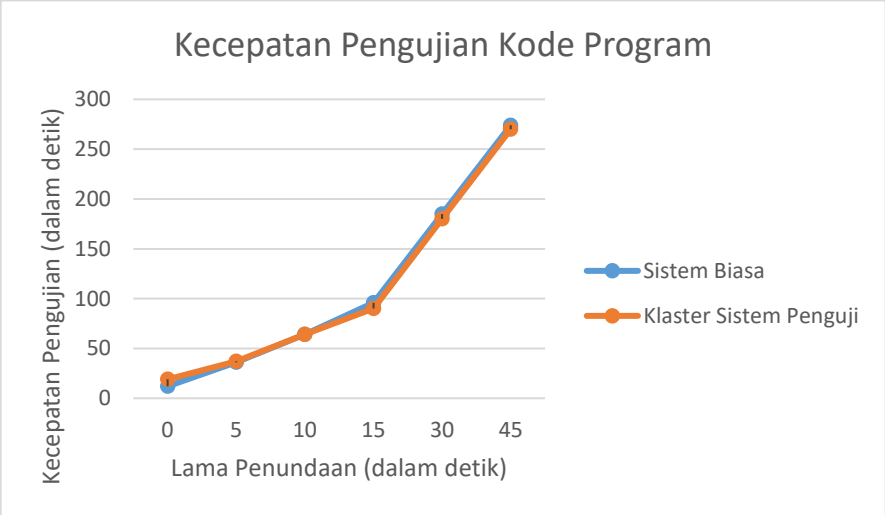
6.4.1 Hasil Pengujian Kecepatan

Hasil dari pengujian kecepatan dapat dilihat pada tabel 6.6.

No.	Lama Penundaan Program (detik)	Sistem Biasa	Klaster Sistem Penguji
1	0	12	19
2	5	36	37
3	10	64	64
4	15	96	90
5	30	185	180
6	45	274	270

6.4.2 Analisis Pengujian Kecepatan

Hasil pengujian kecepatan setelah digambarkan dengan menggunakan grafik dapat dilihat pada gambar 6.7.



Gambar 6.7 Grafik Kecepatan Pengujian Kode Program

Berdasarkan hasil dapat dilihat pada saat program yang diuji membutuhkan waktu berjalan yang sedikit, sistem biasa memilki keunggulan. Ini dikarenakan pada klaster terdapat waktu untuk menyalurkan *file* kode program dan hasil, sehingga proses pengujian menjadi lebih lama. Tetapi pada saat program yang diuji membutuhkan waktu yang lama, klaster sistem penguji menjadi lebih cepat. Hal ini karena waktu yang terbuang pada saat *file* program dan hasil disalurkan antara *master* dengan *slave* tertutupi dengan kecepatan pengujian yang dilakukan, karena klaster sistem penguji memiliki 3 slave yang dapat berjalan bersamaan.

BAB 7 PENUTUP

7.1 Kesimpulan

Berdasarkan penelitian yang telah dilakukan maka diperoleh kesimpulan sebagai berikut:

1. Klaster sistem dapat melakukan pengujian pada docker dengan cara memanggil fungsi yang terdapat pada *slave*. Untuk memulai proses pengujian, pengguna harus terlebih dahulu mengunggah *file* yang akan diuji dan memasukkan nama pengguna. Setelah itu *file* yang sudah diunggah akan diakses oleh *master* dan kemudian *master* memanggil fungsi pada *slave* untuk menjalankan pengujian pada docker.
2. *Master* dan *slave* dapat saling berkomunikasi dengan menggunakan protokol RPC yang ada pada python yaitu RPyC. Dengan menggunakan protokol ini *master* dapat terhubung dengan *slave* dan memanggil fungsi yang telah disediakan oleh *slave*.
3. *Master* mampu mendistribusikan pengujian pada tiga *slave* yang ada dengan menggunakan algoritme *round-robin*. Pada saat *master* hanya memiliki satu *core prosesor* dan 1024 MB, *master* dapat melakukan sampai dengan 104 pengujian pada saat yang bersamaan dan semua pengujian terbagi rata antar *slave*. Dengan menggunakan klaster sistem ini pengujian menjadi lebih efisien dari sisi penggunaan CPU. Selain itu semakin program membutuhkan waktu yang lama maka klaster sistem lebih cepat melakukan pengujian kode program.

7.2 Saran

Pada penelitian ini masih banyak kekurangan sehingga terdapat beberapa saran untuk pengembangan dari penelitian ini, antara lain adalah:

1. Klaster masih dapat dikembangkan dengan memperhatikan keamanan dari proses pengiriman data.
2. Penelitian berikutnya dapat menggunakan database sehingga bisa menambahkan fitur untuk melihat semua nilai/pengujian.
3. Tampilan untuk mengunggah *file* dapat diperbaiki menjadi lebih menarik dan interaktif.

DAFTAR PUSTAKA

Anderson, M. 2017. *What is Load balancing?*. [online] Tersedia di: <https://www.digitalocean.com/community/tutorials/what-is-load-balancing> [Diakses 16 Januari 2018]

Apache Software Foundation. 2018. *Apache JMeter™.* [online] Tersedia di: < https://jmeter.apache.org/> [Diakses 22 Juli 2018]

Nathasya. 2016. *Tutorial Docker dalam Bahasa Indonesia*. [online] Tersedia di: <https://www.dewaweb.com/tutorial-docker-dalam-bahasa-indonesia/> [Diakses 16 Januari 2018]

Docker Enterprise. 2018. *What is a Container*. [online] Tersedia di: <https://www.docker.com/what-container> [Diakses 14 Januari 2018]

Filiba, T. 2018. *RPyC - Transparent, Symmetric Distributed Computing.* [online] Tersedia di <https://rpyc.readthedocs.io/en/latest/> [Diakses 22 Juli 2018]

Godard, S. 2018. *SYSSTAT DocumentationI*. [online] Tersedia di: <http://sebastien.godard.pagesperso-orange.fr/documentation.html> [Diakses 22 Juli 2018]

Heath, N. 2017. *Five highly-paid and in-demand programming languages to learn in 2018*. [online] Tersedia di: <https://www.techrepublic.com/article/five-highly-paid-and-in-demand-programming-languages-to-learn-in-2018/> [Diakses 16 Januari 2018]

McFarlin, T. 2012. *The Beginner's Guide to Unit Testing: What Is Unit Testing?*. [online] Tersedia di: <https://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728> [Diakses 16 Januari 2018]

Pathirathna, P.P.W., Et al. 2017. *Security Testing as a Service with Docker Containerization*. 2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA). [e-journal] 10.1109/SKIMA.2017.8294109. Tersedia melalui: IEEE < https://ieeexplore.ieee.org/document/8294109/> [Diakses 19 Juli 2018]

Programiz. 2018. *Learn Python Programming*. [online] Tersedia di: <https://www.programiz.com/python-programming> [Diakses 16 Januari 2018]

Python Software Foundation. 2018. *Unit testing framework*. [online] Tersedia di: <https://docs.python.org/2/library/unittest.html> [Diakses 14 Januari 2018]

Rouse, M. 2018. *Sandbox*. [online] Tersedia di: <http://searchsecurity.techtarget.com/definition/sandbox> [Diakses 14 Januari 2018]

Saraf, P., Et al. 2015. *Automatic Evaluation System for Student Code. International Journal of Computer Science and Information Technologies*, [online] Tersedia

di: <ijcsit.com/docs/Volume%206/vol6issue02/ijcsit20150602211.pdf> [Diakses 14 Januari 2018]

Sîrbu, N. 2017. *Docker – the solution for isolated environments.* [online] Tersedia di: < http://isd-soft.com/tech_blog/docker-solution-isolated-environments/ > [Diakses 19 Januari 2018]

Špaček, F., Sohlich, R. & Dulík, T. 2015. *Docker as Platform for Assignments Evaluation*. *Procedia Engineering*, [e-journal] 100. Tersedia melalui: ScienceDirect <https://www.sciencedirect.com/science/article/pii/S1877705815005688> [Diakses 09 Januari 2018]