smaran jawalkar · Follow
Last edited by smaran jawalkar on Mar 25, 2025
Contributed by

CC Project-5: Raft3D - 3D Prints Management using Raft Consensus Algorithm

Introduction

- Raft3D is one of projects as a part of the Cloud Computing (UE22CSxxxx) at PES University
- The project requires development of backend API endpoints for a Distributed 3D printer management system with data persistence implemented via the **Raft Consensus Algorithm** rather than traditional centralized databases
- Data consistency across the Raft3D network must be maintained through the Raft Consensus mechanism
- Demonstration requires a minimum of three operational nodes (implementable as separate terminals or containers)
- Project demonstration must effectively showcase core Raft functionality including Leader Election and Data Persistence during node failure
- **Successful implementation of Raft Consensus across minimal endpoints takes precedence over comprehensive API development**
- **Projects demonstrating proper Raft fault tolerance and leader election will receive higher evaluation scores than those with complete API implementations but inadequate Raft functionality**
- Development priorities should be established accordingly, with primary focus on correct Raft implementation

Technologies/Languages to be used

- You are free to use any language such as Python, Go, Java, among others.

- Ensure that the chosen language supports all the required functionality.

- *Implementation requires a Raft library that necessitates manual development of the Raft Finite State Machine (FSM) State Transition Function, such as openraft or hashicorp/raft. Higher-level abstraction libraries such as PyDbSync are expressly prohibited for this implementation.*

- My personal recommendation is to use hashicorp/raft which uses Go as the language

- Given below is a list of libraries that you are permitted to use grouped by language and arranged in a loose order based on my level of recommendation for each:
  - Golang:
    - hashicorp/raft
    - dragonboat
    - etcd-io/raft
  - Rust:
    - openraft
    - async-raft
  - C++:
    - ebay/NuRaft
    - baidu/braft*
  - Java:
    - SOFAJRaft* & its User Guide
    - Apache Ratis
    - raft-java*
    - jgroups-raft & its documentation
    - xraft and how too add a State Machine
  - Erlang: rabbitmq/ra
  - Zig: zaft

- *: These are excellent libraries; however, they are unfortunately documented in Chinese. Please use a chrome translation extention like this one to read thier docs. I highly recommend giving them a try as they are genuinely valuable libraries.

- No you did not read the list wrong, **Python** and **JavaScript** are **NOT** supported. This is because I could not find any good raft libraries in python which are also Not too high level as specified above. I am reviewing JS raft libraries and might allow it in the future

- While I have reviewed the libraries listed above and ensured they meet the necessary criteria, I have not personally used every one of them (with the exception of hashicorp/raft, which I can personally vouch for). Therefore, it is possible that some may be broken or have other issues. If you encounter any such problems, please contact me IMMEDIATELY. Additionally, if you are aware of a good raft library that I have overlooked and would like to use, please reach out to me at +91 8618950413 or email us at cloudcomputing@pes.edu

Project Specification

Raft Node

Must handle the following (going above and beyond these are upto the developer)

- Leader election : selecting the leader for the Kllaft cluster
- Event driven architecture
- Eventual Consistency
- Failover management : must be able to provide standard raft failover guarantees (3 node cluster can handle single failure; 5 node cluster can handle 2 failures)
- Maintaining event log : event log of all changes being made (this is to be used to reconstruct the metadata store)
- Snapshotting : Creation and retrieval of the snapshots
  Take periodic snapshots of the event log at the leader to be able to provide a level of fault tolerance

NOTE : Most of the above are inherently supported by raft as a consensus algorithm; **Implementation requires a Raft library that necessitates manual development of the Raft Finite State Machine (FSM) State Transition Function, such as pyraft or hashicorp/raft. Higher-level abstraction libraries such as PyDbSync are expressly prohibited for this implementation. Refer above for permitted Libraries**

3D Print Management Software

- Raft3D must implement HTTP REST API endpoints accessible to external clients, satisfying the requirements specified below, in addition to maintaining inter-node communication for Raft consensus operations.

Objects Stored

The following objects need to be stored by Raft3D (and be made raft fault tolerant, this will be the state in your RAFT FSM)

1. **Printers** - The individual 3D printers present in the shop

   ```
   {
     "id": "", //can be int or string but must be unique for every printer
     "company": "", //type: string; eg: Creality, Prusa etc
     "model": "", //type: string; eg: Ender 3, i3 MK3+
   }
   ```

2. **Filaments** - The rolls of plastic filament used for 3D printing parts

   ```
   {
     "id": "", //can be int or string but must be unique for every filament
     "type": "", //type:string; options: PLA, PETG, ABS, TPU
     "color": "", //type: string; eg: red, blue, black etc
     "total_weight_in_grams": "", //type: int
     "remaining_weight_in_grams": "", //type: int
   }
   ```

   - `type` is the type of plastic filament which can be one of the following options: **PLA**, **PETG**, **ABS** or **TPU**
   - `color` is the color of the plastic
   - `total_weight_in_grams` is the weight of usable plastic on the filament roll when it comes from the factory, usually 1Kg
   - `remaining_weight_in_grams` is the remaining weight of plastic left on the filament roll. This will need to be reduced after every print is `DONE` based on the weight of print.

3. **PrintJobs** - A job to print individual items on a particular 3d printer using a particular filament

   ```
   {
     "id": "", //can be int or string but must be unique for every print_job
     "printer_id": "", //needs to be a valid id of a printer that exists
     "filament_id": "", //needs to be a valid id of a filament that exists
     "filepath": "", //type: string; eg: benchy.gcode
     "print_weight_in_grams": "", //type: int
     "status": "" //type: string; options: Queued, Running, Cancelled, Done
   }
   ```

   - `filepath` - The path of the 3D file that will be printed (usually in .gcode file format)
   - `print_weight_in_grams` - The weight of the completed print in grams. This cannot exceed the remaining weight of the chosen filament
   - `status` - Can be one for the following:
     - Queued - Default on Creation
     - Running - Print is running on the Printer
     - Done - Print is done
     - Canceled - Print was Canceled (during either Queues or Running State)

API Spec

The following endpoints need to be supported:

1. **POST printer** (e.g: POST /api/v1/printers)
   - Create a Printer
2. **GET printers** (e.g: GET /api/v1/printers)
   - List all available printers with their details
3. **POST filament** (e.g: POST /api/v1/filaments)
   - Create a filament
4. **GET filaments** (e.g: GET /api/v1/filaments)
   - List all available filaments with their details
5. **POST Print Job** (e.g: POST /api/v1/print_jobs)
   - Create a Print Job
   - Make Sure `printer_id` and `filament_id` are valid and belong to existing printers/filaments
   - Make sure that the `print_weight_in_grams` does not exceed (`remaining_weight_in_grams` of the filament MINUS `print_weight_in_grams` of other Queued/Running Print Jobs that use the same filament)
   - Initialize `status` to Queued. Do not let user set the status while creating Print Job
6. **GET Print Jobs** (e.g: GET /api/v1/print_jobs)
   - List all print jobs with their details
   - You may optionally allow for filtering based on status if you are feeling fancy
7. **Update Print Job Status** (e.g: POST /api/v1/print_jobs/<job_id>/status?status="running")
   - Update the status of a Print Job by taking in its ID and the next status which can be either **running**, **done** or **canceled**
   - A job can go to **running** state only from the **queued** state
   - A job can go to **done** state only from the **running** state
   - A job can go to **canceled** state from either **queued** state or **running** state
   - No other transitions apart from the ones specified above are allowed
   - *When a job transitions to **done** state, you need to reduce the `remaining_weight_in_grams` of its filament by the `print_weight_in_grams` of the current job*

Weekly Guidelines

Week 1

- Read the raft paper and understand the nuances
- Pick a language and Raft Library in that language
- Setup a simple HTTP Server
- Explore usage of the picked raft library
- Try starting to build a distributed KV store using the Raft library of your choice to solidify understanding of how to use the Raft Library - This also helps you realize if the library you picked has any issue (if so please reach out to us immediately!)

Week 2

- Build a distributed KV store using the Raft library of your choice to solidify understanding of how to use the Raft Library
- Starting implementing endpoints as specified in the API spec of this document
- Finish at least 2 endpoints (POST/GET Printers)
- Test to make sure the core Raft features such as Leader Election and Data Persistence during node failure work

Week 3

- Finish implementing all 7 endpoints along with the specified business logic
- Test to make all business logic contraints work (Filament weight is reduced when job is done etc)
- Test to make sure the core Raft features such as Leader Election and Data Persistence during node failure work

Demo

Note: You need not implement a frontend/client for the API endpoints, demo-ing using Postman/Insomnia etc is sufficient

The following is a non-exhaustive list of things that must work during your demo (refer the Project Specification Section section for full requirements) provided here for students to sanity check that their implementation is correcting working.

- To demo multiple nodes the following approaches can be done (not an exhaustive list)
  - Using multiple processes on different computers
  - Using multiple processes on same computer
  - Spawn multiple nodes on different VMs
- Leader Election must be demo-d
- The following flow must work:
  1. Create Printer
  2. Terminate the leader node such that New Leader is Elected
  3. List all nodes to make sure the Printer created in Step 1 using old leader is shown

References

- Raft Visualization
- Raft Paper & More