Dynamic load balancing in distributed systems

Mihai Bojescu
Masters in Artificial Intelligence and Optimisation
Faculty of Computer Science of University "Alexandru Ioan Cuza" of Iași
Iași, Romania
bojescu.mihai@gmail.com

***Abstract*—This document details what is load balancing, what is the dynamic load balancing strategy in distributed systems, possible architectures and challenges, case studies on Kubernetes and AWS ELB, and an experimental design created as a demonstration for the previously mentioned work.**

***Index Terms*—Load balancing, distributed systems, architectures, demonstration**

I. Introduction

In distributed systems, load balancing is the act of distributing the workload across multiple nodes. Load balancing is performed in order to make greater use of the available resources (such as I/O devices, memory, CPU, data), increase efficiency of resources, lower response times, or to lower the costs of idle resources. Compared to simple load sharing, load balancing aims to roughly evenly distribute the workload on the available nodes, which is a desired global optima of the system.

Effective load balancing comes with many challenges, such as load estimation, load level comparison, system stability, efficient task transfer, efficient messaging between nodes, peer selection, and more.

Load balancing can be of two distinct types, based on the knowledge used to perform the balancing:

1) Static load balancing
2) Dynamic load balancing

Generally, load balancing is performed at 2 OSI layer levels: OSI layer level 4 and OSI layer level 7.

In the sections below, both static and dynamic load balancing strategy will be detailed.

II. Static load balancing

In static load balancing, prior knowledge about the compute nodes is required. Static load balancing is generally used in systems where the task size is known and is predictable. In such scenario, the topology of the system is well-known ahead of time, and ahead of the task execution. Tasks assigned to one node cannot be transferred to another one. In general, static load balancing is rigid, but less complex to implement.

A. Architectures and topologies

Static load balancing mainly follows 2 architectures:

1) "Explicit" balancing, using a load balancer in front of nodes [1]
2) "Implicit" balancing, using a DNS and exposed nodes [3]

The "explicit" balancing architecture aims to distribute traffic to nodes in a more controlled manner, giving control to the application to perform this action.

This presents multiple advantages:

1) Extracting the responsibility of traffic filtering away from the targeted application nodes
2) Implementation is more accessible
3) Multiple plug-and-play solutions already exist, such as NGINX, Apache, HAproxy

And some disadvantages:

1) When the load balancer fails, the users don't have access to the application
2) When the load balancer gets overloaded, the users will experience high latency
3) The load balancer is an additional resource that needs to be managed
4) The overhead is higher, as the load balancer might not be next to the application server

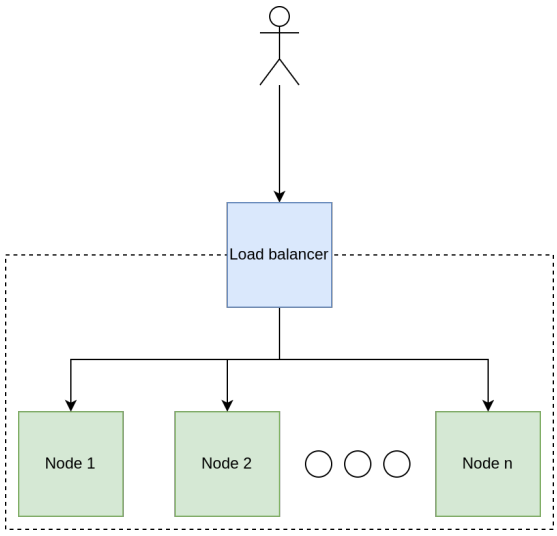

Fig. 1: Explicit load balancing using a load balancer

The load balancer can also be swapped for a message broker, achieving the same effect. The architecture is more complex, now relying on both a gateway and the message broker, but this allows for better availability.

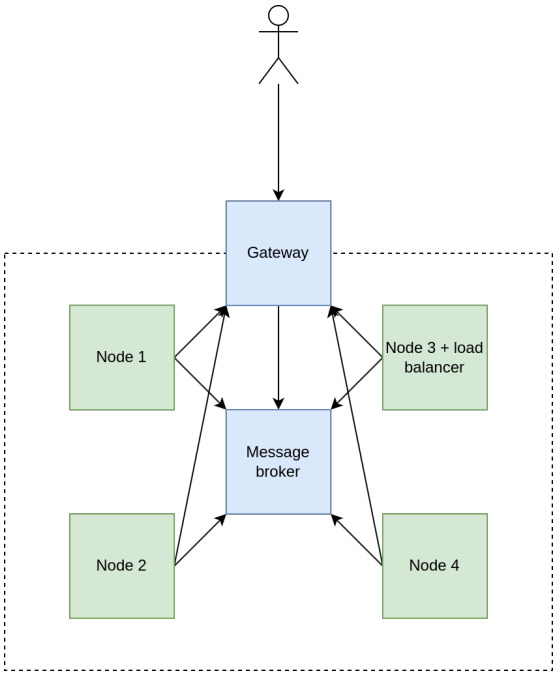

Fig. 2: Explicit load balancing using a message broker

The "implicit" balancing architecture aims to pass the load balancing task from a load balancer to a DNS server, which will rotate addresses, in hopes of achieving even task distribution. This performs implicit load balancing, as the client can decide wether to use the provided host, or to use another one from the DNS server response, assuming the DNS server offers more than one response.

This presents multiple advantages:

1) The design is even simpler than figure 1, as no load balancer needs to be managed and maintained
2) It can be simpler to implement than figure 1 and figure 2
3) The DNS control *can be* outside of the application context (public DNS, for instance), which can reduce costs

Along with the disadvantages from figure 1, the architecture also has the following disadvantages:

1) In instances where the DNS is not owned by the application developer, the owner of the DNS server can perform modifications without the developer's knowledge, leading to issues
2) The users might not know what DNS server to use to access the application

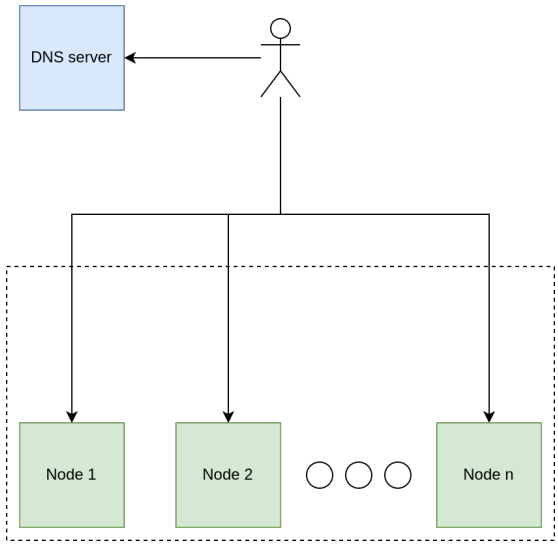

Fig. 3: Implicit load balancing using DNS

B. Algorithms

In static load balancing, the following strategies are used:

1) Round-Robin method aims to evenly distribute tasks to each server, without taking in consideration the load of the server
2) Random choice method aims to distribute tasks to different nodes, at random
3) Hashing method aims to hash the url, and pass the request to a node that matches the hash
4) Least connection method aims to send the request to the least full node that the node knows about, without taking in consideration the load of the server
5) Weight distributed method aims to send the requests to based to a prior-set weighted node selection.

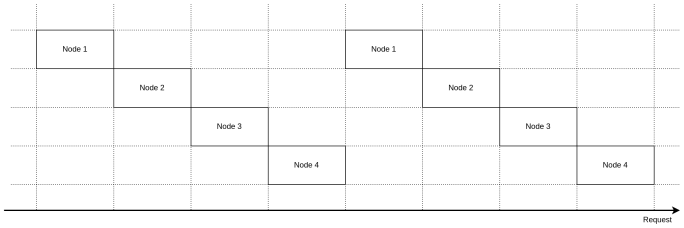

Fig. 4: Round-Robin strategy

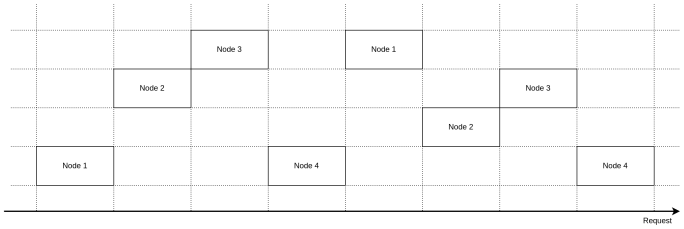

Fig. 5: Random strategy

III. Dynamic load balancing

In dynamic load balancing, no knowledge of the system and tasks is required or assumed a priori. Tasks are dynamically assigned, reassigned and distributed to the available nodes in the system without the prior input from the application developer. The strategy is highly flexible and provides high availability, with generally more performance compared to the static load balancing strategy. While the strategy has many advantages, it requires more resources, is prone to high chattiness, and is more complex overall.

Various studies on dynamic load balancing, such as [2], show that such strategy can increase the resource utilisation, lower response times, increase availability and reduce costs.

Dynamic load balancing algorithms are varied, and an effective one must trade between the disadvantages listed above in order to achieve good performance of the system. They range from simple comparisons to advanced statistical analysis to achieve the desired system. Some well-known algorithms will be listed in the sections below, along with their use-cases.

A. Architectures and topologies

Dynamic load balancing architectures don't overlap with static load balancing architectures in full, as the "implicitly" balanced architectures are not supported.

1) For the static "explicitly" balanced architecture, the load balancer must take into account the load of each node before assigning tasks
2) For the static "implicitly" balanced architecture, the DNS server must account for individual node load. This effectively makes the architecture an "explicitly" balanced one.

Along the two aforementioned architectures, the strategy allows for a decentralised one, supported by the task transfer option of the dynamic load balancing strategy and by individual node load balancing. The architecture allows nodes to be independently scaled, thus achieving higher resource utilisation efficiency.

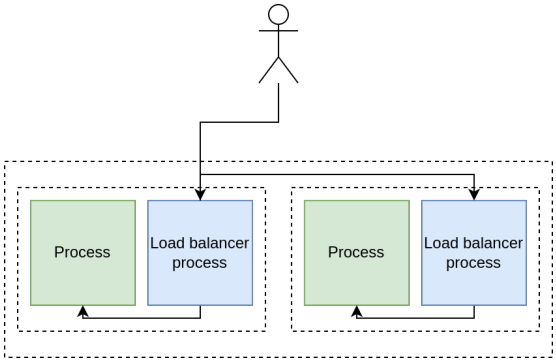

Fig. 6: Internal representation of decentralised, dynamic load balanced nodes

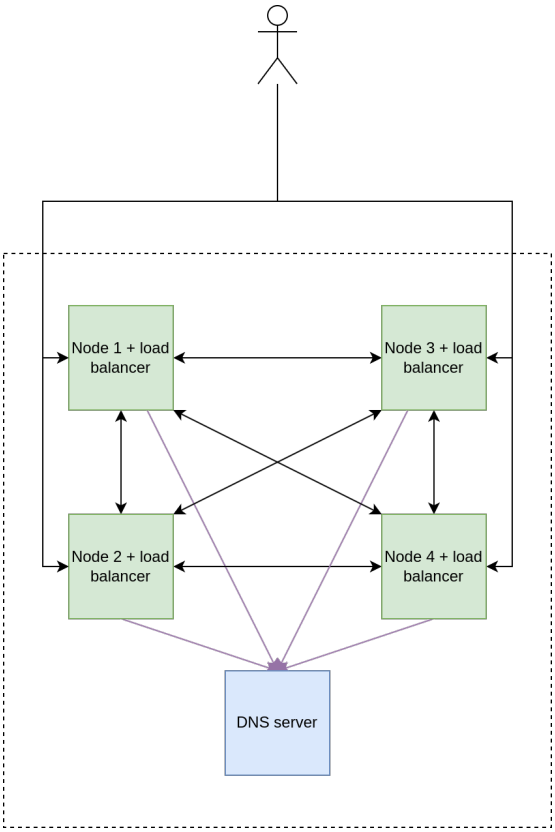

Fig. 7: Decentralised, dynamically load-balanced system, using internally, dynamic load balanced nodes

B. Algorithms

Dynamic load balancing are rather diverse, can take multiple parameters such as CPU load average, used memory, tasks in queue and response times.

Many algorithms are based on one of the following ones:

1) Least compute-loaded method
2) Least response time method
3) Least bandwidth used method
4) Least packets used method

The algorithms try to achieve the same global optima in different ways, and should be used depending on the specific use-cases, some of them being:

1) Least compute-loaded method should be used for tasks such as video transcoding, AI workloads, CI/CD operations
2) Least response time method should be used for tasks such as video and audio calls, stock markets, online games
3) Least bandwidth used method should be used for tasks such as video streaming, cloud storage, CDNs
4) Least packets used method should be used for tasks such as chat applications, external load balancers for DDoS mitigation

Each algorithm uses metrics from the available nodes to perform load balancing effectively. Depending on the system, the metrics can be pushed to the balancer, or pulled by it.

Systems that use the push method tend to be most chatty and have the least latency, while the pulling ones have ones are the polar opposite.

IV. Case studies

A. Case studies: Kubernetes

Kubernetes is a well-known open-source platform for building highly-available, highly-scalable and platform-agnostic web applications. Pods can communicate between each other using the overlay network, without the need for NAT and without regard of running on different nodes. The overlay network provides a flat IP address space, which eases communication.

Kubernetes supports both static and dynamic load balancing across the nodes, depending on the given configuration. By default, static load balancing is used.

Architecturally, Kubernetes works similarly to figure 1, where a load balancer performs traffic management. The load balancer (ClusterIP, NodePort, LoadBalancer) is individually-scalable in order to provide better uptime.

Kubernetes load balancers can perform the balancing at both OSI layer 4 and OSI layer 7, depending on the configuration.

An example of load-balanced system configuration for Kubernetes is the following:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
name: calendar-ingress
spec:
tls:
- hosts:
   - calendar.example.com
   secretName: calendar-secret
rules:
- host: calendar.example.com
   http:
   paths:
   - path: /api/events
      backend:
      serviceName: events-svc
      servicePort: 80
   - path: /api/calendars
      backend:
      serviceName: calendars-svc
      servicePort: 80
```

Listing 1: Code example for a Kubernetes load balanced system

The resiliency of the system highly depends on the configuration of the cluster.

B. Case study: AWS ELB

AWS ELB is a load balancer often used when building projects using the Amazon Web Services Cloud platform. The system follows an architecture similar to figure 1. Similarly to Kubernetes, the system is individually scalable and the scaling is handled implicitly by AWS unlike Kubernetes which perform scaling explicitly using configuration files.

The solution is paid and closed-source, thus it does not permit auditing. As per AWS, AWS ELB can perform load balancing both at OSI layer level 4 and OSI layer 7. The solution is also very well integrated with existing AWS products, such as EC2, EKS and ECS.

Performance auditing can be performed using "siege" or Apache "ab". Since the system is scalable, it is very resilient to large number of requests.

V. Experimental study: Building a decentralised, dynamically load balanced system

In this section, an experimental study will be detailed on building a decentralised, dynamically load balanced system from start to finish. The targeted use-cases for such systems are CI/CD systems and batch processing workers, which need to balance their compute evenly in order to achieve optimal system efficiency. The system can be used on other types of workloads while still offering good efficiency, but might not be the best choice.

The system includes the following components:

1) An OSI layer 7 load balancer
2) A frontend application used to view the metrics of the node, which are used by the load balancer to perform the task placement

The load balancer process polls a DNS server for peer nodes each second, which will be later used for passing tasks when the node is compute-overloaded.

For the balancing algorithm, a "least compute-loaded method" was applied. When a HTTP request is received by the balancer, it queries at most n nodes (configurable) for their metrics, and picks which has the most compute power available.

Architecturally, the system is the same as figure 7, where each node in the system follows figure 6 for its components. The system was initially thought to work for containerised applications using Docker, but can be applied to systems that don't have such prerequisite available.

A. Experiments

The testing methodology was the following:

1) Apache "ab" CLI - a well-known HTTP API stress testing tool - was used
2) A total of 20000 requests were sent in a session
3) A total of 1000 requests were sent concurrently
4) Tests were repeated 5 times

When tested against a homogenous compute environment with two virtual machines, with each machine:

1) Having 2vCPUs and 1GB of RAM
2) Running an application and a load balancer in Docker

The results were:

Metric	Min	Mean	Max
Latency	33ms	1131ms	2381ms
CPU load average	0.98	1.54	2.01
Used memory (load balancer)	109MB	120MB	132MB

TABLE I: Homogenous, dynamically load balanced system metrics for experimental study

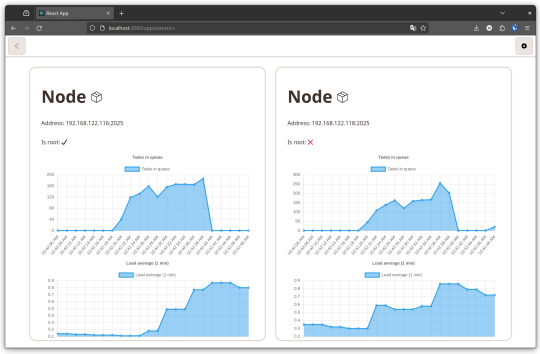

Fig. 8: Homogenous, dynamically load balanced system charts

When using three heterogenous nodes - a laptop with an Intel Core i7 1260p processor, a BCM2711 SoC SBC, and a smartphone with a Qualcomm Snapdragon 8 gen 3 processor, all running under a WireGuard VPN - the results were:

Metric	Min	Mean	Max
Latency	129ms	582ms	3527ms
CPU load average	1.43	1.86	2.71
Used memory (load balancer)	176MB	179MB	182MB

TABLE II: Heterogenous, dynamically load balanced system metrics for the experimental study

B. Future work

The experimental study system presents some issue regarding connections being reset, issue which needs revised. Another area which could use improvement is using push metrics instead of pull metrics for the load balancer, which can be used to further improve efficiency and reduce system chattiness.

Other areas of improvements include switching from Node.JS with Typescript to a statically compiled, high-performance language such as C++ or Rust, improving logging, and adding support for more workload types.

VI. Conclusions

Dynamically load balanced systems aim to improve the performance and efficiency of distributed systems. Building effective algorithms is not trivial and requires efficient communication between nodes for the system to work as intended.

Architecturally, dynamic load balanced systems allow for more options than their static load balanced counterparts, which is an added benefit.

Since nodes can know the state of other nodes in the system, they can perform task transfers to make utilise the resources more efficiently. Using this, a more decentralised system architecture can be achieved, which is more resilient to downtime. This paper includes such a system built from scratch as an experimental study on this topic.

References

[1] Ali M. Alakeel. “Guide to Dynamic Load Balancing in Distributed Computer Systems”. IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.6, June 2010

[2] Z. Khan, R. Singh, J. Alam, and R. Kumar. “Performance Analysis of Dynamic Load Balancing Techniques for Parallel and Distributed Systems”. International Journal of Computer and Network Security, vol. 2, no. 2, February 2010.

[3] V. Cardellin, M. Colajanni, P. Yu. “Dynamic Load Balancing On Web-Server Systems”. IEEE Internet Computing, June 1999.

[4] Karen D. Devine a, Erik G. Boman, Robert T. Heaphy , Bruce A. Hendrickson, James D. Teresco,1 , Jamal Faik, Joseph E. Flaherty, Luis G. Gervasio. “New challenges in dynamic load balancing”. Elsevier Applied Numerical Mathematics 52 (2005) 133-152, October 2004.

[5] https://www.nginx.com/blog/nginx-plus-ingress-controller-kubernetes-load-balancing/

[6] https://docs.aws.amazon.com/elasticloadbalancing/latest/network/introduction.html

[7] https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html