2nd chapter. Modelling and Simulation. Individual Work

Kęstutis Daugėla

02/01/2022

Task:

```
1. (0,5 balo) Parenkama reali sistema (procesas, elgsena) iš savo mokslo srities.
Pateikiamas trumpas aprašas. Formuluojama hipotezė, problema, reiškinys, kas bus tiriama
2. (0,5 balo) Apibrėžiamas tam tikras sistemos abstrakcijos (detalumo) lygis.
Tikslinama hipotezė, problema, reiškinys, kas bus tiriama
3. (2 balai) Formuluojamas konceptualus modelis, apibrėžiami modelio parametrai,
imitavimo parametrai, tyrimo scenarijai, kintamieji
4. (2 balai) Sudaromas pusiau formalus (loginis) arba formalus modelis
5. (4 balai) Kompiuteryje realizuojamas imitacinis modelis:
   5.1. Imitacinis modelis (grafinis arba kodas) ir imitavimo parametrai bei kintamieji
   5.2. Įvesties ir išvesties analizė
   5.3  Verifikavimo atvejai
6. (1 balas) Daromos tyrimo išvados: atsakoma į iškeltą hipotezę, aptariamas reiškinys,
jo jautrumas parametrams, aptariamos modelio validavimo galimybės ir pan.
7. (Taip / Ne) Paruošiama darbo ataskaita ir įkeliama į Moodle.
```

Load balancing process

Load balancing is a term that refers to the process of distributing a set of jobs among a set of resources (computing units) with the goal of increasing the overall efficiency of their processing. Load balancing can improve response time and prevent some computing nodes from being overloaded while others remain idle.

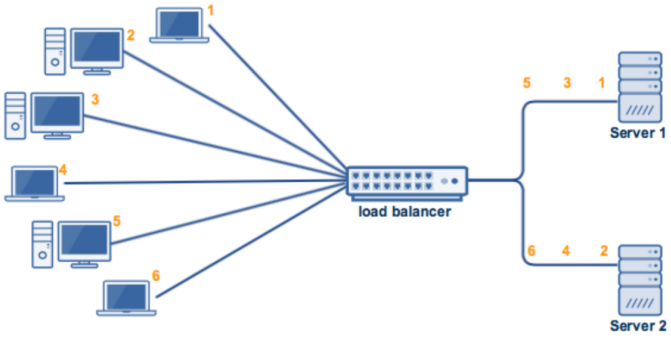


Figure 1: Process diagram

There are numerous load balancing algorithms, both dynamic and static (source - cloudflare)

Dynamic:

- Least connection: Checks which servers have the fewest connections open at the time and sends traffic to those servers. This assumes all connections require roughly equal processing power.

- Weighted least connection: Gives administrators the ability to assign different weights to each server, assuming that some servers can handle more connections than others.
- Weighted response time: Averages the response time of each server, and combines that with the number of connections each server has open to determine where to send traffic. By sending traffic to the servers with the quickest response time, the algorithm ensures faster service for users.
- Resource-based: Distributes load based on what resources each server has available at the time. Specialized software (called an "agent") running on each server measures that server's available CPU and memory, and the load balancer queries the agent before distributing traffic to that server.

Static:

- Round robin: Round robin load balancing distributes traffic to a list of servers in rotation using the Domain Name System (DNS). An authoritative nameserver will have a list of different A records for a domain and provides a different one in response to each DNS query.
- Weighted round robin: Allows an administrator to assign different weights to each server. Servers deemed able to handle more traffic will receive slightly more. Weighting can be configured within DNS records.
- IP hash: Combines incoming traffic's source and destination IP addresses and uses a mathematical function to convert it into a hash. Based on the hash, the connection is assigned to a specific server.

To keep things simple, we are going to simulate a **random load balancing** process without assigning any weights to the servers (in this case, the specifications of the servers are equal). This algorithm connects clients and servers at random, i.e. by the use of an underlying random number generator. When a load balancer receives a huge number of requests, a Random algorithm is capable of evenly distributing the requests to the nodes. Thus, similar to Round Robin, the Random algorithm is acceptable for clusters of nodes with similar configurations (CPU, RAM, etc).

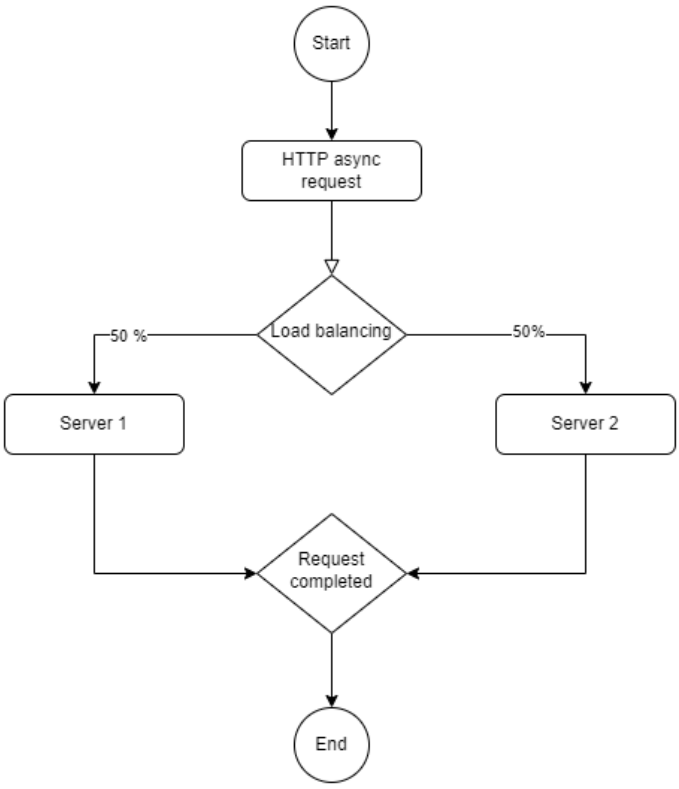


Figure 2: Process diagram

Main modules for system realization:

- New async request
- Request assignment to the server
- Request processing
- Request completion

Events for formal model specification:

- New event in the load balancer
- Request processing in server 1
- Request processing in server 2

And discrete variables for the queue.

X = $\emptyset$, y = $\emptyset$, E' = $\emptyset$

E" = $\{e_1'', e_2'', e_3''\}$

where:

- e_1'' - new request, processing time $\{\phi_i\}$
- e_2'' - processing on server 1, processing time $\{\phi_j\}$
- e_3'' - processing on server 2, processing time $\{\phi_k\}$

z(t_m) = {in(t_m), out(t_m), $Q_1(t_m)$, $Q_2(t_m)$, w(e_1'', t_m), w(e_2'', t_m), w(e_3'', t_m)}

Hypothesis

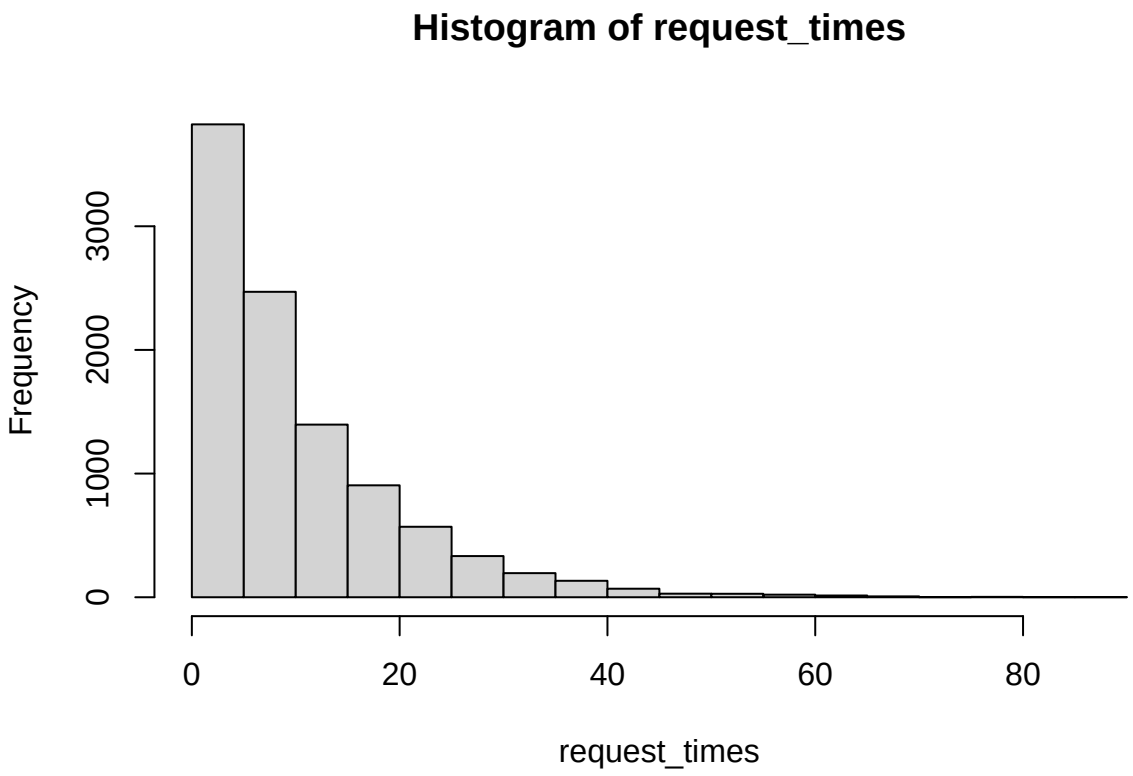
The random load balancing is ideally suited for servers that are comparable in their specifications and conduct the same work.

Simulation

For our simulation were are going to use simmer package. The simmer does process-oriented and trajectory-based Discrete Event Simulation (DES). It is intended to be a general yet robust framework. A distinguishing feature of simmer is that it makes use of the concept of trajectory: a shared path in the simulation model for entities of the same type. It is rather flexible and straightforward to use, and makes use of the magrittr package's chaining/piping approach.

Both the task time (timeout) and arrival time are generated using the rexp function in R: a random generator for the exponential distribution with a provided rate rate.

```
set.seed(404)
request_times <- rexp(10000, 0.1)
hist(request_times)
```



As illustrated in the process diagram above, the load balancer (server) contains two resources. Requests are routed to the server via the load balancer. While it makes sense to produce request time using an exponential distribution, we shall generate processing time using a normal distribution. Consider simulating a 40-request event and defining a pipeline. Each new request, job processing, and work completion are all logged. Given that the policy is written in C++ (as part of the simmer package), we will write our custom policy as a function that returns the server name.

```
library(simmer)
set.seed(404)
request <-
  trajectory("Request load balancing") %>%
  log_("New Request") %>%
  set_attribute("start_time", function() {now(server)}) %>%
  simmer::select(function() {
    paste0("server", sample(1:2, 1, replace=T,prob=c(0.5,0.5)))
    }) %>%
  # simmer::select(c("server1", "server2"), policy = "random") %>%
  seize_selected() %>%
  log_(function() {paste("Waited: ", now(server) - get_attribute(server, "start_time"))}) %>%
  timeout(function() {rnorm(1, mean=10, sd=5)}) %>%
  release_selected() %>%
  log_(function() {paste("Finished: ", now(server))})

request_times <- rexp(40, 0.1)
server <-
  simmer("server") %>%
  add_resource("server1", 1) %>%
```

```
  add_resource("server2", 1) %>%
  add_generator("Request", request, function() {c(0, request_times, -1)})
```

Let's run the simulation!

```
server %>% run()
```

```
## 0: Request0: New Request
## 0: Request0: Waited:  0
## 5.60856: Request1: New Request
## 6.68039: Request2: New Request
## 6.68039: Request2: Waited:  0
## 7.15635: Request3: New Request
## 9.68764: Request0: Finished:  9.68763819191766
## 9.68764: Request1: Waited:  4.07907937880767
## 10.7677: Request4: New Request
## 12.4871: Request1: Finished:  12.4870654724083
## 12.4871: Request3: Waited:  5.33071292727363
## 12.8412: Request2: Finished:  12.8411887396814
## 22.7229: Request5: New Request
## 22.7229: Request5: Waited:  0
## 28.8391: Request3: Finished:  28.8390932040345
## 28.8391: Request4: Waited:  18.0713948043119
## 32.2663: Request4: Finished:  32.2663265171882
## 32.7883: Request5: Finished:  32.7883081363955
## 33.3914: Request6: New Request
## 33.3914: Request6: Waited:  0
## 41.8949: Request6: Finished:  41.894945613313
## 42.5345: Request7: New Request
## 42.5345: Request7: Waited:  0
## 55.2762: Request7: Finished:  55.2761587338866
## 84.9859: Request8: New Request
## 84.9859: Request8: Waited:  0
## 86.5354: Request9: New Request
## 86.5354: Request9: Waited:  0
## 89.3375: Request10: New Request
## 94.5626: Request8: Finished:  94.5625530994197
## 99.0025: Request9: Finished:  99.0025383580774
## 99.0025: Request10: Waited:  9.66499530755473
## 102.715: Request11: New Request
## 103.991: Request10: Finished:  103.990935252682
## 103.991: Request11: Waited:  1.27640213946337
## 105.609: Request12: New Request
## 105.609: Request12: Waited:  0
## 111.19: Request13: New Request
## 111.659: Request12: Finished:  111.658940921054
## 111.659: Request13: Waited:  0.469304081006698
## 114.625: Request11: Finished:  114.625159716165
## 126.124: Request13: Finished:  126.123629920555
## 130.209: Request14: New Request
## 130.209: Request14: Waited:  0
## 137.302: Request15: New Request
## 144.618: Request16: New Request
## 147.836: Request14: Finished:  147.835550135257
```

```
## 147.836: Request15: Waited:  10.5335855679359
## 158.666: Request15: Finished:  158.666414842934
## 158.666: Request16: Waited:  14.0480191355829
## 159.215: Request16: Finished:  159.21501266686
## 161.472: Request17: New Request
## 161.472: Request17: Waited:  0
## 165.816: Request18: New Request
## 165.816: Request18: Waited:  0
## 167.116: Request19: New Request
## 173.747: Request20: New Request
## 176.122: Request18: Finished:  176.1215624491
## 176.122: Request19: Waited:  9.00576865762793
## 178.006: Request21: New Request
## 179.421: Request17: Finished:  179.421308829036
## 179.421: Request20: Waited:  5.67473439191772
## 179.635: Request22: New Request
## 182.159: Request23: New Request
## 183.655: Request19: Finished:  183.65459515094
## 183.655: Request21: Waited:  5.64876837594255
## 189.67: Request20: Finished:  189.670388060011
## 189.67: Request23: Waited:  7.51182116017418
## 190.57: Request23: Finished:  190.57023554788
## 192.928: Request21: Finished:  192.928063517738
## 192.928: Request22: Waited:  13.2932203834534
## 208.341: Request22: Finished:  208.340819548255
## 208.927: Request24: New Request
## 208.927: Request24: Waited:  0
## 219.618: Request24: Finished:  219.618340961256
## 224.098: Request25: New Request
## 224.098: Request25: Waited:  0
## 239.447: Request25: Finished:  239.447498882498
## 241.547: Request26: New Request
## 241.547: Request26: Waited:  0
## 246.695: Request27: New Request
## 253.615: Request26: Finished:  253.61529256187
## 253.615: Request27: Waited:  6.92054850289261
## 256.397: Request28: New Request
## 261.279: Request27: Finished:  261.278913803336
## 261.279: Request28: Waited:  4.88178178042608
## 262.703: Request29: New Request
## 263.581: Request30: New Request
## 273.026: Request31: New Request
## 273.122: Request32: New Request
## 273.122: Request32: Waited:  0
## 275.998: Request28: Finished:  275.998161617105
## 275.998: Request29: Waited:  13.2956431468469
## 276.594: Request33: New Request
## 284.324: Request29: Finished:  284.324437876148
## 284.324: Request30: Waited:  20.7431222402867
## 287.635: Request34: New Request
## 291.507: Request32: Finished:  291.50736758576
## 291.507: Request33: Waited:  14.9133868021935
## 300.435: Request30: Finished:  300.43541699706
## 300.435: Request31: Waited:  27.409536817954
```

```
## 306.485: Request33: Finished:  306.485208112616
## 312.905: Request31: Finished:  312.905463215672
## 312.905: Request34: Waited:  25.2704087456708
## 316.894: Request35: New Request
## 318.416: Request34: Finished:  318.415602696839
## 318.416: Request35: Waited:  1.52195602525933
## 325.948: Request35: Finished:  325.948397147869
## 334.017: Request36: New Request
## 334.017: Request36: Waited:  0
## 336.907: Request37: New Request
## 341.674: Request36: Finished:  341.673716632409
## 341.674: Request37: Waited:  4.76647204060225
## 351.83: Request37: Finished:  351.829778104392
## 373.229: Request38: New Request
## 373.229: Request38: Waited:  0
## 377.107: Request39: New Request
## 377.107: Request39: Waited:  0
## 382.942: Request38: Finished:  382.942153993619
## 385.104: Request39: Finished:  385.103645983866
## 385.265: Request40: New Request
## 385.265: Request40: Waited:  0
## 397.355: Request40: Finished:  397.354771507213
```

```
## simmer environment: server | now: 397.354771507213 | next:
## { Monitor: in memory }
## { Resource: server1 | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Resource: server2 | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Source: Request | monitored: 1 | n_generated: 41 }
```

Resource monitoring

Now we are going to generate a similar table as in the imitation example showed in class.

```
library(knitr)
library(tidyverse)
get_mon_arrivals(server) %>%
  mutate(service_start_time = end_time - activity_time) %>%
  arrange(start_time) %>%
  kable()
```

name	start_time	end_time	activity_time	finished	replication	service_start_time
Request0	0.000000	9.687638	9.6876382	TRUE	1	0.000000
Request1	5.608559	12.487065	2.7994273	TRUE	1	9.687638
Request2	6.680391	12.841189	6.1607976	TRUE	1	6.680391
Request3	7.156352	28.839093	16.3520277	TRUE	1	12.487065
Request4	10.767698	32.266326	3.4272333	TRUE	1	28.839093
Request5	22.722937	32.788308	10.0653716	TRUE	1	22.722937
Request6	33.391415	41.894946	8.5035309	TRUE	1	33.391415
Request7	42.534475	55.276159	12.7416842	TRUE	1	42.534475
Request8	84.985911	94.562553	9.5766421	TRUE	1	84.985911
Request9	86.535448	99.002538	12.4670905	TRUE	1	86.535448
Request10	89.337543	103.990935	4.9883969	TRUE	1	99.002538
Request11	102.714533	114.625160	10.6342245	TRUE	1	103.990935
Request12	105.608774	111.658941	6.0501664	TRUE	1	105.608774
Request13	111.189637	126.123630	14.4646890	TRUE	1	111.658941

name	start_time	end_time	activity_time	finished	replication	service_start_time
Request14	130.208776	147.835550	17.6267746	TRUE	1	130.208776
Request15	137.301965	158.666415	10.8308647	TRUE	1	147.835550
Request16	144.618396	159.215013	0.5485978	TRUE	1	158.666415
Request17	161.472203	179.421309	17.9491062	TRUE	1	161.472203
Request18	165.815733	176.121562	10.3058299	TRUE	1	165.815733
Request19	167.115794	183.654595	7.5330327	TRUE	1	176.121562
Request20	173.746574	189.670388	10.2490792	TRUE	1	179.421309
Request21	178.005827	192.928064	9.2734684	TRUE	1	183.654595
Request22	179.634843	208.340820	15.4127560	TRUE	1	192.928064
Request23	182.158567	190.570235	0.8998475	TRUE	1	189.670388
Request24	208.926629	219.618341	10.6917119	TRUE	1	208.926629
Request25	224.098423	239.447499	15.3490763	TRUE	1	224.098423
Request26	241.546589	253.615293	12.0687032	TRUE	1	241.546589
Request27	246.694744	261.278914	7.6636212	TRUE	1	253.615293
Request28	256.397132	275.998162	14.7192478	TRUE	1	261.278914
Request29	262.702518	284.324438	8.3262763	TRUE	1	275.998162
Request30	263.581316	300.435417	16.1109791	TRUE	1	284.324438
Request31	273.025880	312.905463	12.4700462	TRUE	1	300.435417
Request32	273.122069	291.507368	18.3852984	TRUE	1	273.122069
Request33	276.593981	306.485208	14.9778405	TRUE	1	291.507368
Request34	287.635055	318.415603	5.5101395	TRUE	1	312.905463
Request35	316.893647	325.948397	7.5327945	TRUE	1	318.415603
Request36	334.017376	341.673717	7.6563405	TRUE	1	334.017376
Request37	336.907245	351.829778	10.1560615	TRUE	1	341.673717
Request38	373.229109	382.942154	9.7130454	TRUE	1	373.229109
Request39	377.106735	385.103646	7.9969113	TRUE	1	377.106735
Request40	385.264987	397.354772	12.0897841	TRUE	1	385.264987

This could be presented for each server (resource):

```
get_mon_resources(server) %>%
select(-c(queue_size, limit, replication, capacity)) %>%
  kable()
```

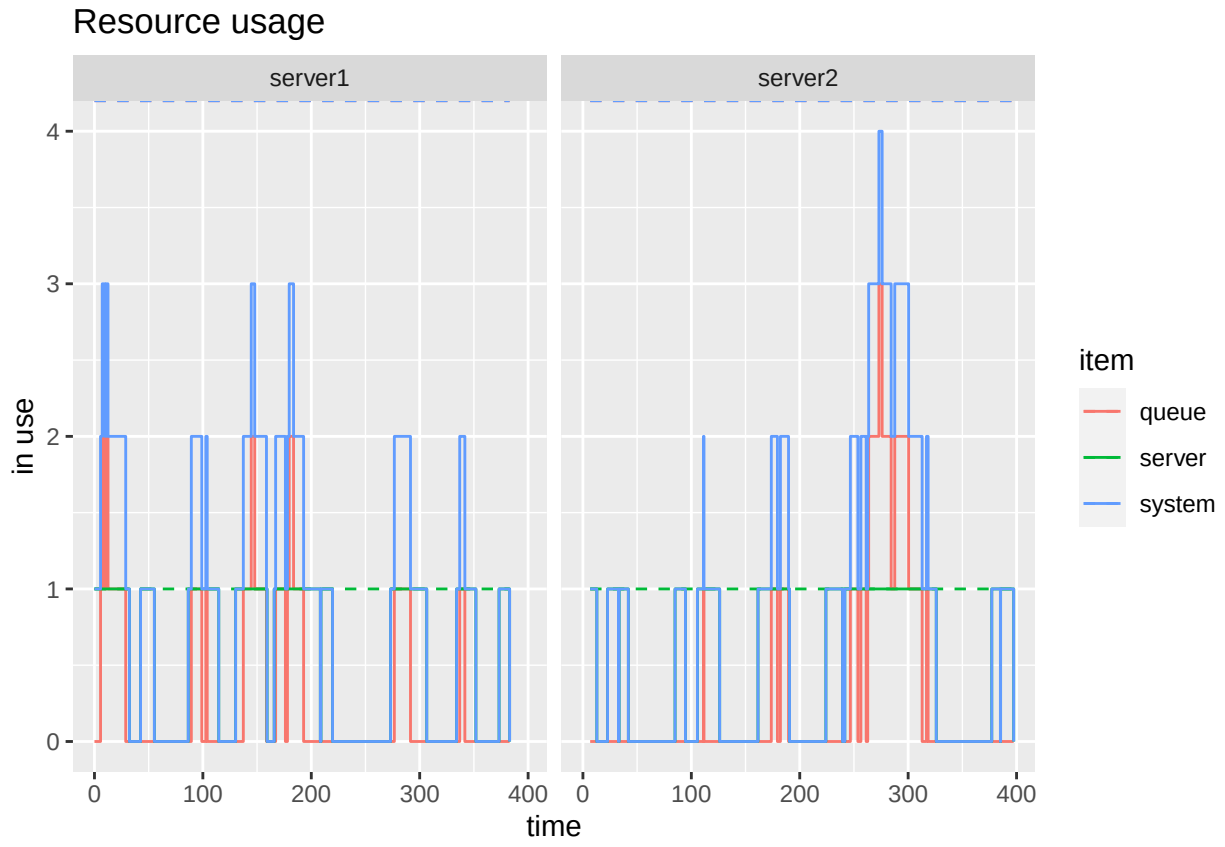
resource	time	server	queue	system
server1	0.000000	1	0	1
server1	5.608559	1	1	2
server2	6.680391	1	0	1
server1	7.156352	1	2	3
server1	9.687638	1	1	2
server1	10.767698	1	2	3
server1	12.487065	1	1	2
server2	12.841189	0	0	0
server2	22.722937	1	0	1
server1	28.839093	1	0	1
server1	32.266326	0	0	0
server2	32.788308	0	0	0
server2	33.391415	1	0	1
server2	41.894946	0	0	0
server1	42.534475	1	0	1
server1	55.276159	0	0	0

resource	time	server	queue	system
server2	84.985911	1	0	1
server1	86.535448	1	0	1
server1	89.337543	1	1	2
server2	94.562553	0	0	0
server1	99.002538	1	0	1
server1	102.714533	1	1	2
server1	103.990935	1	0	1
server2	105.608774	1	0	1
server2	111.189637	1	1	2
server2	111.658941	1	0	1
server1	114.625160	0	0	0
server2	126.123630	0	0	0
server1	130.208776	1	0	1
server1	137.301965	1	1	2
server1	144.618396	1	2	3
server1	147.835550	1	1	2
server1	158.666415	1	0	1
server1	159.215013	0	0	0
server2	161.472203	1	0	1
server1	165.815733	1	0	1
server1	167.115794	1	1	2
server2	173.746574	1	1	2
server1	176.121562	1	0	1
server1	178.005827	1	1	2
server2	179.421309	1	0	1
server1	179.634843	1	2	3
server2	182.158567	1	1	2
server1	183.654595	1	1	2
server2	189.670388	1	0	1
server2	190.570235	0	0	0
server1	192.928064	1	0	1
server1	208.340820	0	0	0
server1	208.926629	1	0	1
server1	219.618341	0	0	0
server2	224.098423	1	0	1
server2	239.447499	0	0	0
server2	241.546589	1	0	1
server2	246.694744	1	1	2
server2	253.615293	1	0	1
server2	256.397132	1	1	2
server2	261.278914	1	0	1
server2	262.702518	1	1	2
server2	263.581316	1	2	3
server2	273.025880	1	3	4
server1	273.122069	1	0	1
server2	275.998162	1	2	3
server1	276.593981	1	1	2
server2	284.324438	1	1	2
server2	287.635055	1	2	3
server1	291.507368	1	0	1
server2	300.435417	1	1	2
server1	306.485208	0	0	0

resource	time	server	queue	system
server2	312.905463	1	0	1
server2	316.893647	1	1	2
server2	318.415603	1	0	1
server2	325.948397	0	0	0
server1	334.017376	1	0	1
server1	336.907245	1	1	2
server1	341.673717	1	0	1
server1	351.829778	0	0	0
server1	373.229109	1	0	1
server2	377.106735	1	0	1
server1	382.942154	0	0	0
server2	385.103646	0	0	0
server2	385.264987	1	0	1
server2	397.354772	0	0	0

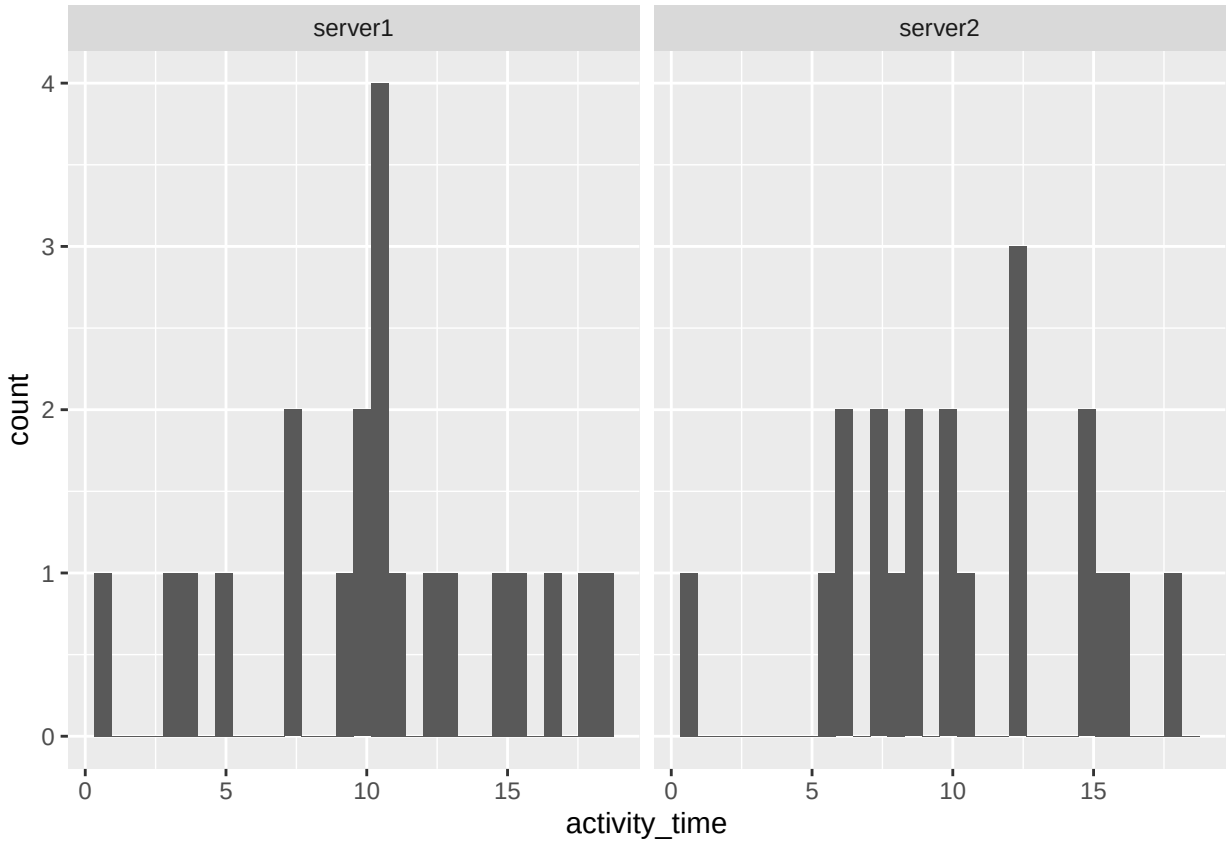
In a graphical depiction, each server's load may be clearly seen. This appears to be well-balanced, according to the graphical evidence.

```
library(simmer.plot)
plot(get_mon_resources(server), steps = TRUE)
```



It appears to be evenly distributed in terms of the length of each task.

```
get_mon_arrivals(server, per_resource = T) %>%
  ggplot() + geom_histogram(aes(x=activity_time)) + facet_wrap(~resource)
```



Results

Graphically, the random load balancing hypothesis is proven. It's a basic, but effective, approach of distributing workloads among multiple systems. Other sophisticated algorithms, on the other hand, can produce superior results in more complex settings (e.g. diverse system hardware, internet connectivity speed, etc.).

Full code can be found in GitHub repository.