An unpretentious introduction to Load Balancing

Matthias Beyer
Faculty of Computer Science
Hochschule Furtwangen University
78120 Furtwangen, Germany
Email: matthias.beyer@hs-furtwangen.de

***Abstract*—Cloud computing is one of the big challenges in the contemporary computer science world. On demand access to computing power and storage resources as offered by cloud computing providers result in new challenges in resource- and capacity management. Cloud hosting providers implement intelligent load-balancing algorithms to automatically distribute load over a large network of physical machines to serve demands in the best possible way and to guarantee a maximum of availability of resources.**

In this paper, we give a short introduction to load balancing in general and discuss different known load balancing algorithms. We compare their behaviour by applying each algorithm to a given problem. Additionally, we shortly introduce some lesser known and more complex algorithms.

Keywords: Algorithm, Load Balancing

1. Introduction

In large scale computing environments such as "the cloud", load balancing can be used to normalize workload of tasks and requests that are issued by clients over the network [1] or to distribute virtual machines over a set of physical machines. In [14], Pearce et al. state that an imbalance on a supercomputer environment can leave hundreds of thousands of processor cores in idle. This number will multiply in future with exascale machines. Pearce et al. show the importance of load balancing algorithms. With load balancing it is possible to distribute workload over a set of workers to either increase the overall capacity, which is called horizontal scaling, or improve resiliency [2], [3], [4].

The following chapters will describe several load balancing algorithms, outline their properties and show issues that emerge from the problem of load balancing in cloud and distributed systems.

2. Known Algorithms

The complexity of a Load Balancing Algorithm (LBA) increases with the complexity of the system it has to work with. One has to consider a load balancing algorithm that computes the target worker for a task, where the size of the tasks are equal, the workers have the same capabilities and the number of overall tasks is known. Such an algorithm is not of complexity. In fact, it can be implemented by a simple mathematical equation:

$$i = j \mod |W| \tag{1}$$

Where i is the number of the worker instance in set W that will be assigned with the task number j. Though, such an environment is unlikely.

In contemporary systems, the size of a task can often not be predicted. Algorithms like "min-min" and "max-min" [5] are not applicable to most systems, because with these algorithms the complete set of tasks must be known prior to scheduling them. In most environments, this is not possible.

The capabilities of a worker instance may be known, but as tasks come in and are assigned to workers, an algorithm has to implement some book-keeping to know which worker has the least load at a given point in time. This is of moderate complexity, though having only one load balancer in a large ecosystem is highly unlikely, as it would be a single point of failure. Instead, multiple load balancers are used. Because of this, some synchronization method between the book-keeping of each load balancer must be introduced to ensure the integrity of the load data on each load balancer node.

Measuring load is often not easy to accomplish or even impossible, but load information is cirtical in the load balancing environment [15].

If worker instance W_1 has the least load at time t_1, the load balancer L_1 might assign a number of tasks to it in t_2. After that, L_1 must notify L_2 about the new load on W_1, which it does in t_3. In the meantime, load balancer L_2 might need to assign another set of tasks, which it assignes to W_1 in t_2, because it does not know yet that W_1 isn't the least loaded worker anymore. This causes W_1 to become overloaded in t_3.

2.1. Random load balancing

A cheap load balancing strategy is randomizing the target.

$$i = rand() \mod |W| \tag{2}$$

Which has several advantages over other approaches:

- Simplicity
- Few edge cases
- Easy failover
- Works in distributed environments

These properties are rather easy to replicate with non-random load balancing algorithms, except from the last one.

In a common load balancing environment, having only one load balancer distributing tasks between worker instances is naive. Commonly, at least one fail-over is deployed, sometimes several. Also, in a large system, load balancers might be spread not only over server racks but rather over continents. Making load balancers work with each other and synchronize state information introduces high complexity.

Nevertheless, distributing load randomly does not guarantee the best distribution over a given set of notes, neither with one load balancer nor with many. It is even possible that a small set of workers are assigned with a huge number of tasks while other workers only handle a very small number of tasks, resulting in a high load on some machines while others idle.

The desireable state is that each machine has the same workload, resulting in a perfect distribution of load over all machines. With an algorithm randomly assigning work to workers, the possibility exists that the system ends up in an unbalanced state. Though this is very unlikely, we can do better.

2.2. Roundrobin (RR)

Another cheap load balancing strategy might be RR.

$$i = number\_of(T) \mod |W| \tag{3}$$

As RR is a static algorithm [6], it yields, like randomized load balancing, problems in distributed environments, where not one but many load balancers are applied to a network of servers.

Also, as uniform tasks only occur in a perfect world, RR load balancing might result in overloaded servers as well.

2.3. Join Shortest Queue (JSQ)

A known load balancing algorithm is JSQ.

With this algorithm, the load balancing mechanism implements some book-keeping mechanism on how many jobs are in the queues of the worker instances and it selects the instance with the shortest queue for a new task.

If this algorithm is applied in a non-distributed environment, it behaves exactly like RR 2.2. Applying it with several load-balancers, though, might yield some better results, though synchronization must be implemented carefully. As synchronization is rather expensive, one might implement a scheduled synchronization which causes the instances to synchronize state information in a periodic manner. However, this can result in a "herd effect" in distributed environments, where a subset of machines is either underutilized or overloaded with tasks. This is caused by the old information of how the nodes are utilized as described in. Mechanisms exists to use this old information to estimate the current situation and improve the decisions of the load balancing [7]. The LI framework as introduced in [7] improves load distribution if the information about the utilization of the nodes is old and therefor prevents the "herd effect" mentioned earlier.

As synchronization might be to complex and faulty to implement, querying the servers for their queue length might be an alternative. To omit synchronization, and therefor blocking, one could introduce a querying-mechanism. However, querying a server instance for its queue length introduces complexity to the servers and might also be a non-trivial task.

Either way, JSQ is a valuable option for load balancing in environments where the request is short-living, like in Domain Naming System (DNS)-environments or web servers.

2.4. Random Select Join Shortest Queue (RSJSQ)

A surprisingly good optimization is the combination of the random load balancing implementation with JSQ.

$$\begin{aligned} x &= rand() \mod |I| \\ y &= rand() \mod |I| \\ i &= \begin{cases} x & \text{if } load(W_x) =< load(W_y) \\ y & \text{if } load(W_x) > load(W_y) \end{cases} \end{aligned} \tag{4}$$

Due to the fact that querying each server for its queue length might be a non-trivial and expensive task, selecting two servers randomly (or by a hash function applied to the task, as stated in [9]) and applying the JSQ algorithm on this subset of servers yields, obviously, better performance as the number of queries for the queue length is minimized.

As stated in [9], RSJSQ might result in $\Theta(log(log(n)))$ tasks on each server, rather than $\Theta(log(n)/log(log(n)))$, where n is the number of servers in the environment.

2.5. Comparing RR, JSQ and RSJSQ

For the following comparison of the algorthms RR, JSQ and RSJSQ, they were implemented in the Rust programming languages. Sample data in form of a CSV file was generated with the code from Fig. 2.

The sample data holds two values in each column: A task name and a time factor as shown in Fig 1. The name is for simple debugging purposes, the latter is an artificial random time factor. It is a number between 1 and 100 and specifies how many *ticks* the task takes to be computed. The algorithms put one task into the workers on each tick.

```
Task 0,55
Task 1,53
Task 2,43
Task 3,19
...
```

Figure 1. Sample data

In an environment with one load balancer where tasks can be distributed before the processing begins, JSQ and RSJSQ behave exactly like RR as shown in Fig. 3 and Fig. 4. This is not a surprise for JSQ (with manual book keeping),

```
(0...50_000).each do |n|
  r = Random::rand(100) + 1
  puts "Task #{n},#{r}"
end
```

Figure 2. Ruby script to generate random task data

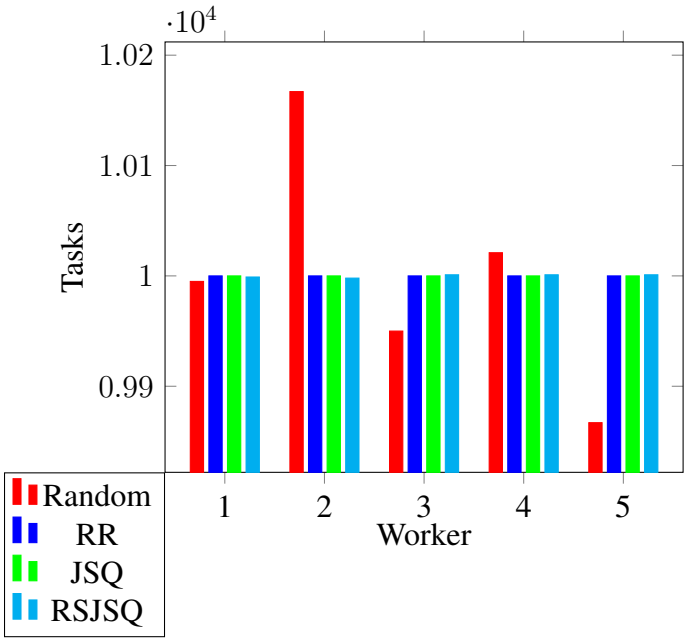

Figure 3. Task distribution

as the node with the least tasks after assigning a task to node n is always the next node: $(n+1) \mod |N|$. Surprisingly, RSJSQ behaves *almost* like RR and JSQ, despite the random selection of workers (compare Fig. 5 and Fig. 6).

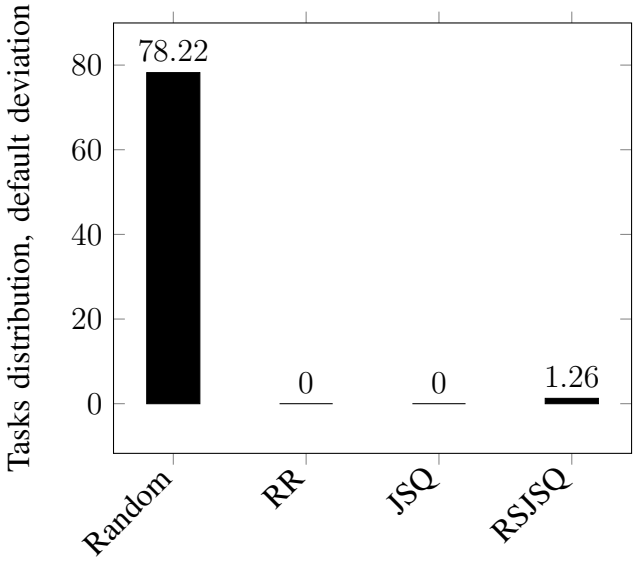

Figure 4. Task distribution, default deviation

```
for t in tasks.into_iter() {
    for mut a in ass.iter_mut()
    {
        a.tick();
    }
    ass.sort_by(|a, b| {
        a.len().cmp(&b.len())
    });
    ass[0].push(t);
}
```

Figure 5. Implementation of JSQ, with time-component

If we take the processing of the tasks on the workers into account, we clearly see the improvement of JSQ over RR. Each tick one of the tasks inside a worker gets computed, which means that the time factor gets decremented by 1. If a task hits time factor 0, it gets removed from the tasks queue in the worker and the next task will be computed on the next tick. After 50.000 tasks are pushed into the workers, the current state is printed out

```
let mut src = random::default()
    .seed([42, 1337]);
for task in tasks.into_iter() {
    for mut a in ass.iter_mut() {
        a.tick();
    }

    let i = src.read::<usize>()
        % nass;
    let j = src.read::<usize>()
        % nass;

    let il = ass[i].len();
    let jl = ass[j].len();
    let x = if il <= jl { i }
            else { j };

    ass[x].push(task);
}
```

Figure 6. Implementation of RSJSQ, with time-component

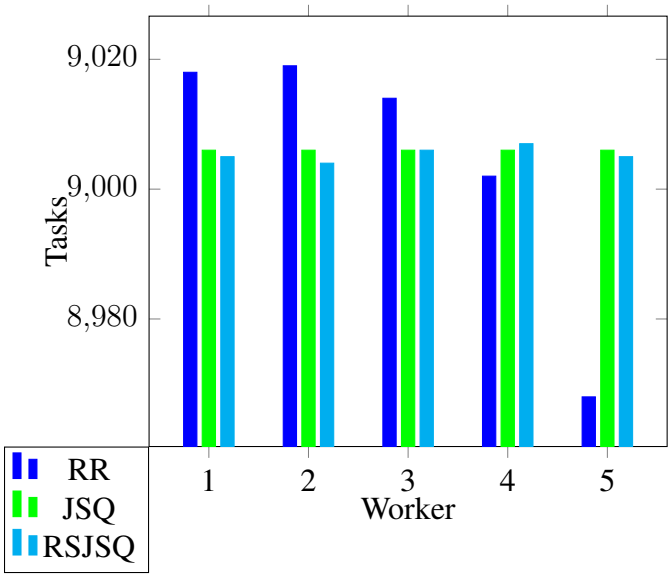

Figure 7. Task distribution

by the implementation (visualized in Fig. 7 and Fig. 8).

As each tick one task gets pushed into the set of workers, we get a constant rate of new tasks. This might not be a real-world scenario, though it serves well enough to illustrate the behaviour of the algorithms.

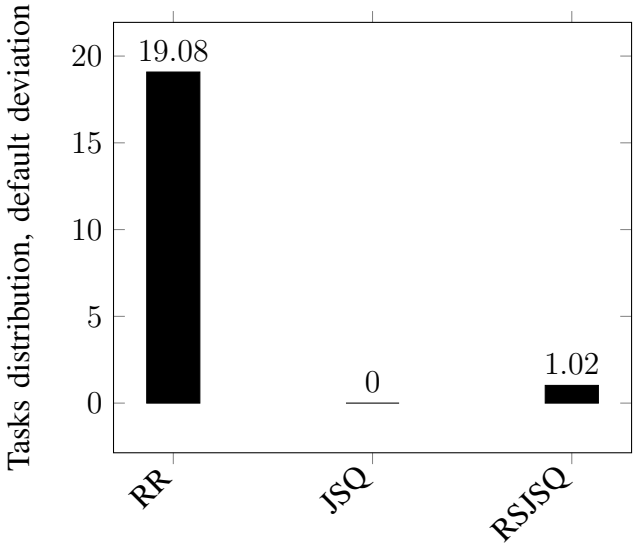

Figure 8. Task distribution, default deviation

2.6. Hash based load balancing

Another way of distributing load is using a hash algorithm to determine the target server. With this approach, each request gets hashed with a hash function. The result of the hash function can be seen as number. Taking this number modulo the number of servers yields the index of the target server. In his scenario, the number of servers has to be constant. If a new server is added to the cluster, the modulo operation yields other servers for the same requests as before, yielding cache effects useless [10]. Another implication of hash based load balancing is that consistent hashing is less than ideal for load balancing as requests are not distributed evenly over content. One could consider a storage-cluster for videos, where each server has a subset of the complete data. Naturally, cat videos are more likely to be requested by the user than other videos, resulting in higher load on the servers holding this particular data.

In [10] it is stated, that as RSJSQ was not suiteable for their present problem. Although, Consistent Hashing with Bounded Loads (CHBL) as proposed by [11] was.

CHBL provides some guarantees which are fortunate for dynamic hash based load balancing in distributed environments. That is, mostly, an upper bound for the load of each server [11] and some more cache-friendly properties [10]. With CHBL, distribution of requests is the same as with consistent hashing.

The CHBL algorithm was implemented and submitted to the HAProxy HTTP load balancer open source project and on November 25, HAProxy 1.7.0 was published with CHBL available. Real world measurements reveal a great benefit from applying CHBL, as shown in [10].

3. Alternative Algorithms

In [12], Dhinesh Babu L.D. and P. Venkata Krishna propose a honeybee behaviour inspired load balancing algorithm. This algorithm takes the priority of tasks into account and works well for

heterogeneous cloud environments. It also reduces the waiting time of tasks in the queues on the worker instances.

The honeybee inspired load balancing algorithm is, as the name tells, based on the foraging behaviour of honey bees and the Artificial Bee Colony Algorithm (ABC) as proposed by N. Karaboga [12].

Dhinesh Babu L.D. and P. Venkata Krishna illustrate that their algorithm improves average execution time and reduce waiting time of queued tasks.

A. Nakai, E. Madeira and L. E. Buzato outline a server-based load balancing policy for world-wide distributed web servers in [13].

Compared to known solutions (RR, Roundrobin with Asynchronous Alarm (RR-AA), Smallest Latency (SL) and Least Used (LU) as implemented in the "HAProxy" open source load balancing solution), their algorithm presented better performance due to the fact that it prevented overloading rather than rebalancing work if overloading occurs.

In [16] M. E. Soklic compares a diffusive load balancing algorithm to common algorithms such as RR and JSQ in a distributed environment. It is shown that diffusive load balancing is more effective than RR or static load balancing.

[4] lists more load balancing techniques, featuring but not limited to

- *Decentralized content aware load balancing*
- *Load Balancing strategy for virtual storage*
- *Load Balancing mechanism based on ant colony and complex network theory*
- *Two-Phased Scheduling*
- ...

4. Related Work

Previous work compared distributed load balancing algorithms for cloud computing environments. Honeybee Foraging Behaviour as well as Biased Random Sampling and Active Clustering were compared by M. Randles et al. in [17]. It is shown that current commercial centralized offerings won't be as scalable as demand.

H. Bryhni et al. compared load balancing techniques in a web environment. They explicitely compare the rotating name-server method to other methods used in web cluster environments, such as remapping requests and responses in the network [18]. They see remapping as viable solution for load balancing. They also argue that load balancing has some positive side effects regarding to fault tolerance. Finally, they conclude that a RR algorithm is feasible for its simple implementation and good load sharing.

5. Conclusion

In this paper, we described some of the known load balancing algorithms but also listed some of the lesser known. We showed a simple comparison of known load balancing algorithms and their behaviour in a non-distributed environment by implementing them in the Rust programming language and feeding them with artificial random data.

Though load balancing has been adressed by various parties, the NP-hard problem of distributing load among a set of worker instances remains interesting.

Acknowledgments

The author would like to thank Tyler McMullen for his talk "Load Balancing is Impossible" at MeshCode 2016, which was a huge inspiration for this paper.

References

[1] Alakeel and Ali M, *A guide to dynamic load balancing in distributed computer systems*, International Journal of Computer Science and Information Security, Vol. 10, No. 6, June 2010, pages 153-160.

[2] Thomas A. Limoncelli, *Are You Load Balancing Wrong?*, ACM Queue, Vol. 14, No. 6, December 2016, pages 10:5-10:13.

[3] Rimal, Bhaskar Prasad and Choi, Eunmi and Lumb, Ian, *A taxonomy and survey of cloud computing systems*, INC, IMS and IDC 2009, pp. 44-51.

[4] Kansal, Nidhi Jain and Chana, Inderveer, *Cloud load balancing techniques: A step towards green computing*, IJCSI International Journal of Computer Science Issues, Vol. 9, No. 1, 2012, pages 238-246.

[5] P.P. Geethu Gopinath and Shriram K. Vasudevan, *An In-depth Analysis and Study of Load Balancing Techniques in the Cloud Computing Environment*, Procedia Computer Science, Vol. 50, 2015, pp. 427-432.

[6] Chaczko, Zenon and Mahadevan, Venkatesh and Aslanzadeh, Shahrzad and Mcdermid, Christopher, *Availability and load balancing in cloud computing*, International Conference on Computer and Software Modeling, Singapore, Vol. 14, 2011.

[7] Michael Dahlin, *Interpreting stale load information*, IEEE Transactions on parallel and distributed systems, Vol. 11, No. 10, 2000, pages 1033-1047.

[8] M. Mitzenmacher, *The power of two choices in randomized load balancing*, IEEE Transactions on Parallel and Distributed Systems, Vol. 12, No. 10, October 2001, pages 1094-1104.

[9] M. Mitzenmacher, A. Richa, and R. Sitaraman, *The Power of Two Random Choices: A Survey of Techniques and Results*, Handbook of Randomized Computing, Vol. 1, October 2001, pp. 255-312.

[10] Vimeo Engineering Blog, *Improving load balancing with a new consistent-hashing algorithm*, https://medium.com/vimeo-engineering-blog/ December 2016.

[11] Vahab S. Mirrokni, Mikkel Thorup and Morteza Zadimoghaddam, *Consistent Hashing with Bounded Loads*, CoRR, Vol. abs/1608.01350, September 2016.

[12] Dhinesh Babu L.D. and P. Venkata Krishna *Honey bee behavior inspired load balancing of tasks in cloud computing environments* , Applied Soft Computing, Vol. 13, No. 13, 2013, pp. 2292-2303.

[13] A. Nakai, E. Madeira and L. E. Buzato, *On the Use of Resource Reservation for Web Services Load Balancing*, Journal of Network and Systems Management, Vol. 23, No. 3, 2015, pp. 502-538.

[14] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato, *Quantifying the Effectiveness of Load Balance Algorithms*, Proceedings of the 26th ACM International Conference on Supercomputing, Series ICS '12, 2012, pp. 185-194.

[15] K. A. Huck and J. Labarta, *Detailed Load Balance Analysis of Large Scale Parallel Applications*, 39th International Conference on Parallel Processing, September 2010, pp. 535-544.

[16] M. E. Soklic, *Simulation of Load Balancing Algorithms: A Comparative Study*, SIGCSE Bull., Vol. 4, No. 34, 2002, pp. 138-141.

[17] M. Randles, D. Lamb and A. Taleb-Bendiab, *A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing*, IEEE 24th International Conference on Advanced Information Networking and Applications Workshops, 2010, pp. 551-556.

[18] H. Bryhni and E. Klovning and O. Kure, *A comparison of load balancing techniques for scalable Web servers*, IEEE Network, Vol. 14, No. 4, 2000, pp. 58-64.