EECS6432: Containerized Orchestration of IoT Lighting Systems

Daniel Parreira
York University
Toronto ON, Canada

Pedro Casas
York University
Toronto ON, Canada

Shane Segal
York University
Toronto ON, Canada

Abstract

We have implemented a dockerized, automatically load-balancing IoT system and associated web-infrastructure for a networked smart-lighting system. Our system can be applied to public spaces with shared resources, potentially containing individual actors with competing goals - all while working to balance a global setpoint that best satisfies the goals of all independent actors in the system.

Created with horizontal scalability in mind, our software architecture is well-suited towards rapid expansion of both users and resources. Our embedded IoT devices are connected to our GraphQL API through a Kafka pub-sub system, ensuring easy scalability. Our various interaction methods (voice, web app, EEG Reader) connect through our Pub-Sub architecture where appropriate, which connects them to our orchestrated containerized server architecture. Our PID based load-balancer ensures that there are enough server resources available and that the requests coming from each available endpoint are satisfied as much as possible in the face of possibly competing goals.

1. Introduction

Public spaces with adaptive smart technologies present interesting problems regarding the satisfiability of concurrent requests for shared resources. We explore this problem in the context of a smart lighting system in a space such as a university library or public co-working space. Prior to the introduction of smart technologies, users might be limited to working in a globally lit environment, or a locally lit environment that did not effectively consider the preferences of those around them, or of the energy usage policy of the building or organization.

Our system enables authenticated and authorized users to make luminance-level requests about the local lighting level. Our system balances this request with the requests of spatially co-located individuals who have made prior requests. These requests can either be trivially compatible with each other (e.g co-located users making similar requests) or be incompatible (brightness preferences outside of 10-15%).

As IoT technology has only recently emerged as a powerful force in the technology space, there are many situations that are in a suboptimal state. We attempt to address one such situation through the application architecture presented here. Our solution works to combine the global needs of a building such as the overally energy usage or brightness level, as well as local needs of a user such as the local brightness level being optimal for the situation at hand. Local optimality for the system at hand can include such things as brightness for the time of day, color of the light produced. or appropriate brightness for the task. In addition, we also focus on accessible and innovative models of user interaction with the goal of reducing the barrier for use of our system by people of differing physical abilities.

The research challenges we faced include several disparate factors. The load balancing of our servers through adaptive software is an open problem in software engineering and one that we apply very modern techniques in software engineering in order to deal with it. We then have the challenge of balancing user-local preferences with potentially contradictory preferences of other co-located users as well as with global preferences that can be set on a location-wide basis. And finally, we have combined new and experimental human computer interaction techniques into an integrated technological solution.

2. Models, Architecture, and Algorithms

2.1. Models

When considering the models in our software system, it is useful to seperate them into the server software and IoT software. The main model in our IoT software is that of the User. We require our users to be authenticated with a verified email and password, in order to create a persistent account for them that can save preferences and locations, as well as a history of commands in order to potentially build a model of their behaviour in a future iteration of this project.

Due to the complex nature of our user interaction methods, we model our commands as a three-way relation between devices, the issuing user, the issuing device type, and the action that the command will instigate in the lighting system. This is fed as input into our PID Controller. The issuing device can be one of: a web interface, an EEG brain wave reader, or a voice command. Voice commands are parsed and translated using Google's Cloud Natural Language API, and the EEG Commands are analyzed and parsed using data mining algorithms based on *INSERT EEG INFO HERE*.

2.2. Architecture

Our server architecture is composed of a set of containers that each run an instance of our server software, including GraphQL endpoints and our Kafka pub-sub infrastructure. Our infrastructure is composed of a MYSQL-backed GraphQL API, a MYSQL database for our EEG infrastructure, an internal API for our EEG brainwave reader, and an Apache Kafka Server to create and manage input from our EEG devices. The distributed nature of our application is managed by Apache Zookeeper, which allows us to maintain a performant and highly distributed application architecture.

Our Philips Hue Bulbs are connected to Raspberry Pi's, which dispatch commands to our GraphQL servers. These servers accept incoming commands from processes and balance the requests from the various users through our custom PID-based load-balancer, and then redistribute the balanced commands back out to the corresponding devices.

In order to read in voice commands, we enable integration with the Google Natural Language API, which parses the sound input and returns one of the specified commands in plaintext. We then issue this command to our load balancing server.

Commands are issued from our load balancing server to the devices they were targeted towards in order of priority as determined by the load balancer. This avoids the problem of commands being issued at the same time, as we simply allow our load-balancer to determine the order based on order of arrival, and take that as our source of truth instead of possibly disparate device-specific timing logs.

Kafka is used to handle the incoming data from the EEG device. When EEG data is received from a user, it is sent from the receiving GraphQL endpoint to the first Kafka partition and saved in a database to be processed by a Kafka Consumer thread, which classifies theand pushes it back to a different Kafka queue specified by the Users authenticated identity. After all EEG input data is received and parsed, the GraphQL API returns the most recent EEG classification belonging to that user.

The load-balancing in our application is two-tiered. The first tier is the load-balancing of user requests, and the second tier is the load balancing of the various servers running in our application. This allows us to balance the requests of multiple users easily and efficiently, as well as providing an easy method for implementing global constraints, in the form of a target setpoint. For more detail, refer to Algorithms 2.3.

Our application is scalable horizontally by simply scaling the number of containers runnning through the Docker Swarm API. We have simple performance guarantees provided by the domain in which our platform exists. Namely, we know that when we can handle the maximum physical capacity of the location, our scaling needs have been met for that location. As such, costs associated with autoscaling are capped at a known value.

2.3. Algorithm