Network Programming Lab. 1

Warning

Any `academic misconduct` is not tolerated.

I highly recommend to avoid plagiarism, as it would be reported and will have serios consequence for you.

Restaurant Simulation

Let's get to the meat of the problem

(get it? because it's a problem about a restaurant)

The purpose of this task is for you to write a somewhat realistic simulation of how a restaurant works.

The general idea is that you have `Dining hall` and `Kitchen` .
The `Dining hall` generates `orders` and gives these orders to the `Kitchen` which prepares them and returns prepared orders back to the `Dining hall`.

Let's break problem into components

For ease of use and how clear is the json format, I used it to explain the design of each of the data structures you will need in your system.

Dining hall

The `Dining hall` has a finite amount of `tables` that "clients" can occupy. For simplicity's sake, at any given time a `table` can have only one order, thus if a restaurant has 6 `tables` occupied from total number of 10 `tables`, it can at most have 6 orders.

Main work unit of the `Dining hall` are `waiters` (which are a bit counter intuitively named) that take orders from the `tables` (clients).
The purpose of `waiters` is to "find" `orders`. Restaurant has limited number of `waiters` and is less than number of `tables`. For example at restaurant with 10 tables we can have 4-5 waiters.

Restaurant Menu

`Tables` (clients) generates `orders` based on restaurant menu. Menu consist of `foods`.
`Foods` used in our simulation are given in the list bellow:

Foods

- pizza, id=1

```
{
  "id": 1,
  "name": "pizza",
  "preparation-time":     20 ,
  "complexity":     2 ,
  "cooking-apparatus":    "oven"
}
```

- salad, id=2

```
{
  "id": 2,
  "name": "salad",
  "preparation-time":     10 ,
  "complexity":     1 ,
  "cooking-apparatus":    null
}
```

- zeama, id=3

```
{
  "id": 3,
  "name": "zeama",
  "preparation-time":     7 ,
  "complexity":     1 ,
  "cooking-apparatus":    "stove"
}
```

- Scallop Sashimi with Meyer Lemon Confit, id=4

```
{
  "id": 4,
  "name": "Scallop Sashimi with Meyer Lemon Confit",
  "preparation-time":     32 ,
  "complexity":     3 ,
  "cooking-apparatus":    null
}
```

- Island Duck with Mulberry Mustard, id=5

```
{
  "id": 5,
  "name": "Island Duck with Mulberry Mustard",
  "preparation-time":     35 ,
  "complexity":     3 ,
  "cooking-apparatus":    "oven"
}
```

- Waffles, id=6

```
{
  "id": 6,
  "name": "Waffles",
  "preparation-time":     10 ,
  "complexity":     1 ,
  "cooking-apparatus":    "stove"
}
```

- Aubergine, id=7

```
{
  "id": 7,
  "name": "Aubergine",
  "preparation-time":     20 ,
  "complexity":     2 ,
  "cooking-apparatus":    "oven"
}
```

- Lasagna, id=8

```
{
  "id": 8,
  "name": "Lasagna",
  "preparation-time":     30 ,
  "complexity":     2 ,
  "cooking-apparatus":    "oven"
}
```

- Burger, id=9

```
{
  "id": 9,
  "name": "Burger",
  "preparation-time":     15 ,
  "complexity":     1 ,
  "cooking-apparatus":    "stove"
}
```

- Gyros, id=10

```
{
  "id": 10,
  "name": "Gyros",
  "preparation-time":     15 ,
  "complexity":     1 ,
  "cooking-apparatus":    null
}
```

- Kebab, id=11

```
{
  "id": 11,
  "name": "Kebab",
  "preparation-time":     15 ,
  "complexity":     1 ,
  "cooking-apparatus":    null
}
```

- Unagi Maki, id=12

```
{
  "id": 12,
  "name": "Unagi Maki",
  "preparation-time":     20 ,
  "complexity":     2 ,
  "cooking-apparatus":    null
}
```

- Tobacco Chicken, id=13

```
{
  "id": 13,
  "name": "Tobacco Chicken",
  "preparation-time":     30 ,
  "complexity":     2 ,
  "cooking-apparatus":    "oven"
}
```

Order

`Tables` (clients) generates `orders`. An order should contain the following information:

- unique order id
- one or more menu items where the items indicate the ids of the menu items.
- the priority of the order (where it ranges from 1 to 5 , 1 being the smallest priority, and 5 - with the highest one)
- maximum wait time that a client is willing to wait for its order and it should be calculated by taking the item with the highest preparation-time from the order and multiply it by 1.3.

An example of an order:

```
{
  "id": 1,
  "items":  [ 3, 4, 4, 2 ],
  "priority": 3 ,
  "max_wait": 45
}
```

The timer of an order starts from the moment it's created.

`Order` has to be picked up by a waiter. The time it takes for a waiter varies, and I would say that a time between `2 and 4` should be realistic enough.

Kitchen

The `Kitchen` has a finite `order list`. This `order list` is shared across all `kitchen` instances. All orders which `kitchen` receives have to be added to a single instance of `order-list`.

Main work unit of the `Kitchen` are `cooks`. Their job is to take the `order` and "prepare" the menu item(s) from it, and return the orders as soon and with as little idle time as possible. `Kitchen` can prepare foods from different orders and it is not mandatory that one cook have to prepare entire order. Order is considered to be prepared when all foods from order list are prepared.

Each cook has the following characteristics:

- rank , which defines the complexity of the food that they can prepare (one caveat is that a cook can only take orders which his current rank or one rank lower that his current one):
  - Line Cook (rank = 1)
  - Saucier (rank = 2)
  - Executive Chef (Chef de Cuisine) (rank = 3)
- proficiency : it indicates on how may dishes he can work at once. It varies between 1 and 4 (and to follow a bit of logic, the higher the rank of a cook the higher is the probability that he can work on more dishes at the same time).
- name
- catch phrase

So a cook could have the following definition:

```
{
  "rank": 3,
  "proficiency": 3,
  "name": "Gordon Ramsay",
  "catch-phrase": "Hey, panini head, are you listening to me?"
}
```

Get creative on where and when to use this precious information about the cooks.

Another requirement not for the faint of heart is to implement the `cooking apparatus` rule. It comprises of the fact that a kitchen has limited space, thus a finite number of ovens, stoves and the likes.

Your `kitchen` configuration have to include a limited number of `cooking apparatus`. For example at `kitchen` with 3-4 cooks, we can have no more than 2 stoves and only one oven.

As you've noticed some menu items require one of these appliance and it's up to you to define what happens when a cook runs into the problem of no available machinery.

You will have to define the mechanism that will decide which cook takes which order.

System Design and Requirements

Time

I hope you noticed that I haven't indicated the time units, the numbers given are a "general" unit of measurement. In your system you should have the ability to easily modify the time units that you're using.

Example (this is more of a pseudo code):

```
#define TIME_UNIT 250
take_order(random(2, 4) * TIME_UNIT);
```

This is so you could experiment and check whether your system will behave differently depending on the time frames you chose.

System components

First of all you have to break down your system into components. The main 2 components are:

- `Dinning Hall`
- `Kitchen`

Each component, the `Dinning Hall` and the `Kitchen` in our case will represent a dedicated web server. The `Dinning Hall` and the `Kitchen` have to communicate with each other over the network using `HTTP` protocol.

Technical Requirements

- Each server should be developed as individual component, meaning that it should be developed in separate repository.
- Each server should run inside docker container.
- It is not mandatory to have `Dinning Hall` and `Kitchen` developed in the same programming language
- You can use any programming language except JavaScript(JS), but I encourage you to use such languages as Rust, Elixir, Erlang, Golang, Scala, Kotlin and any other language which has advanced concurrent model. Java, C#, Python are also allowed.

Let's analyze each component individually.

Dinning Hall

The `Dinning Hall` consists of `tables` and `waiters`. You have to design a mechanism which will simulate tables occupation. At start of simulation, tables should not be totally occupied and you have to take into a count that it takes time for a table to be occupied after it was vacated.

In the `Dinning Hall` you should have a collection(array) of `tables`.
`Tables` should be a dedicated objects. Each `table` should have a state of:

- being free
- waiting to make a order
- waiting for a order to be served

`Waiters` should be an object instances which run their logic of serving tables on separate `threads`, one thread per `waiter`. `Waiters` should look for tables which was not served, meaning that order was not picked up yet. For `Waiters` which are running on separate `threads`, tables are shared resource. `Waiters` are looking in the collection of `tables` for such table which is ready to make a order. When `waiter` is picking up the order from a `table`, it(table) should generate a random order with random foods and random number of foods, random priority and unique order ID.

Number of `tables` and `waiters` should be configurable.

After picking up an `order`, don't forget that this operation takes some amount of time. `Waiter` have to send `order` to kitchen by performing `HTTP` (POST) request, with order details.

When order will be ready, `kitchen` will send a `HTTP` (POST) request back to `Dinning Hall`. Your `Dinning Hall` server has to handle that request and to notify `waiter` that order is ready to be served to the table which requested this order.
Your task here is to design a mechanism for serving prepared `orders` to `tables`. The `order` should be served to the `table` by the `waiter` which picked up that specific order. When `order` is served `table` should check that served `order` is the same order what was requested.

Reputation system

Based on your implementation, the restaurant simulation will get a reputation, based on all orders prepared during the simulation. It indicates the success of your implementation.

After serving prepared order, you have to stop order timer and calculate order total preparing time.

```
Order_total_preparing_time = order_pick_up_timestamp - order_serving_timestamp
```

It is based on the 0 to 5 ⭐ system, 0 being the worst rating, and 5 - the highest. How you get the assigned the stars, you might ask? Well, since we can't really give it a taste test, what will define the number of stars given to each order would be the time-frame it took to complete.

Time frame	> ⭐
< max_wait	5
max_wait * 1.1	4
max_wait * 1.2	3
max_wait * 1.3	2
max_wait * 1.4	1

> max_wait * 1.4 | 0

After calculating order total preparation time you have to calculate order mark according specified logic. All orders mark are recorded and average restaurant reputation is calculated after each order was served.

Kitchen

The `Kitchen` consists of `order list`, `cooks` and `cooking apparatus`. The `order list` should be a single instance which holds all orders received from `Dinning Hall`.

The `Kitchen` should handle `HTTP` (POST) requests of receiving `orders` from the `Dinning Hall` and add received order to `order list`. For all received orders `kitchen` have to register time it was received and time is was totally prepared. Cooking time should be added to order before sending it back to `Dinning Hall`.

`Cooks` should be an object instances which run their logic of preparing foods on separate `threads`, one thread per `cook`. Your task is to design a mechanism which will prepare orders by using `cooks` as work unit. It is up to you to decide how orders will be managed and how foods will be assigned to `cooks` in order to be prepared. Your main goal is to reduce preparation time of each order.

The `kitchen` has a limited number of `cooking apparatus` and in our case we will use only stoves and ovens. `Cooking apparatus` should be object instances which work independently and in parallel. `Cooking apparatus` are sharable resources across all `cooks` and you have to carefully use them.

Number and types of `cooks` and `cooking apparatus` should be configurable.

When order is prepared, meaning that all foods from order are prepared. `Kitchen` should perform `HTTP` (POST) request with prepared order details to `Dinning Hall` in that way returning prepared order to be served to the `table`.

Communication and messages format (Communication protocol)

Your `Dinning Hall` and `Kitchen` API and requests payload should strictly follow defined format and configuration

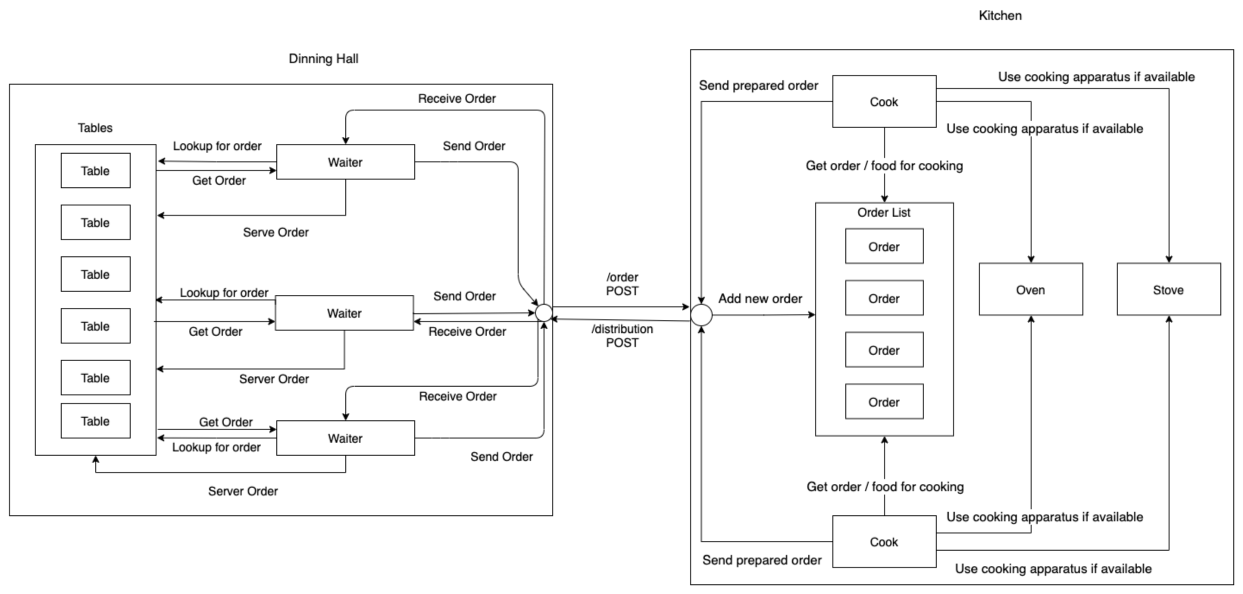
`Kitchen`

```
Endpoint: /order
Method: POST
Payload:
{
  "order_id": 1,
  "table_id": 1,
  "waiter_id": 1,
  "items": [ 3, 4, 4, 2 ],
  "priority": 3,
  "max_wait": 45,
  "pick_up_time": 1631453140 // UNIX timestamp
}
```

`Dinning Hall`

```
Endpoint: /distribution
Method: POST
Payload:
{
  "order_id": 1,
  "table_id": 1,
  "waiter_id": 1,
  "items": [ 3, 4, 4, 2 ],
  "priority": 3,
  "max_wait": 45,
  "pick_up_time": 1631453140 // UNIX timestamp
  "cooking_time": 65
  "cooking_details": [
    {
      "food_id": 3,
      "cook_id": 1,
    },
    {
      "food_id": 4,
      "cook_id": 1,
    },
    {
      "food_id": 4,
      "cook_id": 2,
    },
    {
      "food_id": 2,
      "cook_id": 3,
    },
  ]
}
```

System Architecture



Work process requirements

1. Your repositories should contain README.md files with project build and run instructions.
2. All development should be performed in dedicated feature branches. Direct commits and pushes to main repository branch are not allowed and you will be punished for such misbehavior
3. For all feature branches you should have dedicated PR (Pull Request) for merging it in main repository branch
4. Each pull request should have a description, background or some context and bullet points of what work have been done in this PR.

Suggested Order of Working on This task

1. Create 2 empty repositories and initialize 2 web servers in selected programming language(s).
2. Setup docker environment and run initialized servers inside docker containers.
3. Setup communication between your servers. Ensure you can make HTTP calls from `Dinning Hall` to `Kitchen` and viceversa.
4. At `Dinning Hall` implement basic logic of random orders generation without taking into a count `tables` and `waiters` logic and send generated orders to the `Kitchen`.
5. At `Kitchen` implement basic logic of cooking orders foods by at least 2 cooks, without taking into a count cooking apparatus rule.

6. Implement additional logic at `Dinning Hall` and `Kitchen` to follow all mentioned rules and ensure right communication between 2 components.

Test configuration

Your simulation will be tested using specifed configuration. Based on this configuration your simulation will be marked.

Dinning Hall

In dinning hall you have to have:

- 10 Tables
- 4 Waiters

Kitchen

4 Cooks:

1. Rank = 3 Proeficiency = 4
2. Rank = 2 Proeficiency = 3
3. Rank = 2 Proeficiency = 2
4. Rank = 1 Proeficiency = 2

Cooking aparatus:

- 2 Ovens
- 1 Stove

Submission and Grading Policy

In order to present laboratory work you have to pass mandatory checkpoints. Each checkpoint could be passed only in one day. It is not possible to pass all checkpoints together. The idea of checkpoints is to present your work progress in time.

Without passing checkpoints is not possible to get the final grade

Checkpoints Submission

1. 40% Checkpoint -> 1 week

    i. Two repositories with initialized projects. Don't forget about Readme and all other configurations
    ii. Your programs, which are actually web-servers, should run in docker containers.
    iii. Communication between containers should be configures. You have to prove that containers perform some https communication. Some logs in std output will be enough.
    iv. Initial logic of generating random orders and sending that orders to kitchen have to be implemented. Without taking into a count any synchronization logic. Just having some multiple threads which send some random order to kitchen.
    v. Initial logic of preparing foods at kitchen should be implemented. Just logic of having multiple threads picking up orders, preparing them all and returning them. Without any additional logic regarding to priority, cooking apparatus or sharing order foods between different cooks. Just simple logic of picking up order and preparing it all and returning it back to dinning hall.

2. 70% Checkpoint -> 1 week

    i. Implementation of dinning hall logic. Logic of having multiple waiters picking up orders from tables and serving that orders back.
    ii. Implementation of kitchen logic. It should include logic of preparing orders according orders priority and cooking of foods according foods complexity and cooks proficiency. Logic with cooking apparatus is not required.
    iii. Implementation of logging of components communication and logs of components work process

3. 100% Checkpoint -> 1 week

   Implementation of all task requirements, at this stage you have to focus on cooking apparatus rules, sharing of order food cooking between cooks and calculation of order rating and restaurant simulation average rating. Don't forget about optimization of your food cooking, your goal is to minimize cooking time and to have your simulation rating as bing as possible

Grading

Your laboratory work grade will be calculated based on your checkpoints grades and your simulation average rating. It will be calculated according the formula.

Grade = 0.5 * (0.4 * 40% check grade + 0.3 * 70% check grade + 0.3 * 100% check grade) + simulation average rating

For each week being late you will be punished with -1 point.

Minimum acceptance criteria

In order to get the minimum acceptable mark, which is 5, you have to present a project which includes:

1. Two web servers which communicates over HTTP protocol between them.
2. First web server is producer which produces some data on multiple threads (more than 5) and it sends these data from multiple threds to the second web server.
3. Second server is consumer, which receives and consumes data from first server and populates shared resources, a queue or stack with received data.
4. Second server also has multiple threads which extracts one element from shered resource and is sending that extracted data element from second server to the first.

Example implementation.

You have to create / reproduce your own and to not copy example.