Communication between webserver and embedded server

Team: Launchpad
Spring 2023
CMPT 433
Member:Yiwen Wang, Connor Read, Jason Nguyen, Serena Bal-Pietrantoni

Basic logic:
The configuration of the launchpad is sent from Reactjs app to Nodejs webserver in the form of HTTP request. Each HTTP request will be saved into a file on disk of BeagleBone. The file representing a request will be picked up by the C++ embedded server and processed. After the request is processed, the corresponding file will be deleted by the C++ server. This design is essentially a local version of cloud message queue based system.

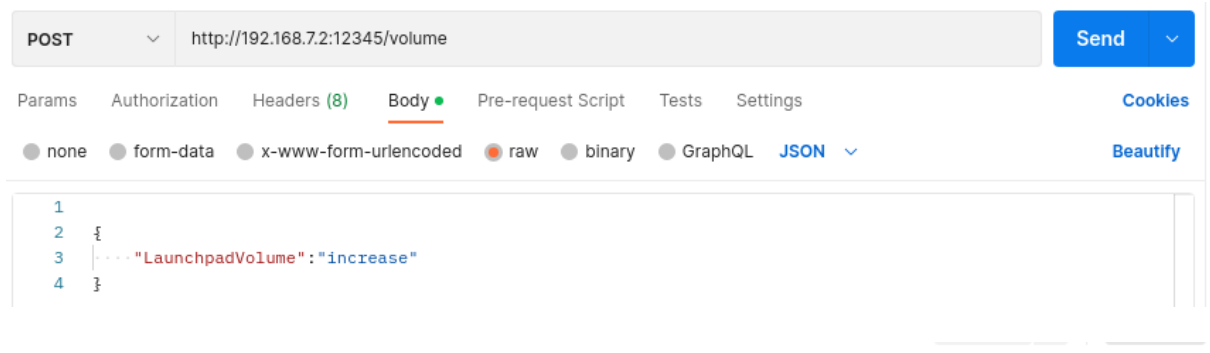
Advantage:

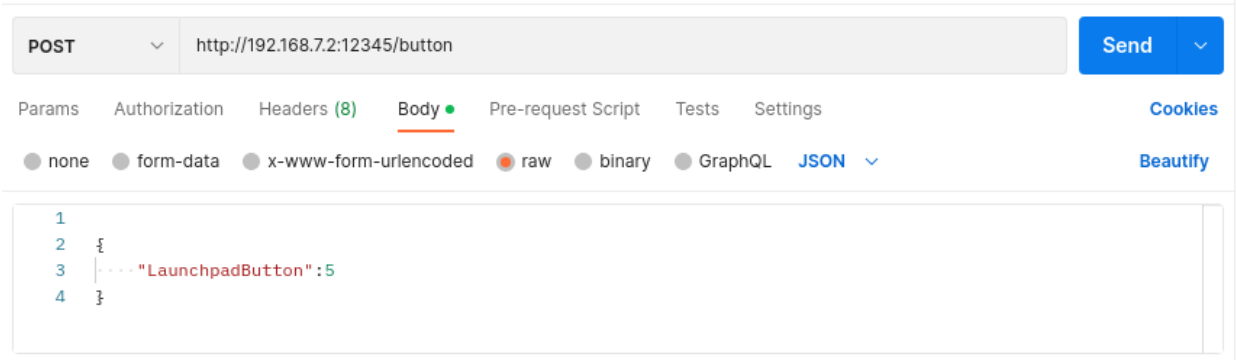
- The Nodejs server (i.e., the requestor) and C++ server (i.e., the processor) have no dependency on each other as they are not even aware or each other.
- Each request that cannot be processed for some reason can be retried by the C++ server itself without needing the Nodejs sever to resend the request.
- Only requires one thread for C++ server to handle all the requests.

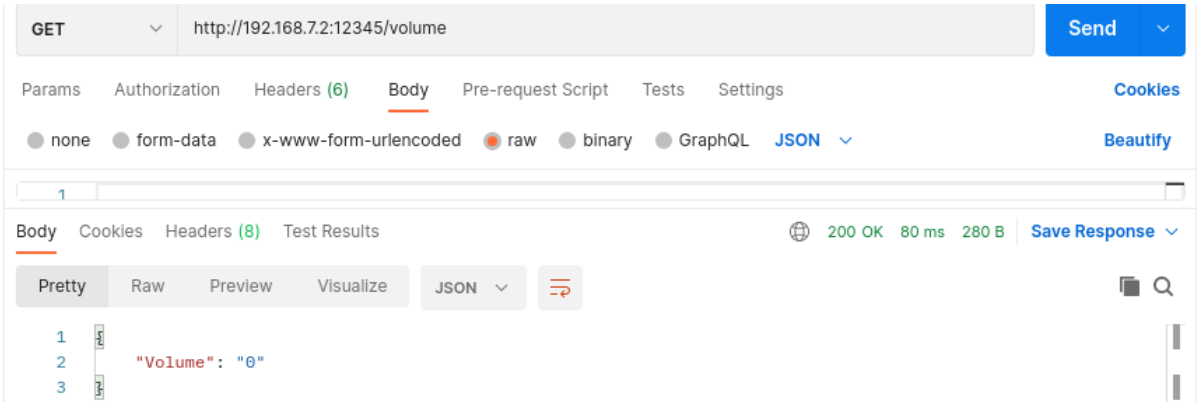
Disadvantage:

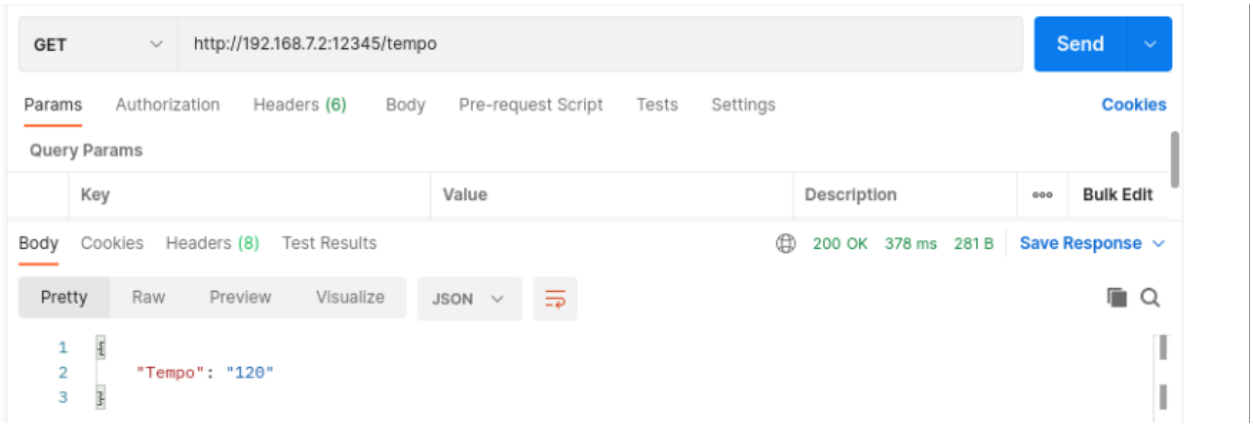
- None. Although saving and reading file is slower compared to UDP based socket transmission, the slowness can be omitted because both Nodejs and C++ servers are running on the same machine.

Here are some examples on how to communicate with the C++ server by sending HTTP requests to the Nodejs server via Postman (2 post requests and 2 get requests):









Trouble-shooting:

Uing ‘console.log()’ to print out the received message in the terminal.

The output looks like this:

```
debian@BeagleBone:/mnt/remote/myApps/ApiServer$ node app.js
App listening on port 12345!
request body: { LaunchpadButton: 5 }
button: 5
```

Step-by-step code guide:

In Nodejs (post request):

The code in the black box is receiving requests from the Reactjs app. It sends an HTTP 400 Bad Request response with an error message.

The red box generates a random file name and a file path is constructed and the message is written into the file.

If the file write is successful, an HTTP 200 OK response is sent.

volume.js 1.41 KB Edit

```
const express = require('express');
const crypto = require('crypto');
const router = express.Router();
const fs = require('fs');

router.post('/', (request, response) => {
    console.log("request body:", request.body);
    const volumeStr = request.body['LaunchpadVolume'];

    if (!volumeStr) {
        response.status(400).json({ "Error": `Invalid input "${volumeStr}"` });
        return;
    }

    console.log("volume: " + volumeStr);

    let volumeSetting = "Volume\n" + volumeStr;
    let uuid = crypto.randomUUID();
    let filePath = "/tmp/changeFeed/" + uuid.toString() + ".txt";
    fs.writeFile(filePath, volumeSetting, err => {
        if (err) {
            console.error(err);
            let errJson = {
                "Error": `Unable to save volume due to error ${err}`
            };
            response.status(err.status).json(errJson);
        }

        response.sendStatus(200);
    });
});
```

In Nodejs (get request):

In black box:

Read the contents of a file located at a certain location.

If there is an error reading the file, an error response is sent with a JSON object containing the error message.

In red box:

A JSON object is created containing the data read from the file.

An HTTP 200 OK response is sent with the JSON object.

```
router.get('/', (request, response) => {
    fs.readFile("/tmp/launchpad_volume/value.txt", 'utf8', function (err, data) {
        if (err) {
            console.error(err);
            let errJson = {
                "Error": `Unable to save volume due to error ${err}`
            };
            response.status(err.status).json(errJson);
        }
        console.log(data);
        let volumeJson = {
            "Volume": data
        };
        response.status(200).json(volumeJson);
    });
});
```

C++ code:

The function compares the last modified timestamps of the two files.

```
bool compareFilesByTimestamp(const std::string &filePath1, const std::string &filePath2)
{
    return std::filesystem::last_write_time(filePath1) < std::filesystem::last_write_time(filePath2);
}
```

The sorting is done based on the last modified timestamps of the files using the compareFilesByTimestamp function

```
std::vector<std::string> sortFilesByTimestamp(const std::vector<std::string> &fileList)
{
    std::vector<std::string> sortedFileList = fileList;
    std::sort(sortedFileList.begin(), sortedFileList.end(), compareFilesByTimestamp);
    return sortedFileList;
}
```

The function returns a sorted vector of file paths inside the directory in ascending order of their last modified timestamps.

```
std::vector<std::string> getChangesInAscOrder(DIR *dir, const char* dirPath)
{
    std::vector<std::string> fileList;

    if (dir == nullptr)
    {
        std::cout << "Failed to open directory\n";
        return fileList;
    }

    dirent* entry;
    while ((entry = readdir(dir)) != nullptr)
    {
        if (entry->d_type == DT_REG && entry->d_name[0] != '.')
        {
            fileList.push_back(std::string(dirPath) + "/" + entry->d_name);
        }
    }

    return sortFilesByTimestamp(fileList);
}
```

Finally, we can work with the JSON object which is stored in the file as the following code.

```
auto lines = readLinesFromFile(filePath);
if (lines.empty())
{
    //      std::cout<< "line is enpty"<<std::endl;
    return;
}

auto cmd = lines[0];
auto cmd_value = lines[1];
if (cmd == "Volume")
{
    if (cmd_value == "increase")
    {
        volume_increase_command();
    }
```