Load Balancing
Group 7

Ashish Sharma 13CS30043
Apurv Jain 17CS60R78
Chanderki Rakesh Kumar 17CS60R71

1 Introduction

Load Balancing is a vital component of any distributed environment. It is the process of distributing / redistributing the workload and computing resources of a system over multiple nodes with the objective of improving the system's overall performance. In most distributed environments, tasks arrive randomly and each of them have varying CPU time. This makes some of the nodes heavily loaded while others remain almost idle. Load balancing helps in moving the processes from heavily loaded nodes to lightly loaded nodes and ensures an even distribution and allocation of resources (Processor, Memory, Network). This kind of distribution maximizes the throughput and minimizes the response time.

Load balancing has several advantages in large practical systems involving web servers, cloud environments, grid computing setups, P2P systems, etc. The even distribution of load makes the system easy to scale. It not only makes an already scaled system say, having large number of nodes more efficient, but also allows smooth transitioning from a small system to a large one. Without load balancers, new nodes in a newly scaled system would not receive loads in a coordinated manner. One of the crucial aspects of large-scale applications is handling sudden surges in demands ("spikes"). In scenarios like sale in e-commerce websites or declaration of university results, large number of requests can arrive at the same time which can cause system breakdown. By using load balancers, even these large requests can be wisely distributed among different nodes of the system and thus, can be effectively processed. With load balancing, systems can even span multiple geographical regions which helps applications work faster in more regions around the world (ex. AWS). On the whole, utilizing load balancers improves service availability, reliability and performance. Even most cloud providers offer load balancing techniques to facilitate even workload distribution. For instance, AWS provides *Elastic Load balancing* technology for distributing traffic among EC2 instances. In fact, it forms a key architectural component of most applications utilizing AWS. Likewise, Microsoft Azure uses *Azure's Traffic Manager* to allocate its traffic across various datacenters.

As noted by Li et al. [8], typically, load balancing algorithms consist of 3 phases — information collection, decision making and data migration. *Information collection* phase collects the status of workload in each node and uses this status to detect load imbalance. In case of load imbalance, *Decision making* phase computes the optimal load distribution. *Data migration* phase then uses the computed load distribution to redistribute the workload by transferring excess load from heavily loaded nodes to the lightly loaded/idle ones.

We next discuss various types of load balancing algorithms, then present major design issues & challenges and finally describe some popular load balancing systems.

1.1 Types of Load Balancing algorithms

Load Balancing algorithms can have different classification methods. We present 5 common basis on which these algorithms are classified:

1. **Based on Initiator:** The load balancing algorithms can be initiated by an overloaded node, underloaded node or both and based on this, the algorithms can be categorized as Sender-Initiated, Receiver-Initiated, or Symmetrically-Initiated. In *Sender-Initiated* algorithms, an overloaded node searches for an underloaded node to which it sends its extra load. Generally, repeated random choices are made for the potential target node and the polled queue length (or the required definition of load) of the chosen node is utilized to determine if it is a suitable receiver. On the contrary, in *Receiver-Initiated* algorithms, an underloaded node searches for an overloaded node which sends it a task to execute. In this class of algorithms, generally, a randomly chosen node is probed to check if it is overloaded by making use of its load indicator. *Symmetrically-Initiated* algorithms have both sender-initiated and receiver-initiated components. In case of low system loads, sender-initiated component finds under-loaded nodes with ease and in case of heavy system load, receiver-initiated component is able to find the over-loaded nodes with ease.

2. **Based on Information Collection:** Depending upon whether the application's and system's current state information is used or not, load balancing algorithms can be divided into Static, Dynamic, and Adaptive. *Static* algorithms assume that the system's state does not change over time and use only the prior knowledge of the system for making their load balancing decisions. *Dynamic* load balancing algorithms, on the other hand, utilize the application and system state information, fully or partially, in their load balancing decisions. Static algorithms have low overhead as compared to dynamic algorithms as the latter needs to collect, store and process the system state. However, static algorithms are not capable of handling variations in state and thus, are likely to do a poorer job than the dynamic ones in balancing the load. *Adaptive* algorithms are a special case of dynamic algorithms which change their parameters to suit the changing system needs.

3. **Based on System's View:** Based on the available system's view, the load balancing algorithms can be classified as Centralized and Distributed algorithms. As the name suggests, *Centralized* load balancing algorithms have a central manager which collects load information and does load balancing based on global knowledge. In *Distributed* load balancers, each node keeps a local view of system's load (say, based on its neighbors) which is then used for making load balancing decisions. Centralized algorithms require global synchronization which might be costly. On the other hand, Distributed algorithms may take a lot of time (typically, quadratic in number of nodes) to converge.

4. **Based on Performance Desired:** The load balancing algorithms can also be divided into Application-level and System-level algorithms based on the end-goal. *Application-level* algorithms are concerned with minimizing the completion time of a job. *System-level* algorithms aim to maximize the overall process throughput or the overall utilization of nodes.

5. **Based on mechanism of Deployment:** A load balancer can be deployed as a hardware or as a software. In *Hardware* load balancers, a dedicated hardware such as a router or a switch is deployed between the server and the client which does load balancing. *Software* load balancers have a software running at different nodes (centralized or distributed) which handles balancing of load.

Load balancing plays a critical role in high-performance computing, and thus, has been extensively studied in academia and has several commercially deployed systems as well. Some of the popular Load balancers proposed traditionally include Round Robin, Weighted Round Robin, Least Active Connections, and Randomized [13]. More recently, researchers have worked towards tackling specific challenges while utilizing load balancers in different applications. For instance, there has been a lot of study on building load balancers for cloud environment with systems like VM-Assign Load balancer [3], User-Priority Aware Load Balanced Improved Min-Min scheduling (PA-LBIMM) [2], 2-Phase Load Balancer [14], Exponential Smoothing forecast based Load Balancer [12], ANT Colony [9], etc. getting proposed. Similarly, in a Grid computing setup, works like DRUM [5], and random sampling based load balancer [11] have been proposed. There have been several commercially deployed load balancers such as Google's Maglev [4], F5 [1], Fortinet [2], NGINX [3], Barracuda Load Balancer [4] as well.

2 Major Design Issues & Challenges

As discussed above, Load balancing has several advantages especially if employed in real-world distributed systems. But, in order to develop load balancers for such systems, several design issues are needed to be addressed. Moreover, the nature of distributed systems and the user requirements pose several additional challenges to any load balancing scheme. Below we discuss the major design issues and challenges encountered while building load balancers:

1. **Estimating the Load:** The first decision one needs to make while building any load balancer is what should be the definition of load at a node in distributed system and what measure should one use for computing it. In order to define a load, one needs to figure out what is a good indicator of load for the targeted application. The most common definition of load used by many systems is the *CPU queue length* (length of queue at a node) as it is correlated with completion time of tasks and is easy to compute. Some of the other definitions of node popularly used are CPU utilization of the node (measured by periodically recording the state of CPU), total number of processes running at the node, resources required by the running processes, etc. However, the stronger the definition of load is (eg. CPU utilization), the larger is the complexity of its computation. Simpler definitions such as number of processes are easier to compute and thus, reduce the overhead.

2. **Policies Employed:** A load balancing algorithm comprises of 4 major decision making steps — Transfer, Selection, Location & Information Policy. *Transfer Policy* is responsible for deciding whether a task should be transferred to/from a node (usually based on a threshold).

[1]https://f5.com/
[2]https://www.fortinet.com/products/wan-appliances/fortiwan.html
[3]https://www.nginx.com/resources/wiki/
[4]https://www.barracuda.com/products/loadbalancer

Selection Policy deduces the task which should be transferred among the eligible tasks (usually based on remaining execution time, waiting time, etc.). *Location Policy* figures out the target node to which the task should be transferred to (usually based on polling, randomly, etc.). Finally, *Information Policy* decides when and how frequently should the system state collection procedure be triggered, where should the state be collected from and what information needs to be collected.

A load balancing algorithm needs to wisely define these policies depending upon its application. Moreover, it also needs to overcome the overhead which some of these policies might offer (eg, we might need to collect the information from other nodes as per our location policy which will incur delay).

3. **Network Design:** While designing a load balancer, we need to take care of the kind of network formed by the distributed system. While a network with closely located nodes (say, intranet) is likely to incur negligible communication delays, a network with spatially distributed nodes might have significant communication delays. The load balancing algorithm should take into account, the effect that such a network variation might have on it. Based on the application, some load balancing algorithms might be tolerant to high delays, but some might have them as the bottleneck.

4. **Complexity of the Load Balancing Algorithm:** Ideally, load balancing algorithms should be less complex both in terms of their implementation and the computational overhead incurred by their operations [1]. High complexity would also require it to collect more information and would thus, experience higher communication delays. Hence, a highly complex algorithm might effect the overall performance of the system.

5. **Meeting Desired Performance Criterion:** Different applications may focus on different class of performance measures. Thus, while designing a load balancing algorithm, we might be required to tune it to maximize the targeted performance criterion even if it compromises on some other important measures. Some applications might focus on minimizing the overall completion time of the tasks while some might just be concerned about the overall throughput. Some applications might target to get the same amount of load at each node while some might want to reduce the waiting time of the processes. The desired performance forms a critical aspect of any load balancer.

6. **Prioritizing Tasks:** We might be required to prioritize certain tasks in a distributed system (common in a cloud computing environment). A load balancing algorithm should consider the priority associated with the task before distributing it to different nodes.

7. **Task Deadlines:** Each task might have a deadline before which it needs to be completed. The load balancer needs to acknowledge this deadline before making any decisions. Transferring the task to another node might reduce the overall completion time of the system but it might imply crossing the deadline of certain tasks. This might not be acceptable in certain scenarios.

8. **Heterogeneity:** A system might have heterogeneity both with respect to the incoming tasks and the nature of nodes in the system. In a typical distributed system, the incoming tasks

have varying resource requirements. Different tasks will thus require different handling by the load balancer. Moreover, each node might have different computation power, storage, memory, etc. The heterogeneous nature of the tasks and the nodes acts as an added challenge to the load balancer as it needs to account for the nature of load.

9. **Scalability:** The load balancing algorithm should allow the system to scale to user and traffic growth. Normally, in order to scale, a *scale-out model* (like Google's Maglev) is adopted where more number of nodes are added to the system. The load balancing algorithm needs to quickly adapt to the addition of new nodes or removal of nodes in a large distributed system. It should be effective in balancing load in the scaled system while at the same time being able to make quick decisions with low overhead.

10. **Availability & Reliability:** A load balancer should be highly available, be quick to response and should be adaptable to faults of nodes involved in load collection and redistribution. Many load balancers just employ one controller for the whole system. Thus, if the controller fails, the entire system would fail. To counter this, some load balancers are often deployed in pairs to avoid single point of failure. However, this would still just entail a 1+1 redundancy. The distributed load balancers do a much better job at handling faults, but they are more complex to design.

11. **Migration time:** Migration time is the amount of time needed for transferring a task from one system node to another node during load redistribution. This time has to be minimized for better performance.

12. **Flexibility & Programmability:** A load balancer should be flexible and it should be easy to modify its different components.

13. **Upgrading Overhead:** A load balancer should be easy to upgrade if needed, say, for incorporating a newly arrived technology.

14. **Determinism & Preemption:** A load balancing algorithm needs to ensure *Determinism*, i.e., a task transferred to some other node should output the same result as it would had originally. Moreover, the load balancing algorithm needs to make sure that transferring a task to a node should not significantly deteriorate the performance of the target node as viewed by the original user of the node. In case of a degraded performance, there should be a mechanism for preempting the transferred task and releasing the resources for running tasks of the node's owner.

15. **Transparency:** A load balancer should not be transparent to the clients of the distributed system. For ensuring this, the client should get the feedback from the same node which it originally connected to and not the one where the task actually got executed.

3 Popular Load Balancing Systems

Next, we describe some of the popular load balancing systems.

3.1 Traditional Load Balancing Algorithms

There are 3 common algorithms used by many traditional as well as commercial load balancers:

1. **Round Robin:** In a round robin load balancer, nodes are sequentially ordered and tasks are assigned to nodes in a rotating fashion. For the first task, a random node is chosen and the subsequent tasks are assigned in a circular order. Though all nodes have equal number of tasks assigned, the tasks might be heterogeneous in nature having varying complexities which might make some of the nodes heavily loaded and some idle. Nodes are also assumed to have similar configuration.

2. **Weighted Round Robin:** In weighted round robin, a weight is associated with each node and tasks are assigned to the nodes in a circular order based on weights. So, two tasks will be assigned to a node having a weight of 2. This kind of approach can handle heterogeneous nodes but is not capable of efficiently handling heterogeneous tasks and loads.

3. **Weighted Least Active Connections (WLC):** In WLC, a node handling least number of persistent connections is chosen for assigning a task. This is suited for systems having large number of persistent connection with uneven distribution of traffic across nodes. This also works well for session aware balancing where a client must get the feedback from the server it connected to.

3.2 Exponential Smoothing Forecast Based Load Balancing [12]

Exponential Smoothing Forecast [7] is a prediction algorithm based on time series. It uses a smoothing factor, α, for setting a greater emphasis on most recent value than the historical data for prediction of the upcoming value. Let $x_1, x_2, x_3, ...$ denote a time series. Single exponential smoothing predicts the value x_{t+1} using:

$$\hat{x}_{t+1} = \alpha * x_t + (1 - \alpha) * \hat{x}_t$$

where $\hat{x}_{t+1}$ is the predicted value at $t + 1$ and $\hat{x}_t$ is the predicted value at t. α is the smoothing factor. Exponential smoothing forecast-based on weighted least connection (ESBWLC) makes use of the above prediction scheme for load balancing. It predicts the assignment of task based on the experience of the node's capabilities tracked over a large time series.

3.2.1 ESBWLC Architecture

Let L_{CPU} denote CPU utilization of a node, L_{MEM} denote the memory usage of the node, L_{NET} denote its number of connections and L_{DIO} denote the disk occupation of the node. Then, ESB-WLC defines the total load at a node as:

$$L = r_{CPU} * L_{CPU} + r_{MEM} * L_{MEM} + r_{NET} * L_{NET} + r_{DIO} * L_{DIO}$$

where r's denote the coefficient used as indicators of impact factor, each of the load component would have in the overall load of a node. ESBWLC works on the $\frac{Load_i}{P_i}$ value for each node i in the system (P_i denotes power of node i). It creates a time series of these values and predicts the upcoming value using Exponential smoothing forecast. At any time instant, the node with minimum forecasted value is chosen for assignment of a task.

3.2.2 Discussion on the Design Issues Handled

ESBWLC computes the load as a combination of 4 different components of a node. This might act as an overhead for each node as these components need not be directly available. Moreover, it also needs to make an estimation of their respective coefficients which may change with time. However, as it uses the estimated value of load for the current time instant, it is capable of handling heterogeneous tasks as well as heterogeneous nodes. As the decision of the node where a task would be run is made only once and no redistribution of tasks is handled, there would be minimal network overhead and it should work well for spatially distributed nodes as well. However, for estimating different values complex computations might be required which makes this algorithm implementation heavy.

3.3 User-Priority Aware Min-Min Scheduling Algorithm for Load Balancing (PA-LBIMM) [2]

One of the key issues of cloud computing environments is load balancing. Cloud environments have unique characteristics which act as an added challenge to the traditional load balancing algorithms. Firstly, they comprise of heterogeneous resources (operating systems, memory, architecture, multiple resource providers) and heterogeneous tasks (multiple resource consumers / user demands). Moreover, cloud environments typically work on pay-per-use basis where different level of services may be offered to different consumers in order to fulfill varying level of demands. This means that a load balancing algorithm designed for cloud additionally needs to take into account, the priority given to a user (VIP user or ordinary user) while meeting their demands. Traditional algorithms such as Min-Min scheduling neither do a reasonable load balancing in heterogeneous scenarios nor do they take care of priority assigned to a user.

PA-LBIMM has been designed to schedule tasks in a cloud computing environment while addressing the above mentioned issues. We first discuss min-min scheduling algorithm and then describe PA-LBIMM.

3.3.1 Min-Min Scheduling

Min-Min scheduling aims at minimizing the completion time of all tasks. Let S be the set of all unmapped tasks. Min-Min scheduling first finds the node i which will take the minimum time to complete all the tasks and then assigns the minimum sized task i to the selected node j. The task i is removed from S and the same process is repeated till S is empty. The Expected Completion time CT_{ij} for task i on node j is defined as:

$$CT_{ij} = ET_{ij} + RT_j$$

where ET_{ij} is the Execution Time and RT_j is the ready time for node i.

3.3.2 User-Priority Aware Load Balanced Improved Min-Min Scheduling

Chen et al. [2] showed that Min-Min algorithm fails to utilize the resources efficiently and suffers from load imbalance. As it does not considers the load at individual nodes, this might make some nodes very busy while some may remain idle. To overcome this, LBIMM does the following:

- **Step 1:** $\forall\, i \in S, j \in nodes$, compute CT_{ij}
- **Step 2:** Run Min-Min scheduling algorithm and get all the task to node assignments.
- **Step 3:** Find the most heavily loaded node. Let that node be V_j. Choose task i which has the minimum execution time on V_j.
- **Step 4:** Find k such that CT_{ik} is minimum. If $CT_{ik} < \max_{ij} CT_{ij}$, reassign task i to k.
- **Step 5:** Repeat steps 3 & 4 till no rescheduling needed for the most heavy node.

LBIMM ensures effective load balancing. However, as discussed before, in a cloud setup, we might as well need to take care of the priority assigned to a user. To handle this, PA-LBIMM divides the received tasks into groups $G_1, G_2, ..., G_k$ ordered by priority where G_1 represents the highest priority task and G_k represents the lowest priority task. In the Min-Min scheduling step of LBIMM, tasks are scheduled as per the above order: tasks in G_1 are scheduled first, then G_2, G_3 and so on. And load balancing is performed on top of it. This makes it a User-Priority aware load balancer.

3.3.3 Discussion on the Design Issues Handled

PA-LBIMM is effective in balancing the load across nodes because:

1. It always moves tasks from the most heavy node. Also, it chooses the target to be the one where CT_{ik} would be minimum. This ensures that the target node will be light on load or idle. Thus, it always frees up a heavy node and assigning tasks to a lighter node imposing load balance.
2. $\max_{ij} CT_{ij}$ will be determined by the completion time of most heavily loaded node. Choosing k such that $CT_{ik} < \max_{ij} CT_{ij}$ ensures that task i will be moved to a node whose new completion time won't be greater than the overall completion time of the system. Moreover, moving the task from the heavily loaded j will make $\max_{ij} CT_{ij}$ smaller. Thus, the overall completion time of the system is reduced.

PA-LBIMM is aimed at reducing the overall completion time (*makespan*) of the system and handling user priority. Load at a node is computed in terms of estimated completion time of all the tasks running at that node. Transfer, Selection and Location policies are self-explanatory as per the algorithm and have been designed to maximize the makespan. It, however, does not take care of the geographic location of the nodes and is thus, likely to suffer with large communication delays in case of spatially distributed nodes. As it uses estimated execution time, ET_{ij} for a task i on node j, it is capable of handling heterogeneous tasks as well as heterogeneous nodes. In terms of handling scalability, the algorithm does have scope of adapting to addition of new nodes and should do a fine job in balancing the load. However, as it goes over all the nodes for all its load balancing decisions, it is likely to slow down as the system scales. No effort has been made in making the system fault tolerant. However, popular fault handling techniques can be employed for doing the same.

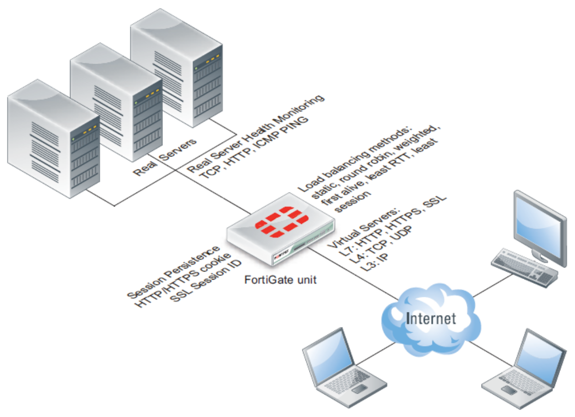


Figure 1: Configuration of FortiNet's Hardware Load Balancer (Source - FortiNet Load Balancing Handbook)

3.4 FortiNet ADC: A Hardware Load Balancer

The increased traffic in websites post 1990s required use of additional servers. To incorporate it, methods like DNS resolution were proposed. DNS allowed addition of large number of servers (all associated to a DNS name). However, the capacity of these servers was still needed to be known in order to direct the requests wisely. To tackle this, software load balancers could have been used, but they were facing issues in scaling. Hardware-based load balancing appliances were thus proposed. Load balancing was separated from the applications and thus, the load balancer could rely on network layer techniques such as NAT for directing the traffic in a balanced manner. These appliances also had a server health-checking component, which would periodically check the server's status in terms of its availability and load. This ensured that the load be balanced across different servers accordingly. The traditional hardware load balancers work on Layer 4 (Transport Layer) for making their load balancing decisions.

3.4.1 Application Delivery Controllers

Over the past decade, hardware load balancers have evolved into *Application Delivery Controllers* (ADCs). ADCs distribute load to servers based on the type of content requested by a user. For instance, the request for an image might go to a server dedicated to images; servers for static content like HTML, javascript, css, etc. might be different from servers for PHP driven requests.

3.4.2 FortiNet ADC Description

FortiNet is a commercial hardware and virtual ADCs provider. It guarantees a high load balancing performance and at the same time addresses scalability. It also supports additional features such as SSL Offloading (offloading of resource intensive SSL transactions to its ADCs), HTTP Compression (compressing data for reducing network traffic), Global Server Load Balancing (deployment supported in any geographic location), Firewall and Link Load Balancing (distribution of traffic over multiple ISPs for increasing resilience and reduced bandwidth upgrades). Its advanced models include 10-GE SFP+ ports, hardware-based SSL ASICs, dedicated management channels and dual power supplies to meet the demands of datacenter environments with throughput upto 50 Gbps.

3.5 Maglev: Google's Load Balancer [4]

Google is one of the leading platforms available on the Internet which provides a large number of services covering a huge amount of audience all over the world. It thus receives a huge amount of requests from its large consumer base. To handle such huge amount of traffic, effective and reliable load balancing techniques are required. Google hosts its different services on a number of replicated servers all around the world. These replicated servers exist in clusters at a particular location to satisfy low response latency to the clients.

Earlier, Google used to implement load balancers as dedicated hardware devices (**hardware load balancers**). However, dedicated hardware devices are not flexible, they have fixed capacities and may fail if the amount of requests are more than their overall capacity. Moreover, if they need to increase their capacities (for handling increased requests), they need to be upgraded which is quite costly. Thus, hardware load balancers were unable to cope up with the Google's traffic growth, due to which they looked for alternative solutions to solve the load balancing problem.

Thus, Google proposed a software load balancer – *Maglev* which effectively addressed scalability, is highly available & reliable, ensures even distribution of traffic and provides connection persistence. To incorporate these features, Maglev uses *Equal Cost Multipath* (ECMP) forwarding and *connection tracking*. Figure 2 gives an overview of Maglev.

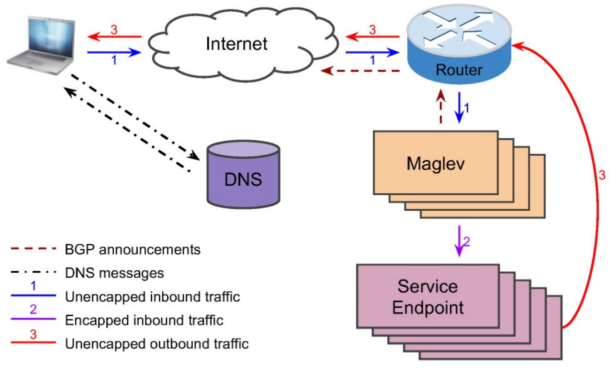


Figure 2: An overview of Maglev (Image Source - Eisenbud, Daniel E., et al. "Maglev: A Fast and Reliable Software Network Load Balancer." NSDI. 2016)

3.5.1 Maglev's Architecture

Now, we describe the architecture of Maglev and how it performs load balancing. Each service offered by Google has one or more Virtual IPs (VIPs). Each VIP represents all the (cluster of) service endpoints of a particular service. Google DNS server maps the domain names of the services to their respective VIPs. A request from a client is handled by Google as follows:

1. When a request from a client, associated with a particular service, arrives at the DNS server, the user is directed to the service's Virtual IP taking into consideration the user's geo-location as well as the current load at each location.

2. The client then establishes connection with the VIP it is supplied with.

3. A router upon reception of the above VIP connection packet, forwards it to one of the Maglev's machines present in the cluster of the VIP using ECMP forwarding (to evenly distribute the traffic amongst the nodes in the cluster).

4. The Maglev machine then selects a suitable service endpoint belonging to the service's VIP from the cluster and forwards the request packet to that endpoint.

5. The service endpoint then receives and processes the packet and a required response is sent back directly to the router (which had directed the request packet to Maglev) to be forwarded to the client. This is done using Direct Server Return (DSR).

6. The response packet is then transferred to the client using routing measures that the network may be following.

3.5.2 Discussion on the Design Issues Handled

Maglev is scalable. It ensures this by adopting the *scale-out model*. In Maglev, the capacity of the load balancer is improved by increasing the number of machines in the system and through ECMP forwarding, traffic is evenly distributed across all machines.

Maglev is also highly available and reliable and provides N+1 redundancy. In order to incorporate fault tolerance, incoming packets must be forwarded to only non-faulty / healthy Maglevs. Each Maglev machine consists of a controller and a forwarder. The controller periodically checks the forwarder's health and depending upon the results received it announces or withdraws the VIPs from the DNS server via BGP. The forwarder handles the incoming VIP packets with each VIP being associated with one or more service endpoints (directly or recursively with VIP being associated to a backend pool that in turn points to another backend pool). Also, all backend pools are associated with health checkers for ensuring packet forwarding only to non-faulty backends. The forwarder is responsible for quickly and reliably handling the huge number of packets it receives.

We also need to take care of that for TCP connections, a client must communicate (send and receive packets) with the same service endpoint to which it was directed the first time. But, following the load balancing procedures, we must also select the service endpoint for a new incoming connection such that the traffic is distributed evenly. Google incorporates this by using two simple techniques in the Maglev forwarder:

1. Maintaining a lookup table

   (a) Maglev assigns a backend to a new incoming connection/packet and stores this assignment in a lookup table in case the entry for the connection is not already present.

   (b) Packets from the same client/IP are then routed to the same backend service endpoint by utilizing the lookup table entry for that connection.

2. Selecting a service endpoint for each new incoming connection using *consistent hashing* such that it evenly distributes the traffic.

   (a) *Maglev hashing* assigns a preference list of all the lookup table positions to each of the backend service endpoints as follows:

i. It incorporates two hash functions for each backend service endpoint, namely offset and skip defined as:

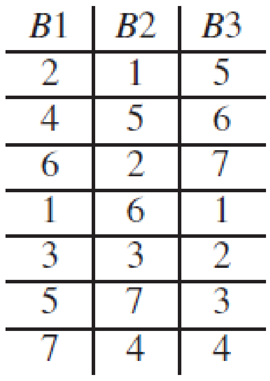
$$offset = hashfn1(name\ of\ backend\ service\ endpoint) mod M$$

and

$$skip = hashfn2(name\ of\ backend\ service\ endpoint) mod(M - 1) + 1$$

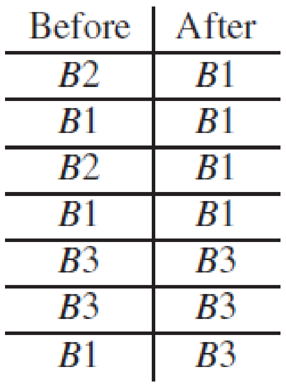
ii. Here, name of each $service\_endpoint$ is unique and M is a prime number with $M > 100 * N$ with N being the size of a backend pool. Sample preference lists of the endpoints are shown in Figure 3a.

(b) Now, backend service endpoints pick the first available slot in their preference list in turns and fill the lookup table. This also assists in maintaining the relative stability of the mappings whenever a service endpoint is added or removed. This stability is depicted in Figure 3b.



$B1$	$B2$	$B3$
2	1	5
4	5	6
6	2	7
1	6	1
3	3	2
5	7	3
7	4	4

(a) Preference table for each endpoint



Before	After
$B2$	$B1$
$B1$	$B1$
$B2$	$B1$
$B1$	$B1$
$B3$	$B3$
$B3$	$B3$
$B1$	$B3$

(b) Stability depiction in mappings

Figure 3: Maglev Hashing (Image Source - Eisenbud, Daniel E., et al. ”Maglev: A Fast and Reliable Software Network Load Balancer.” NSDI. 2016)

In summary, Google, smartly and quickly, routes and distributes the traffic amongst its various redundant servers for various services offered and ensures high Quality of service (QoS) and reliability to its users. It achieves all this by using ECMP to distribute the traffic amongst various Maglevs which in turn distributes traffic among the various redundant servers (service endpoints) based upon their current load.

3.6 Duet: Hardware And Software based Load Balancing [6]

A load balancer is one of the key component of any cloud service system. The demand for cloud services is increasing rapidly and to scale up dedicated hardware is being added. Adding hardware devices is very costly and instead software based load balancers are being used. They are easily deployable on commodity servers and offer low cost with high availability. However, the major drawback of any software based load balancer is low capacity and high latency. Duet, a hybrid solution, uses software based approach with low latency and high throughput by using an overlooked resource of the network, the switches. Duet uses switch based load balancing with software load balancing a backstop.

Every modern day cloud service provides a variety of services running on multiple servers each with their individual direct IP address (DIP). These services advertise one or few virtual IP (VIP). The load coming into these set of VIPs is split and routed among various DIPs.

3.6.1 Ananta Software Load Balancer [10]

The software load balancer used in Duet is Ananta which runs on commodity servers. Its architecture is highly scalable with high availability. It consists of a central controller and several software muxes (Smux). Each of these Smux maintains VIP-DIP mappings for all the VIPs configured in the data center (DC) to ensure traffic splitting. The Smux advertises itself as next hop for every VIP using BGP protocol. All the packets are directed to one of these Smux using the ECMP (Equal-Cost Multi-path) protocol and then forwarded to respective DIPs. It mainly relies on ECMP to distribute the traffic among various Smuxes.

3.6.2 Duet's Architecture

Duet is based on leveraging the data center switches as hardware based load balancers (Hmux) without need of any extra hardware support. The hardware switches offer low latency with high capacity but have certain drawbacks. So to overcome these, the hardware switches (Hmux) are combined with software load balancer (Smux), Ananta. It relies on ECMP for traffic splitting and modern day switches come with API to control these ECMP and tunneling tables. By carefully customizing these tables we can make commodity switch act as a Hmux.

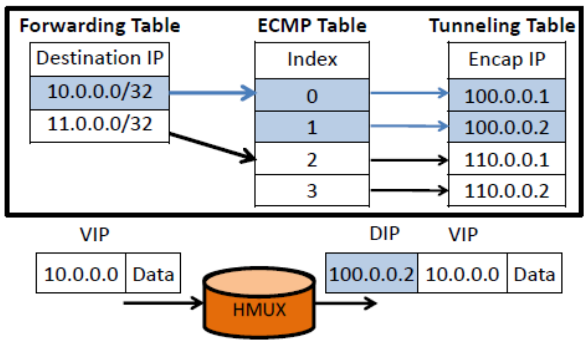


Figure 4: Design of Hmux (Image Source : Gandhi, Rohan, et al. ”Duet: Cloud scale load balancing with hardware and software.” ACM SIGCOMM Computer Communication Review 44.4 (2015): 27-38)

Figure 4 shows the design of a Hmux. A packet entering the switch is processed and matched with one entry of the forwarding table. This entry further points to multiple entries in ECMP table that represent next hop information for the packet. This directs to multiples servers of a particular

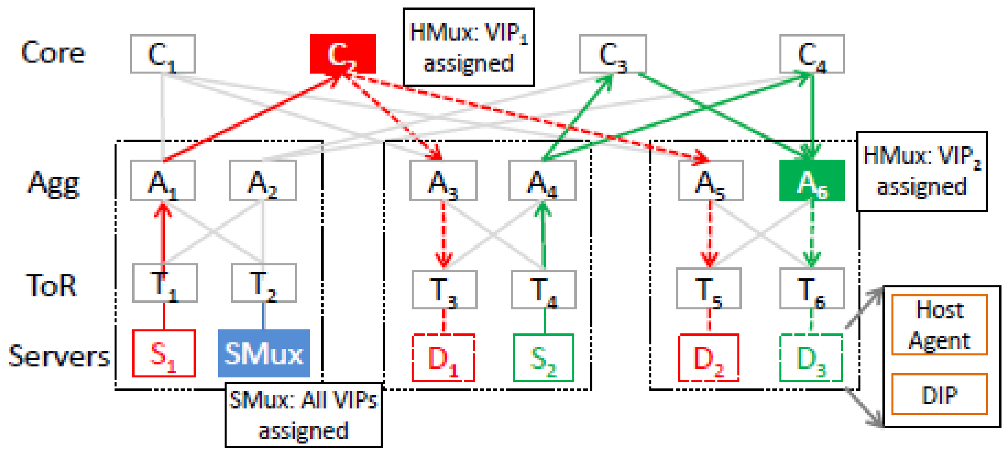


Figure 5: Duet Architecture (Image Source : Gandhi, Rohan, et al. ”Duet: Cloud scale load balancing with hardware and software.” ACM SIGCOMM Computer Communication Review 44.4 (2015): 27-38)

cloud service which is simply splitting the traffic among various replicated servers. The issue with this design is the limited amount of space in switches. We cannot store all the mappings of VIP-DIP in single switch. The host forwarding table accounts for 16k entries but in a large DC they may not be sufficient to store all its VIPs. Next issue is that ECMP table accounts only for 4k entries and the tunneling table for 512 entries, thus we can store only 512 DIPs at maximum in a single Hmux.

Duet addresses this problem by using two methods: (1) Dividing the VIP-DIP mappings to multiple switches but stores all DIPs of anindividual VIP(of particular service) in a single switch. This means only a subset of VIPs are stored in a single switch and traffic to a particular VIP is directed to that specific switch. (2) Advertising the switch assignment of VIPs using BGP, so that others can route the packets of a VIP to that particular switch. Figure 5 illustrates that VIP1 has D1 and D2 as its DIPs assigned to switch C1 and VIP2 has D1 assigned to switch A6. Links with solid lines indicate VIP traffic and the ones with dashed lines indicates DIP traffic. Smux acts as backstop to Hmux in case of failures.

This method also achieves organic scalability, that is when number of servers in DC are increased, the traffic demand also increases and to handle it more switches need to be incorporated thus increasing the aggregate capacity of Hmuxes proportionally. While partitioning of the VIP-DIP mappings allows us to use more number of DIPs, but still it is limited. This design also lacks VIP availability during network failures as VIPs are assigned to only single specific Hmux. This may be overcomed with replication of VIPs on multiple switches which improves the ability to handle failures but still cannot make it highly available.

Assigning of VIPs to specific switches is done on the basis of their traffic volume and maximum resource utilization (MRU) of the switches and links in network of the DC. Assigning the VIPs to different switches leads to different MRU. Among the various combinations of VIP assignment, Duet selects the one that leads to minimum MRU. If the smallest MRU is more than 100%, implies no assignment can handle the load in which case no VIP is assigned to switches and the entire load balancing is done only by the Smuxes. Due to the network failures and traffic dynamics, the VIP-DIP mappings previously calculated become outdated. A periodic update of the VIP-DIP mappings needs to be done. Duet recalculates the VIP assignment that can handle the new traffic with reduced MRU and migrate old assignment to the new one. This dynamic approach to handle dynamic nature of network with least MRU assignment and Smux as backup to failures or delays makes it a highly available system.

3.6.3 Discussion on the Design Issues Handled

1. **Hmux / Switch Failure :** This is detected by neighbouring switches of the failed switch. Routing entries related to failed switch are removed from all other switches using BGP withdraw message. After the process of routing convergence, the new VIP assignments of failed switch are forwarded to Smux as they are the components which advertise the VIP information.

2. **Smux Failure :** This has no effect on the Hmux and is a problem to those VIPs that are assigned only to that specific Smux. The switches can easily detect the Smux failure and redirect their traffic to other Smuxex using the ECMP protocol. It does not break any network connections but may suffer some amount of packet drop during convergence.

3. **Link Failure :** If a link failure leads to isolation of a switch, it is simply handled as a switch failure. It may lead to some amount of rerouting but does not impact the availability of a system as the Smux takes over as load balancer during transition.

4. **DIP Failure :** The DIP or server health is monitored continuously and the set of DIPs of a specific VIP are updated by removing the failed DIP entry. The existing connections of that particular DIP are terminated but connections to other DIPs are still kept intact.

5. **VIP Addition :** Firstly it is added to Smux and later the migration algorithm assigns it to the appropriate switch which leads to least MRU assignment.

6. **VIP Removal :** The entries are removed from both Hmux and Smux and then corresponding routing entries from all switches are removed using the BGP withdraw messages.

7. **DIP Addition :** To add a DIP to particular VIP, it is first removed from the switch. We then add DIP to the VIP and move it to appropriate Hmux. During this transition the Smux takes over to load balance that particular VIP service.

8. **DIP Removal :** It is handled in a similar way to that of DIP failure.

9. **Heterogeneous systems :** In case of heterogeneous servers we can use WCMP (Weighted-cost Multi-path) with weighted distribution of traffic proportional to the capacity of the servers.

In summary, Duet uses the ECMP/WCMP to distribute load across various DIPs of a particular VIP and relies on Smux as backstop when there is failure of switches or during rerouting. Duet provides a low cost, highly available load balancing method with latency reduced by a factor of 10 over a pure software based load balancer.

References

[1] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. A survey of load balancing in cloud computing: Challenges and algorithms. In *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, pages 137–142. IEEE, 2012.

[2] Huankai Chen, Frank Wang, Na Helian, and Gbola Akanmu. User-priority guided min-min scheduling algorithm for load balancing in cloud computing. In *Parallel computing technologies (PARCOMPTECH), 2013 national conference on*, pages 1–8. IEEE, 2013.

[3] Shridhar G Domanal and G Ram Mohana Reddy. Optimal load balancing in cloud computing by efficient utilization of virtual machines. In *Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*, pages 1–4. IEEE, 2014.

[4] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI*, pages 523–535, 2016.

[5] Jamal Faik, Luis G Gervasio, Joseph E Flaherty, Jin Chang, James D Teresco, Erik G Boman, and Karen D Devine. A model for resource-aware load balancing on heterogeneous clusters. *Computing and Science Engineering*, 2005.

[6] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2015.

[7] Everette S Gardner. Exponential smoothing: The state of the art. *Journal of forecasting*, 4(1):1–28, 1985.

[8] Yawei Li and Zhiling Lan. A survey of load balancing in grid computing. In *International Conference on Computational and Information Science*, pages 280–285. Springer, 2004.

[9] Kumar Nishant, Pratik Sharma, Vishal Krishna, Chhavi Gupta, Kuwar Pratap Singh, Ravi Rastogi, et al. Load balancing of nodes in cloud using ant colony optimization. In *Computer Modelling and Simulation (UKSim), 2012 UKSim 14th International Conference on*, pages 3–8. IEEE, 2012.

[10] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[11] O Abu Rahmeh, P Johnson, and A Taleb-Bendiab. A dynamic biased random sampling scheme for scalable and reliable grid networks. *INFOCOMP*, 7(4):1–10, 2008.

[12] Xiaona Ren, Rongheng Lin, and Hua Zou. A dynamic load balancing strategy for cloud computing platform based on exponential smoothing forecast. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 220–224. IEEE, 2011.

[13] Subhadra Bose Shaw and AK Singh. A survey on scheduling and load balancing techniques in cloud computing environment. In *Computer and Communication Technology (ICCCT), 2014 International Conference on*, pages 87–95. IEEE, 2014.

[14] Shu-Ching Wang, Kuo-Qin Yan, Wen-Pin Liao, and Shun-Sheng Wang. Towards a load balancing in a three-level cloud computing network. In *Computer Science and information technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 1, pages 108–113. IEEE, 2010.