Distributed System - Service Discovery & Load Balancing

Anushree Desai and Divyesh Chitroda

Changelog:

1. Initial Draft - 11/08/2018
2. Revised - 11/21/2018

1. Overview

Implement a distributed system to communicate with web services. There is a many-to-many relationship between collection of **web services** and **sites** hosting those services. We have to build an architecture to efficiently locate and execute a web service majorly using two concepts: service discovery and load-balancing.

2. Abbreviations and notations

LB - Load Balancer
SLB - Service Load Balancer
S# - Server #
"|" is a API call chaining and not OR operator
SR - Service Registry
CSR - Cached Service Registry
HB - Heartbeat

3. System Architecture and Design

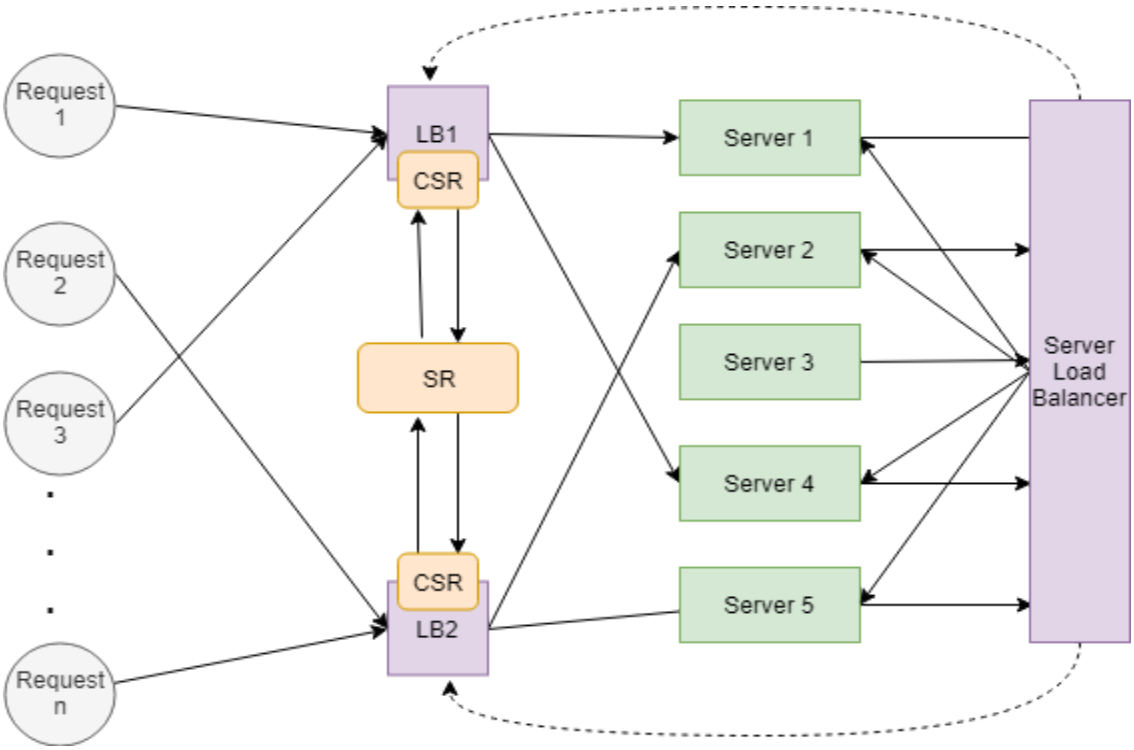

Figure 1: LB1, LB2, SLB, SR and client interactions

3.1. **LB1** and **LB2** are load balancers used to handle multiple requests from client to web services. They are hosted on domain name as services.maths.com on different machines and will have different IP address.

3.2. Each web service may reside on one or more than one servers. Thus, we need a load balancer to pick the least loaded site hosting that web service through LB1 and LB2.

3.3. When a service request is made using API endpoint services.maths.com/math/add?x=5&y=7, DNS resolves the host to one of the LBs in a round-robin manner.

3.4. Now the LBs will forward the *add* request to the server that hosts *add* method. The *add* web service is hosted on multiple machines. The machine to which the *add* request is forwarded is chosen by LB on the basis that the machine which is the least loaded site.

3.5. For multiplication operation, users call services.maths.com/math/mul?x=5&y=7, which in turn calls *add* and aggregates the result and sends it with the response.

3.6. **Service Load Balancer (SLB)** is used to handle load generated by multiple requests from one service to another service(basically for request forwarding).

3.7. In order to avoid single point of failure at SLB, when a request comes to SLB and if SLB is down, we forward the request to LB1.

3.8. Note that *add* is also called by the other services and not just by clients, which makes the services *add* load heavily if we always forward the the requests made by servers to one static site hosting *add*. To overcome this, we have added another load balancer SLB, that distributes the requests for sites that are requested by other services.

3.9. **Service Registry (SR)** is a central repository which maintains a list of all the web services, sites that hosts those services and metadata information.

3.10. Service description for each service will be fetched through WSDL and REST API. It will contain information such as service name, endpoints, service status.

3.11. We maintain the SR data cached at LB1, LB2 and SLB and we use them to discover services in order to avoid single point of failure at SR. The cached registries are updated whenever the status of any service changes.

4. Functional Requirements

4.1. Web service interface

Name	Input	Output	Endpoint	Processing
Addition	Integer(*x*),Integer(*y*)	result(sum)	arith/add?x=6&y=-7	ans = x + y
Subtraction	Integer(*x*),Integer(*y*)	result(difference)	arith/sub?x=7&y=3	ans = x - y
Multiplication	Integer(*x*),Integer(*y*)	result(product)	arith/multiply?x=7&y=2	ans = arith/add?x=7&y=1 | arith/add?x=7&y=1 | ... (y times)
Division	Integer(*x*),Integer(*y*)	result(quotient)	arith/div?x=5&y=2	ans = arith/sub?x=5&y=1 | arith/sub?x=5&y=1 | ... (y times)
Square	Integer(*x*)	result(square)	arith/square?x=5	ans = arith/multiply?x=5&y=5
Double	Integer(*x*)	result(double)	arith/double?x=5	ans = arith/multiply?x=5&y=2

4.2. Load balancer

4.2.1. On each load balancer machine, we maintain a min-heap for each service to store its endpoint and its current load as the key value.

4.2.2. For all services k_i, we have Q_i, where $k \in S_i$. Q is the min-heap queue, S is the server and k is the web service.

4.2.3. Each item in Q is a tuple (c,s), where c is the current load/cost on server s.

4.2.4. When we receive a request, we lookup the service request to access its list of servers available. Then we do a Extract-Min operation on the Q to get the least loaded site and forward the request to that server's endpoint.

4.2.5. We reinsert that server's endpoint into the Q by appending the new value of the load to its key value.

4.2.6. When the server responds with the reply for the request placed, we call a Decrease-Key on Q for that server and decrease its key value to reflect the current load which is now have been reduced.

4.2.7. Similarly we will implement SLB to load balance the requests generated from other services, i.e. request forwarding.

4.2.8. If SLB fails and we get 500 response status, we will redirect the request using IP endpoint (not by domain name) to LB1 as a backup for SLB.

4.2.9. APIs with LB as below:

Name	**Input**	**Output**	**Endpoint**	**Processing**
Update Cache Registry	Object (registry data)	None	/updateSR	Update current registry data

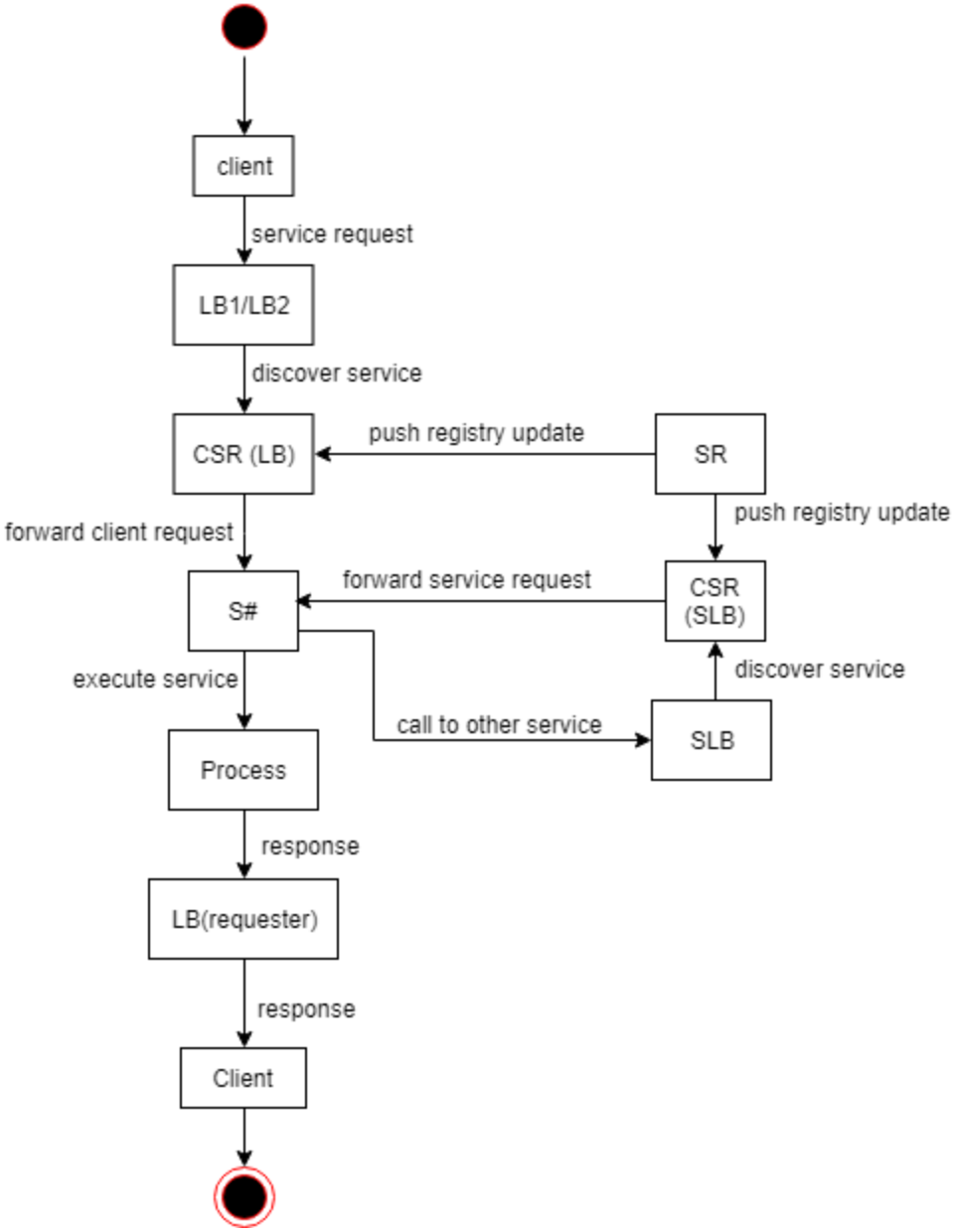

Figure 2: LB1, LB2, SLB, SR and client interactions

4.3. Service Registry

4.3.1. A central Service Registry (SR) maintains a permanent and updated list of all the services and host sites as <key, value> pairs, where key = serviceName and value = list of objects containing data of hosted sites and health status of the service 'serviceName'.

4.3.2. CSR is the cached object of SR available at LB1, LB2 and SLB.

4.3.3. SR will push update to LB1, LB2 and SLB, each time there is an update in the list of services (i.e. Each time any service registers or unregisters itself or change in service endpoint).

4.3.4. SR will always return list of servers that are up and running for a particular service.

4.3.5. The health checkup of the services is done periodically using a heartbeat mechanism as explained in below steps to check the status of any service instance.

4.3.6. When a service node is created, it registers itself to SR by calling registerService API.

4.3.7. SR will save details of the service and spawn a heartbeat socket.

4.3.8. If SR misses 5 heartbeats for a service, it will unregister the service from its registry.

4.3.9. The changes in the registry is then sent to LBs by calling *updateSR* API to update the CSR at each LBs.

4.3.10. APIs with SR as below:

Name	Input	Output	Endpoint	Processing
Discover Service	String (*serviceName*)	Result (list of endpoints hosting services) - ['10.0.0.40:5001', '10.0.0.40:5004']	service/<*serviceName*>	Fetch list of endponts from registry using *serviceName*
Register Service	String (*serviceName*), endpoint = ip & port of server	Result (success/failure)	register/<*serviceName*>?endpoint=10.0.0.50:8009	Append endpoint to list of the endpoints for *serviceName*
Unregister Service	String (serviceName)	Result (success/failure)	Method -> unregisterService(*serviceNode*)	Service endpoint will be deleted from SR

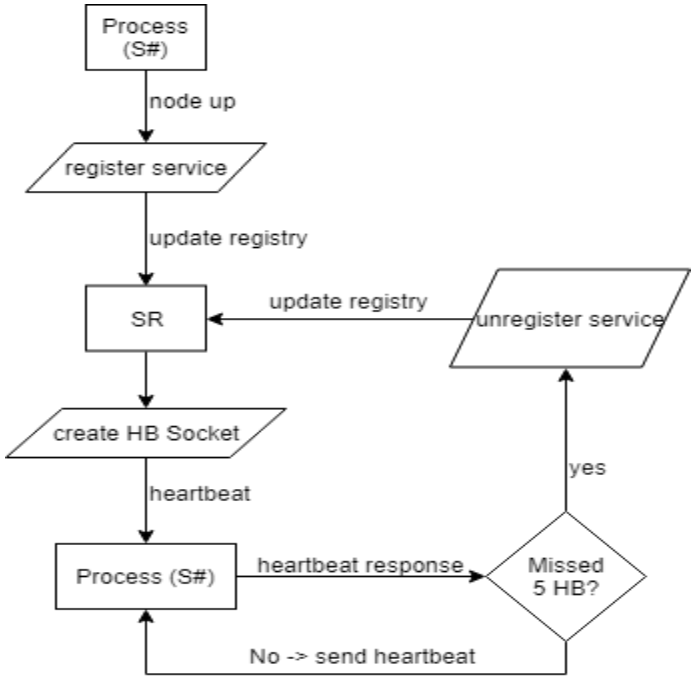

Figure 3: SR and CSR interaction

5. Testing and Simulations

5.1. Setup

5.1.1. We will use multiple VMs to simulate this distributed system.

5.1.2. One dedicated machine running LB1 and LB2 with local DNS entries in dnsmasq.

5.1.3. There are 4 dedicated machines as server A, B, C, D with services distributed as below:

Service	**Server**
Add	A, B, C, D
Multiply	B, C
Subtract	B, C, D
Divide	C, D
Square	B, D

5.1.4. One machine with multiple client scripts generating requests.

5.2. Load and Scalability Testing

5.2.1. Testing loads with multiple requests for *add* and multiple requests for *multiply.*

5.2.2. Testing load balancing when only single LB is up and when both LBs are up.

5.2.3. Testing load balancing when SLB has failed.

5.2.4. Testing the load distribution with random failure of sites hosting services.

5.2.5. Testing the failure of SR and when cached registry data is inconsistent.

5.3. Simulations

5.3.1. Traffic analysis with both LBs running and when only single LB is running.

5.3.2. Load analysis at each server.

5.3.3. Cost vs Performance analysis.

5.3.4. Scenarios with DoS attacks.

6. Assumptions

- Load-Balancing is done only for web-services
- All client applications are hosted locally
- All web services have equal priorities and all servers have equal load capacities
- Notional load is used for each service instead of actual load
- Failed nodes are recoverable in finite amount of time

7. Timeline

Tasks	Item List	Due Date	Status	Contributor	Comments
Initial Report	System Architecture	11/08/2018	Done	Anushree, Divyesh	Revised (11/19/2018)
	Desgin Goals	11/08/2018	Done	Anushree, Divyesh	
	Timeline	11/08/2018	Done	Anushree, Divyesh	Revised (11/19/2018)
Server Applications(web services)	Add, Subtract	11/14/2018	Done	Anushree	
	Multiply, Divide, Square	11/14/2018	Done	Anushree	
Load Balancer	Round Robin Alogrithm	11/21/2018	Done	Divyesh	
	LB1 and LB2 servers	11/21/2018	Done	Divyesh	
	Server Load Balancer	11/21/2018	WIP	Divyesh	
	DNSmasq for multiple load balancers	11/28/2018	WIP	Divyesh	
	LeastConn algorithm	11/28/2018		Divyesh	
Service Registry	Scalable nodes(un/register services)	11/21/2018	Done	Anushree	
	Distributed registry(cached registry)	11/21/2018	Done	Anushree	
	Push updates of registry	11/28/2018	WIP	Anushree	
	Heartbeat sockets	12/03/2018	WIP	Anushree	
Testing & Simulations	Create VM nodes	12/05/2018		Anushree	
	Scripts for setup	12/05/2018		Divyesh	
	Load testing and scripts	12/05/2018		Divyesh	
	Scalability testing and scripts	12/05/2018		Anushree	
	Experiments & Simulation	12/08/2018		Anushree, Divyesh	
Final Report		12/12/2018		Anushree, Divyesh	

8. Future Scope

8.1. Optimizations at SR

Instead of sending the complete data from SR to the LB to cache service registry, we can just send the changes in the registry and update data associated with only specific services.

8.2. Auto scaling nodes and services

We can auto spawn nodes and processes for services if we have too many requests and all our servers are heavily loaded for better performance but incurring some server cost.

8.3. Connection-oriented services

For connection-oriented services, we can hash client's IP address and store it as <key, value> pair, where key = hash and value = service endpoint. This will ensure that a client is always connected to a particular server for connection-oriented services. This can avoid overhead of sending the client specific data or session data to other servers to ensure consistency or using a centralized database to store all the client data which can become a single point of failure.