Loadbalancer Projekt

Steffen Reimann August 2022

Loadbalancer Bestandteile

Loadbalancer Server mit Konfiguration Server

Der Loadbalancer kann mit entsprechend vielen execute-servern konfiguriert werden. Dabei wird der Loadbalancer nach Möglichkeit immer 2 Webserver online lassen und alle weiteren runterfahren. Wobei nur der Webserver offline genommen wird, der execute-server läuft weiter damit er befehle vom Loadbalancer entgegennehmen kann. Falls die anfragen, einen Schwellwert überschreiten werden weitere Webserver über die execute-server gestartet. Das Ganze überwacht der Loadbalancer und beeinflusst, wann welche gestartet werden.

Execute Server

Der Execute Server nimmt befehle vom Loadbalancer zum Starten und Stoppen des Web Servers entgegen und führt sie aus.

Webserver

Einfacher NodeJS Web Server der einfach nur eine anfrage beantworten kann. Dieser wird von execute-server gestartet und überwacht.

Anforderungen

Der Loadbalancer muss eingehende anfragen annehmen können.

Der Loadbalancer muss Anfragen an einen Server stellen können.

Der Loadbalancer muss entscheiden können an welchen Server er die anfrage stellt.

Der Loadbalancer muss Server Starten und Stoppen können.

Arbeitsschritte

Einrichtung der Programmierumgebung

Voraussetzungen git, vscode, nodejs, npm

Einrichtung eines git reposetory und Ordnerstrucktur für die verschiedenen Server und tools.

- mys-projekt-load-balancer-auto-scaler
  - Exec-server
  - Main-server
  - Test-client
  - Web-server

Schritt 1

Recherche welche NodeJS internen Module und externen Module benötigt werden.

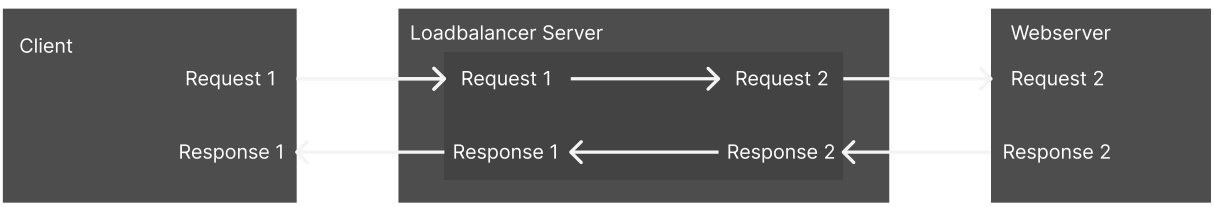
Benötigte Module:

- Intern
  - Cluster – benötigt um Mehrere Threads starten zu können
  - http
  - os
  - fs – Filesystem Module
  - url
- Extern
  - Express – Webserver Module
  - Socketio – übernimmt die Kommunikation zwischen Servern
  - easy-nodejs-app-settings – Speichert Config daten auf der Festplatte

Schritt 2

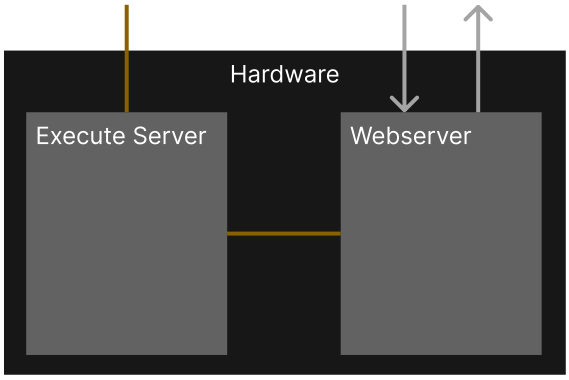
NodeJS Script schreiben welches Anfragen von einem Client annehmen kann und Anfragen an einen Server Stellen kann schreiben.

Diese Funktionalitäten richtig in einer neuen Funktion verbinden.



Schritt 3

Starten und Stoppen eines NodeJS Webservers. Diese Funktionen muss der Loadbalancer bieten. Diese Sollten ursprünglich per SSH direkt von dem Loadbalancers ausgeführt werden. Dafür habe ich entsprechende SSH Keys ausgetauscht. Letztendlich habe ich mich doch dafür entschieden das es noch eine extra Instanz (exec-server) gibt, welche den Webserver direkt startet und stoppt.



Schritt 4

Weboberfläche mit welche der Loadbalancer konfiguriert werden kann.

Das beinhaltet die Funktionalität einen Execute Server mit IP, Port und Webserver Port hinzufügen zu können und entsprechende Informationen über diesen Server zu erhalten.

Schritt 5

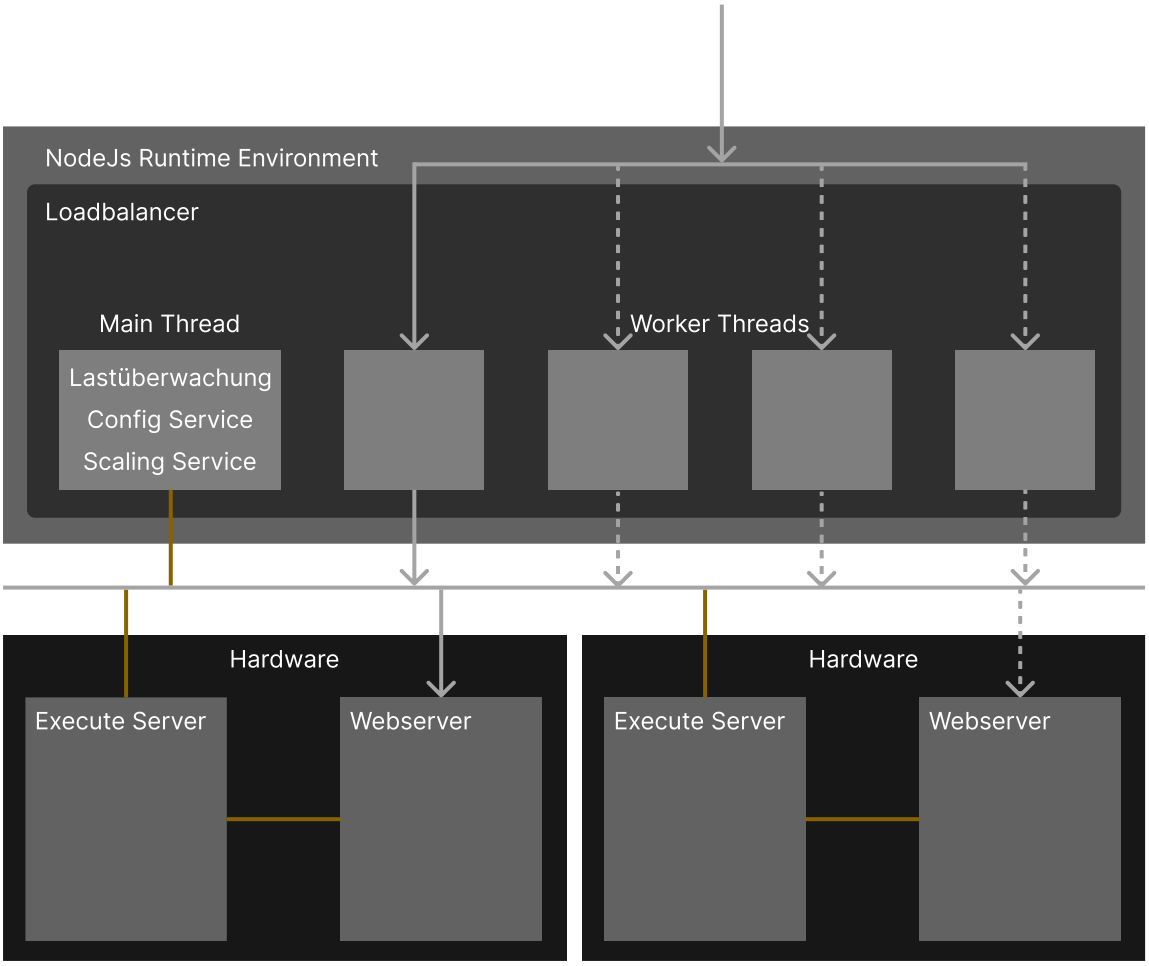
Loadbalancer und Autoscaler Funktionen implementieren.

Diese Funktionalitäten entscheiden darüber welche Webserver angesprochen, gestartet oder gestoppt werden.

Schritt 6

Loadbalancer für Multithreading optimieren.

Architektur



Installation

Requirments

NodeJS

Git

Install Loadbalancer

Clone Repositery from github

```
git clone git@github.com:steffenreimann/mys-projekt-load-balancer-auto-scaler.git
```

Go into Dir

```
cd /mys-projekt-load-balancer-auto-scaler/main-server
```

Install node modules

```
npm install
```

Run Loadbalancer

```
node main-server.js
```

Install Execute Server

Go into Dir

```
cd /mys-projekt-load-balancer-auto-scaler/exec-server
```

Install node modules

```
npm install
```

Run Execute Server

```
node index.js -execp 7070 -webp 6060
```

Install Web Server

Go into Dir

```
cd /mys-projekt-load-balancer-auto-scaler/web-server
```

Install node modules

```
npm install
```

Code Dokumentation

Klassen

Das Programm enthält zwei Klassen. Einmal den SecondaryServerManager und SecondaryServer. Der SecondaryServerManager hält unteranderem alle SecondaryServer Objekte.

```
class SecondaryServer {
    host – Speichert die IP Adresse des Execute Servers
    port – Speichert den Port des Execute Servers
    webserver_port – Speichert den Port des Web Servers
    socket – Speichert das Socket Objekt von SocketIO
    connections – Wird bei jeder neuen Verbindung hochgezählt und jede Sekunde
zurückgesetzt
    lastReqPerSec - Speichert wie viele Anfragen letzte Sekunde an diesen Web-Server
gestellt wurden
    status – Speichert ein Status Objekt von exec-server und web-server

    connect() {} – verbindet main-server mit dem exec-server per socketio
    calcReqPerSec() {} – berechnet wie viele Anfragen in der letzten Sekunde an diesen
Server gestellt wurden
    changehost(ip, exec_port, webserver_port) {} – Ändert die Adresse und Ports des
exec-serves
    start() {} – Startet den Webserver
    stop() {} – Stoppt den Webserver
    getStatus() {} – Gibt den Status des exec-servers und web-server zurück
}
```

```
class SecondaryServerManager {
    secServers – Speichert alle SecondaryServer Objekte
    connections – Speichert wie viele Anfragen letzte Sekunde an diesen Loadbalancer
gestellt wurden
    runningServers – Speichert welche Webserver laufen
    notRunningServers – Speichert welche Webserver nicht laufen aber gestartet werden
können
    bestServer – Speichert welcher derzeit die geringste Auslastung hat

    getReqPerSek(cb) {} gibt Objekt mit allen web-server Anfragen pro Sekunde zurück
    addServer(id, secServer) {} – fügt eien exec-server hinzu
    removeServerById(id) {} – löscht einen exec-server
    getServerById(id) {} – gibt einen exec-server zurück
    async getBestServer() {} – gibt derzeit besten web-server zurück
    async getServerStatus() {} – gibt exec-server status zurück
    getRunningServers() {} – gibt laufende web-server zurück
    getNotRunningServer() {} – gibt nicht laufende web-server zurück
    async startNextServer() {} – startet nächst besten web-server
    getServerWithLowestReqPerSec() {} – gibt web-server mit der geringsten last zurück
    calcAllReqPerSec() {} – Berechnet von allen Threads zusammen die Anfragen pro
Sekunde
}
```

Initialisierung des Loadbalancers

Die Funktion init() initalisiert als allererstes die configurations datei mit in welche alle execute server gespeichert sind. Diese werden danach in der funktion initServers() zu Klassen Objekten von SecondaryServer gemacht und dem secondaryServerManager übergeben um die eine Referenz zu speichern. Als letztes wird noch die Methode connect des SecondaryServer Klassen Objekt aufgerufen.

```
async function init() {

    config = new fm.File({
        appname: 'mys-loadbalancer-main-server',
        file: 'DataStore.json',
        data: {
            port: 8080,
            secondaryServer: {}
        }
    })
    await config.init()
    initServers();
    startService();
}

async function initServers() {
    var keyArray = Object.keys(config.data.secondaryServer);
    for (let index = 0; index < keyArray.length; index++) {
        const key = keyArray[index];
        const element = config.data.secondaryServer[key];
        const secs = new SecondaryServer(element)
        secondaryServerManager.addServer(key, secs);
        secs.connect();
    }
}
```

Start Multithreading Services

Wenn die Initalisierung der SecondaryServer und secondaryServerManager abgeschlossen ist wird die funktion startService() aufgerufen. Damit werden die Worker Threads und der Main Thread gestartet. Code Beispiel wie der Main Thread und die Wocker Threads gestartet werden.

```
function startService() {
    if (cluster.isMaster) {
        //Alles was im Master gemcht werden soll hier rein.
        for (var i = 0; i < threads; i++) {
            cluster.fork();    //Starten eines Workers
        }

        cluster.on('exit', (worker, code, signal) => {
            //Wenn ein Worker beendet wird
            if (signal) {
                console.log(`worker was killed by signal: ${signal}`);
            } else if (code !== 0) {
                console.log(`worker exited with error code: ${code}`);
            } else {
                console.log('worker success!');
            }
        });

        cluster.on('fork', (worker) => {
            //Wenn ein Worker erfolgreich gestartet wird und verbunden ist.
            worker.on('message', (msg, cb) => {
                console.log(msg);
            })
        })
    } else {
       //Alles was nur im Worker Thread ausgeführt werden soll hier rein.
    }
}
```

Test

Um den Loadbalancer zu testen, gibt es das test-client Tool. Dieses Tool kann ebenfalls mit mehreren Threads so viele Anfragen hintereinander stellen wie man möchte.

Meinung

Meine Idee einen Loadbalancer mit JavaScript mit NodeJS zu Programmieren hat sich als nicht so sinnvoll herausgestellt. Gar nicht wegen der Sprache an sich, sondern weil das JavaScript Runtime Environment (NodeJS) dort an sei IO Limit kommt. Was ich messen konnte und auch nach Recherchen liegt das Limit ca bei 700 Anfragen pro Sekunde pro thread. Also konnte ich diese noch merklich erhöhen als ich mein Programm auf multithreding umgebaut habe. Allerding auch dann habe ich nicht mehr als 2200 Anfragen pro Sekunde erreichen können. Wenn man das zb. mit nginx vergleicht ist nginx je nach dem ca. 30x schneller. Was mich zudem Schluss bringt, NodeJS für einen Loadbalancer einzusetzen mach im Hinblick auf die Performance keinen Sinn.