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

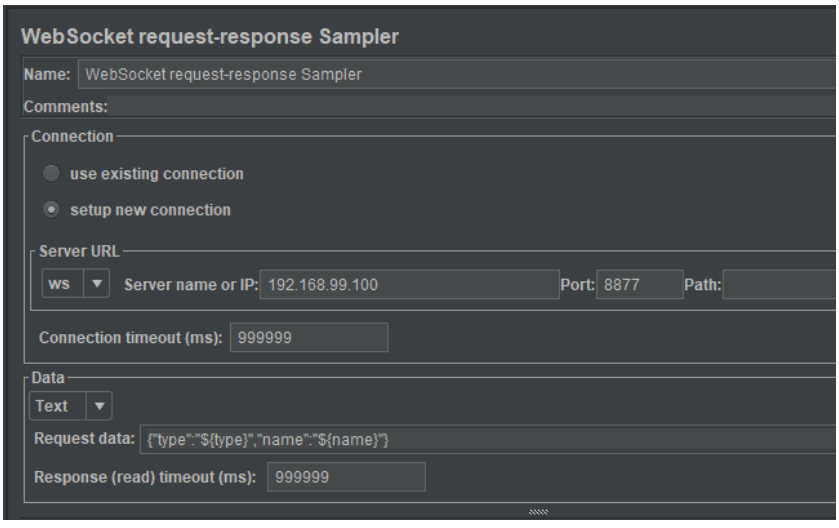
6.1.2 Analisis Pengujian Fungsional

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*

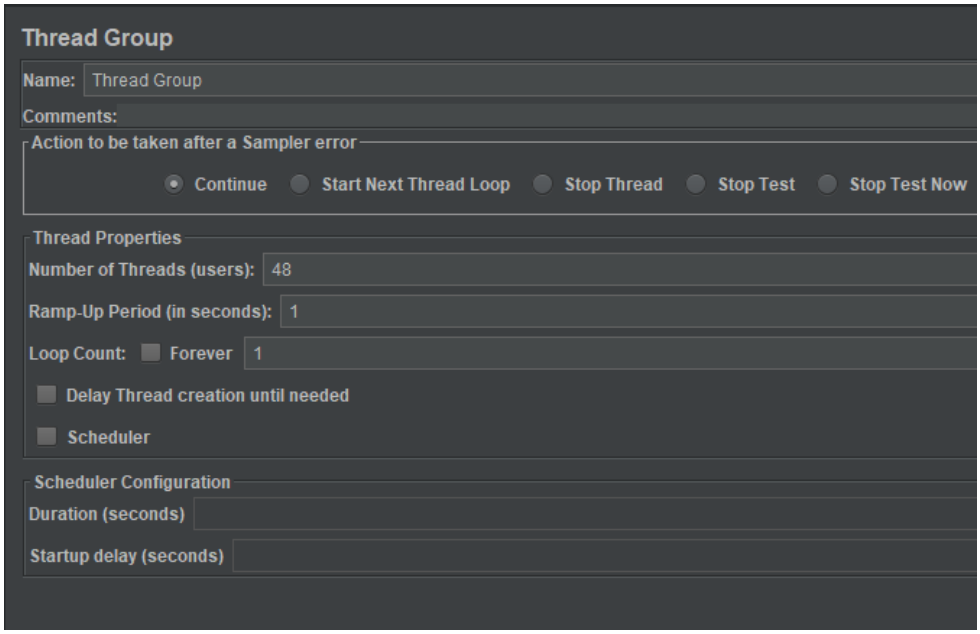
menerima koneksi juga dapat berjalan sesuai dengan pengujian nomor 5. Pada pengujian nomor tiga proses selanjutnya yaitu *master* memanggil fungsi dapat berjalan dengan baik. Setelahnya, proses pengujian yang dilakukan oleh *slave* juga dapat berjalan, ini dibuktikan dengan pengujian nomor 6. Pengembalian 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

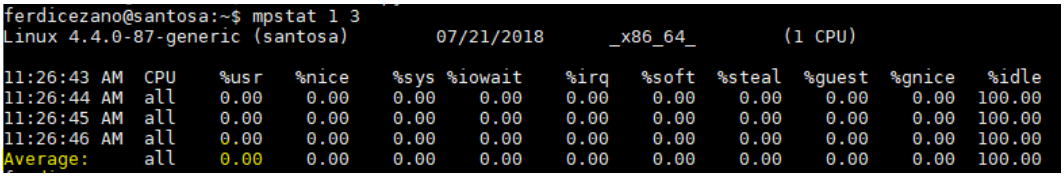
6.2.2 Analisis Pengujian *Load balancing*

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*

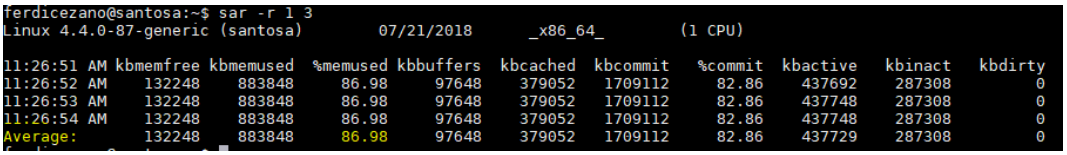
Pengujian penggunaan *resource* dilakukan untuk mengetahui seberapa besar *resource* yang digunakan pada saat ada pengguna yang ingin melakukan pengujian. Pengujian dilakukan dengan cara melakukan variasi pada pengguna. Hal yang diamati adalah hasil monitoring *resource*. Data yang diambil merupakan 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

```
    def dataReceived(data):
        nama = data ["name"]
        tipe = data ["type"]
        file = openfile()
        if(tipe==py):
            system(create new folder)
            system(copy tester program to folder)
            system(copy file to folder)
            system(docker run tester)
            return hasil
        elif(tipe==java):
            system(create new folder)
            system(copy tester program to folder)
            system(copy file to folder)
            system(docker compile tester)
            system(docker run tester)
            return hasil
        elif(tipe==cpp):
            system(create new folder)
            system(copy tester program to folder)
            system(copy file to folder)
            system(docker run tester)
            return hasil
        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

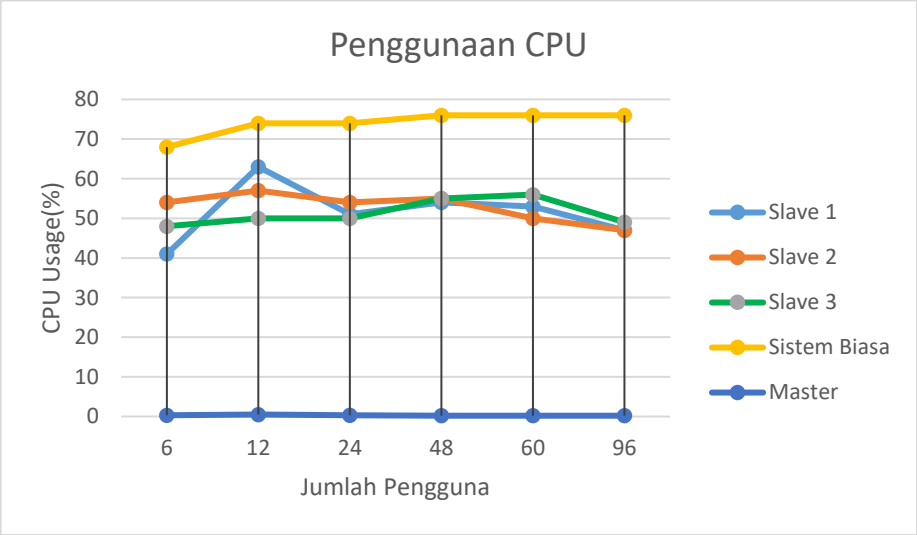
Tabel 6.5 Hasil Pengujian Kehandalan pada Sistem Biasa

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.



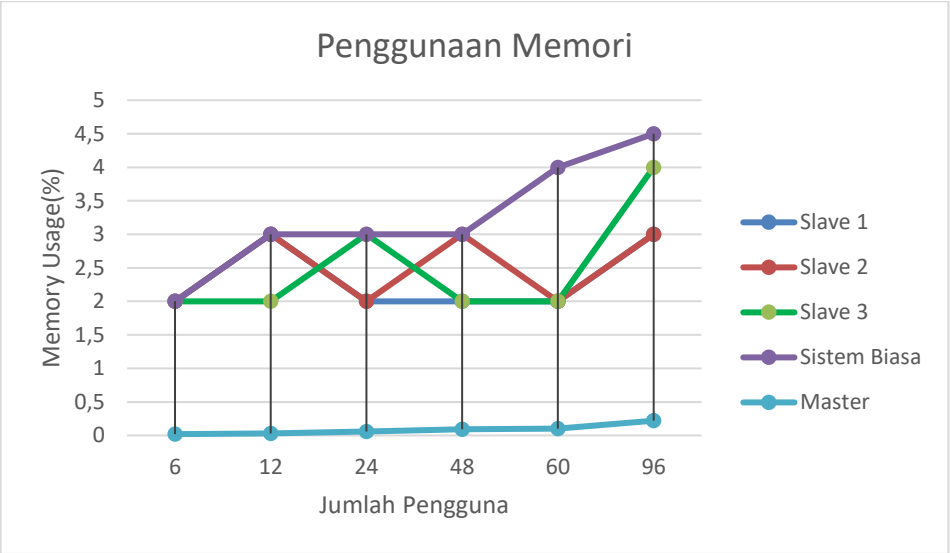
Gambar 6.5 Grafik Penggunaan CPU

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.

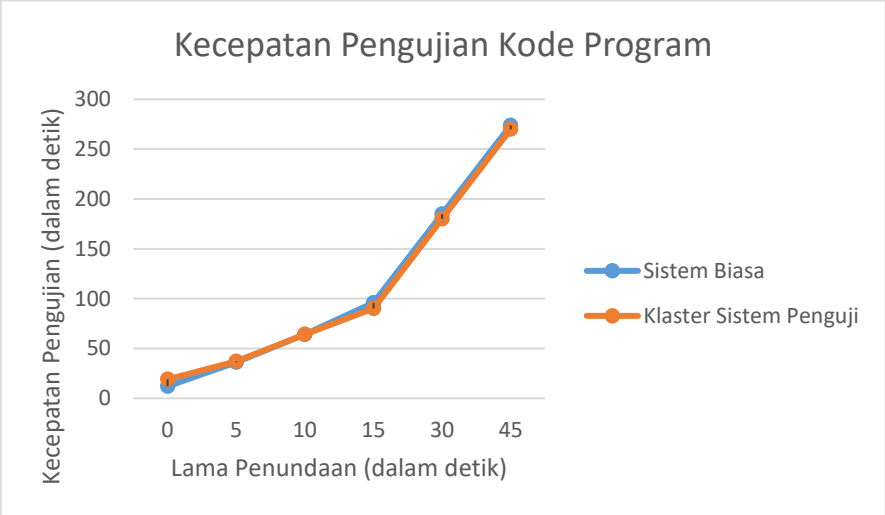
6.4.1 Hasil Pengujian Kecepatan

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.