Load-balance Modeling based on Fuzzy Logic

Ramtin Abbas Khalatbari (94012269022)
Mojtaba Rouhi Bakhsh (94122691003)

Data Structures

Primary logic of project is implemented by functions that modify a data structure so let's start by the Data Structures.
The data structure contains three main entities, Cluster, Server, Client.
On the root of the data structure there is a list of Clusters.
Each Cluster contains a list of Servers.
Each Server contains a list Clients.
Let’s start talking about deeper data structure Client:

Client

```
export interface Client {
 ramUsage: number;
 cpuUsage: number;
 hardUsage: number;
}
```

A client only has three properties: ramUsage, cpuUsage, hardUsage:

- ramUsage: is the amount of ram a client uses from server total ram capacity and it’s stored in megabytes.
- cpuUsage: is the percentage of cpu a client uses from one core of a server.
- hardUsage: is the amount of ram a client uses from server total ram capacity and it’s stored in megabytes.

Server

```
export interface Server {
 ram: number;
 cpu: number;
 name?: string;
 hard: number;
 clients?: Array<Client>;
 ramUsage?: number;
 cpuUsage?: number;
 hardUsage?: number;
 ramUtilization?: number;
 cpuUtilization?: number;
 hardUtilization?: number;
 load?: number;
 fuzzyLoad?: fuzzyLoad;
 fuzzyOutput?: fuzzyOutput;
 status?: status;
 loadMinusAverage?: fuzzyLoad;
}
```

- Ram: Total ram capacity of server.
- Cpu: number core * 100.
- Hard: Total hard capacity of server.
- Clients: A list of clients that is assigned to this server.
- ramUsage: Is calculated by adding ramUsage of clients that are assigned to this server.
- cpuUsage: Is calculated by adding cpuUsage of clients that are assigned to this server.
- hardUsage: Is calculated by adding hardUsage of clients that are assigned to this server.
- ramUtilization, cpuUtilization, hardUtilization: All calculated by function named calServerUtilization.
- Load: Is determined by a function called calcServerLoad,
- fuzzyLoad: An array calculated by fuzzifying load **[very low, low, medium, high, very high].**
- fuzzyOutput: Fuzzy output of controller for each server **[r, n, s]**
- Status: Can take 3 values: -1 | 0 | 1, -1 means: receiver, 0 means: neutral and 1 means: sender.
- loadMinusAverage: Is also an array of [very low, low, medium, high, very high]

Cluster

```
export interface Cluster {
 servers: Array<Server>;
 delay: number;
 name?: string;
 ramCapacity?: number;
 cpuCapacity?: number;
 hardCapacity?: number;
 ramUsage?: number;
 cpuUsage?: number;
 hardUsage?: number;
 ramUtilization?: number;
 cpuUtilization?: number;
 hardUtilization?: number;
 load?: number;
 fuzzyLoad?: fuzzyLoad;
 status?: Array<status>;
 active?: boolean;
 senderIndex?: number;
 receiverIndex?: number;
 senderKey?: string;
 receiverKey?: string;
 sens?: number;
}
```

Cluster is data structure is very similar to Server, so we just talk about the different ones:

- sens: determines the sensitivity of algorithm meaning how much cluster tries to balance it self
- active: If the load balancing finished or not
- delay: stored in milliseconds, determines the delay between submission to the database

The Algorithm, Fuzzy Rules, Fuzzy Sets

The user creates a cluster by specifying sensitivity, delay and a name.
Then adds some servers by specifying server name, number of cores, ram capacity, hard capacity.
Then the users adds some clients to servers.
Now algorithm can start balancing clients between servers.
Let's explore the flow of algorithm:

We start by looping through servers and calculating utilization of each resource

```
export function calServerUtilization(server: Server): Server {
 let totalRamUsage: number = 0;
 let totalCpuUsage: number = 0;
 let totalHardUsage: number = 0;
 if (server.hasOwnProperty("clients")) {
   for (let key in server.clients) {
     totalRamUsage = totalRamUsage + server.clients[key].ramUsage;
     totalCpuUsage = totalCpuUsage + server.clients[key].cpuUsage;
     totalHardUsage = totalHardUsage + server.clients[key].hardUsage;
   }
 } else {
   server.ramUtilization = 0;
   server.cpuUtilization = 0;
   server.hardUtilization = 0;
   return server;
 }
 server.ramUsage = totalRamUsage;
 server.cpuUsage = totalCpuUsage;
 server.hardUsage = totalHardUsage;
 server.ramUtilization = (totalRamUsage / server.ram) * 100;
 server.cpuUtilization = (totalCpuUsage / server.cpu) * 100;
 server.hardUtilization = (totalHardUsage / server.hard) * 100;
 return server;
}
```

The function calServerUtilization loops through clients of the server and adds them to calculate the total amount of each resources. Then it divides the amount over total capacity of each resource to get the utilization value (percentage).
Then we calculate the load for each server:

```
export function calServerLoad(server: Server): Server {
 if (server.hardUtilization >= 100) {
   server.load = 100;
 } else {
   server.load = Math.max(server.cpuUtilization, server.ramUtilization);
 }
 return server;
}
```

In load balancing we mostly care about ram & cpu and not much about hard, so we get max of ram & cpu utilization for server load unless hard utilization is 100% in which we set the load 100%.
Now we fuzzify our load numbers so we can use them in our fuzzy controller.

Our fuzzy set includes linguistics hedges: **very low, low, medium, high and very high**. We specify them with the following formulas:

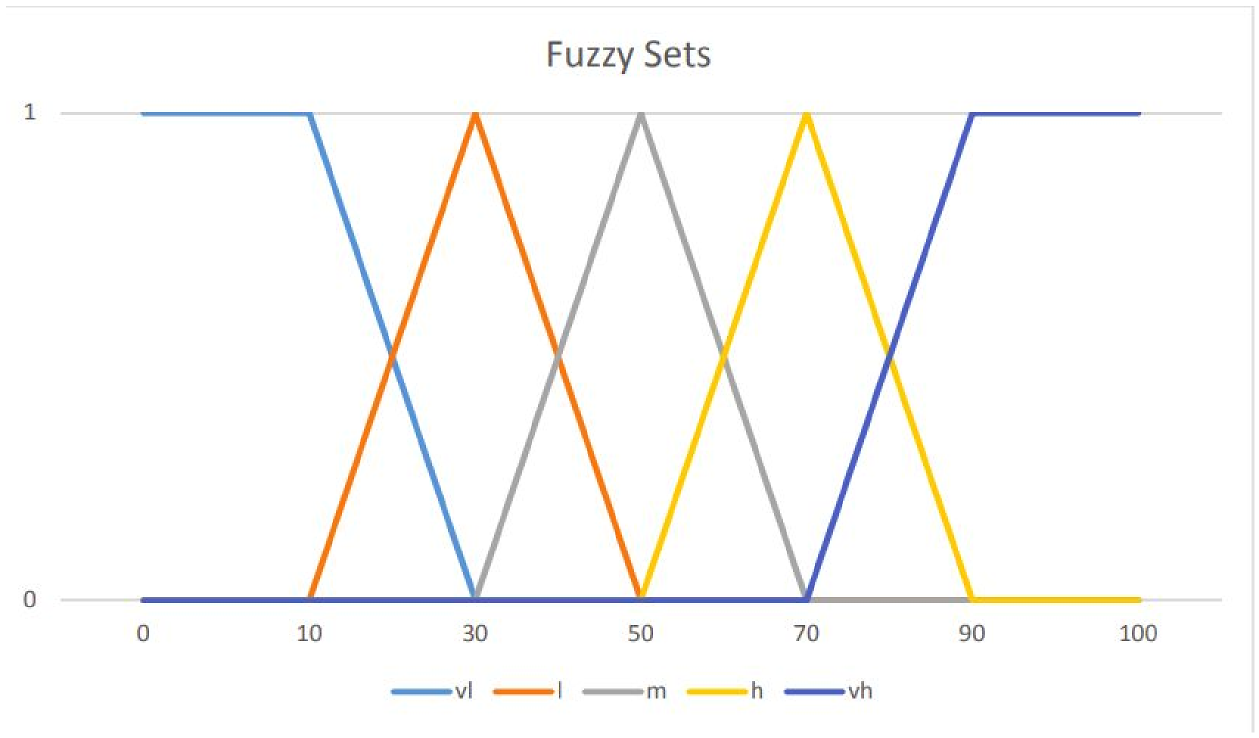
$$vl \begin{cases} 1 & Load \le 10 \\ 0 & Load < 30 \\ \frac{30-Load}{20} & 10 < Load \le 30 \end{cases}$$

$$l \begin{cases} 0 & Load \le 10 \text{ or } Load \ge 50 \\ \frac{Load-10}{20} & 10 < Load \le 30 \\ \frac{50-Load}{20} & 30 < Load < 50 \end{cases}$$

$$m \begin{cases} 0 & Load \le 30 \text{ or } Load \ge 70 \\ \frac{Load-30}{20} & 30 < Load \le 50 \\ \frac{70-Load}{20} & 50 < Load < 70 \end{cases}$$

$$h \begin{cases} 0 & Load \le 30 \text{ or } Load \ge 70 \\ \frac{Load-50}{20} & 50 < Load \le 70 \\ \frac{90-Load}{20} & 70 < Load < 90 \end{cases}$$

$$vh \begin{cases} 1 & Load \ge 90 \\ 0 & Load \le 70 \\ \frac{30-Load}{20} & 70 < Load \le 90 \end{cases}$$



We wrote a function based on above rules called calcServerFuzzyLoad:

```
export function calcServerFuzzyLoad(server: Server): Server {
 let load = server.load;
 let veryLow = 0;
 let low = 0;
 let medium = 0;
 let high = 0;
 let veryHigh = 0;
 if (load <= 10) {
   veryLow = 1;
 } else if (load > 10 && load <= 30) {
   veryLow = (30 - load) / 20;
   low = (load - 10) / 20;
 } else if (load > 30 && load <= 50) {
   low = (50 - load) / 20;
   medium = (load - 30) / 20;
 } else if (load > 50 && load <= 70) {
   medium = (70 - load) / 20;
   high = (load - 50) / 20;
 } else if (load > 70 && load <= 90) {
   high = (90 - load) / 20;
```

```
    veryHigh = (load - 70) / 20;
  } else if (load > 90) {
    veryHigh = 1;
  }
  server.fuzzyLoad = [veryLow, low, medium, high, veryHigh];
  return server;
}
```

After calculating the required properite for our servers we can calculate the properties for our Clusters.
We start by calculating the capacity and usage of resources based on the servers that are inside the cluster. We use a function named calcClusterCapacityAndUsage:

```
export function calcClusterCapacityAndUsage(cluster: Cluster): Cluster {
  let servers = cluster.servers;

  let totalRamUsage = 0;
  let totalCpuUsage = 0;
  let totalHardUsage = 0;

  let totalRamCapacity = 0;
  let totalCpuCapacity = 0;
  let totalHardCapacity = 0;

  for (let key in servers) {
    totalRamCapacity = totalRamCapacity + servers[key].ram;
    totalCpuCapacity = totalCpuCapacity + servers[key].cpu;
    totalHardCapacity = totalHardCapacity + servers[key].hard;

    totalRamUsage = totalRamUsage + servers[key].ramUsage;
    totalCpuUsage = totalCpuUsage + servers[key].cpuUsage;
    totalHardUsage = totalHardUsage + servers[key].hardUsage;
  }
  cluster.ramCapacity = totalRamCapacity;
  cluster.cpuCapacity = totalCpuCapacity;
  cluster.hardCapacity = totalHardCapacity;

  cluster.ramUsage = totalRamUsage;
```

```
 cluster.cpuUsage = totalCpuUsage;
 cluster.hardUsage = totalHardUsage;
 return cluster;
}
```

This function loops through servers inside of a cluster and adds the values of capacity and usage and set result on properties of cluster.
Now that we have capacity and usage we can calculate the utilization of resources we did that by writing a function named calcClusterUtilization:

```
export function calcClusterUtilization(cluster: Cluster): Cluster {
 cluster.ramUtilization = (cluster.ramUsage / cluster.ramCapacity) * 100;
 cluster.cpuUtilization = (cluster.cpuUsage / cluster.cpuCapacity) * 100;
 cluster.hardUtilization = (cluster.hardUsage / cluster.hardCapacity) *
100;
 return cluster;
}
```

This function just divides the usage value on capacity values and stores them as percentage values.
Then we calculate the load of the server based on utilization values using a function named calcClusterLoad:

```
export function calcClusterLoad(cluster: Cluster): Cluster {
 if (cluster.hardUtilization >= 100) {
   cluster.load = 100;
 } else {
   cluster.load = Math.max(cluster.cpuUtilization,
cluster.ramUtilization);
 }
 return cluster;
}
```

This function is exactly like the calcServerLoad, the only difference is that it operates on a cluster.
Now we have the load of the server, this is basically the average load of the cluster. We fuzzify this value so our fuzzy controller can used it.
We use function called fuzzifyCluster:

```
export function fuzzifyCluster(cluster: Cluster): Cluster {
 let load = cluster.load;
 let veryLow = 0;
 let low = 0;
 let medium = 0;
 let high = 0;
 let veryHigh = 0;

 if (load <= 10) {
   veryLow = 1;
 } else if (load > 10 && load <= 30) {
   veryLow = (30 - load) / 20;
   low = (load - 10) / 20;
 } else if (load > 30 && load <= 50) {
   low = (50 - load) / 20;
   medium = (load - 30) / 20;
 } else if (load > 50 && load <= 70) {
   medium = (70 - load) / 20;
   high = (load - 50) / 20;
 } else if (load > 70 && load <= 90) {
   high = (90 - load) / 20;
   veryHigh = (load - 70) / 20;
 } else if (load > 90) {
   veryHigh = 1;
 }

 cluster.fuzzyLoad = [veryLow, low, medium, high, veryHigh];

 return cluster;
}
```

This function is also very similar to calcServerFuzzyLoad.
The reason we want the fuzzy values of our cluster is that we can use this values for comparing to each server fuzzy load. If server is more loaded than the average of the cluster then it needs to send some of it’s clients to other servers and if server is less loaded than average of cluster then it will receive some clients from other servers to balance the load of the cluster between servers.
So we need to define some rules to make this comparison between fuzzy load of each server and average fuzzy load of the cluster.

We need to produce some fuzzy outputs that helps us determine which server needs to be receiver, which server should not do anything and which server needs to send servers to receivers.
Our fuzzy output for out fuzzy controller uses three linguistics hedges [r, n, s] this values determine the willingness of a server to be receiver, neutral, or sender.
We produce our fuzzy output by using three functions named calcFuzzyOutput2 (Second approach), loadIsMore and loadIsLess
:

```
export function calcFuzzyOutput2(cluster: Cluster): Cluster {
 let average = cluster.fuzzyLoad;
 let servers = cluster.servers;
 for (let key in servers) {
   let load = servers[key].fuzzyLoad;
   let loadMinusAverage : fuzzyLoad = [0, 0, 0, 0, 0];
   for (let i = 0; i < load.length; i++) {
     loadMinusAverage[i] = load[i] - average[i];
   }
   servers[key].loadMinusAverage = loadMinusAverage;
 }
 for (let key in servers) {
   for (let i = 0; i < servers[key].loadMinusAverage.length; i++) {

     if (servers[key].loadMinusAverage[i] < 0) {
       servers[key].fuzzyOutput =
loadIsMore(servers[key].loadMinusAverage);
       break;
     } else if (servers[key].loadMinusAverage[i] > 0) {
       servers[key].fuzzyOutput =
loadIsLess(servers[key].loadMinusAverage);
       break;
     }
   }
 }
 return cluster;
}
function loadIsMore(input): fuzzyOutput {
 let s = 0;
 for(let i = 0; i < input.length; i++) {
   if(input[i] > 0) {
     s = s + input[i]
   }
 }
 return [0, 1 - s, s]
}
function loadIsLess(input): fuzzyOutput {
 let r = 0;
 for(let i = 0; i < input.length; i++) {
   if(input[i] > 0) {
     r = r + input[i]
   }
 }
 return [r, 1 - r, 0]
}
```

First we calculate the difference between load of each server and average load. To produce the appropriate fuzzy output the rules are: if the load is explicitly higher than average load. The server wants to be receiver. If the load is explicitly lower than the average the server wants to be sender,
If the load is higher than the average but not by that much, it will want to be neutral or receiver. If the load is higher than the average but not by that much, it will want to be neutral or sender. The willingness to be each of theses is determined by fuzzy load of the server and the cluster.
After calculating the fuzzy output of the controller we need to defuzzify the values so we can use them in our application. We wrote a function named calcOutput for this purpose:

```
export function calcOutput(cluster: Cluster): Cluster {
 cluster.active = false;
 //   cluster.status = Array(cluster.servers.length).fill(0);
 let servers = cluster.servers;
 let maxSender = 0;
 let maxReceiver = 0;
 let senderIndex = 0;
 let receiverIndex = 0;

 let senderKey = "";
 let receiverKey = "";
```

```javascript
for (let key in servers) {
  servers[key].status = 0;
  if (servers[key].hasOwnProperty("fuzzyOutput")) {
    if (servers[key].fuzzyOutput[0] > maxReceiver) {
      maxReceiver = servers[key].fuzzyOutput[0];
      receiverKey = key;
    }
    if (servers[key].fuzzyOutput[2] > maxSender) {
      maxSender = servers[key].fuzzyOutput[2];
      senderKey = key;
    }
  } else {
    break;
  }
}

if (
  maxSender < 1 - cluster.sens &&
  maxReceiver < 1 - cluster.sens
) {
  return cluster;
} else {
  if (typeof servers[senderKey] === 'undefined' || servers[receiverKey]
=== 'undefined') {
    return cluster;
  } else {
    servers[senderKey].status = 1;
    servers[receiverKey].status = -1;
    cluster.senderKey = senderKey;
    cluster.receiverKey = receiverKey;
    cluster.active = true;
  }
  cluster.servers = servers;
  return cluster;
}
```

```
}
```

This function determines which server should be the sender and which server needs to be the receiver and also if it’s even necessary to do load-balancing at this time.

This function compare the willingness of each server to be a sender and each server to be a receiver. The server with most willingness to be sender will be chosen as the sender and the server with most willingness to be receiver will be the receiver.

We saw properly called sens in our Cluster data structure. We use this value here. This value is between 0 and 1. The higher means that we do more load-balancing or means that even if the willingness of the server to be sender or receiver is low we move the clients. We compare the 1 - value of this property to highest willingness of the server to be receiver or a sender. If the willingness is not high enough compared to sense we will not choose a sender and receiver and we stop the load-balancing.