UPDATE: Please note small update in auto-scaling algorithm: The orchestrator must start the auto-scaling timer *after* it receives the first Acts API request. More details in "Task 4" section.

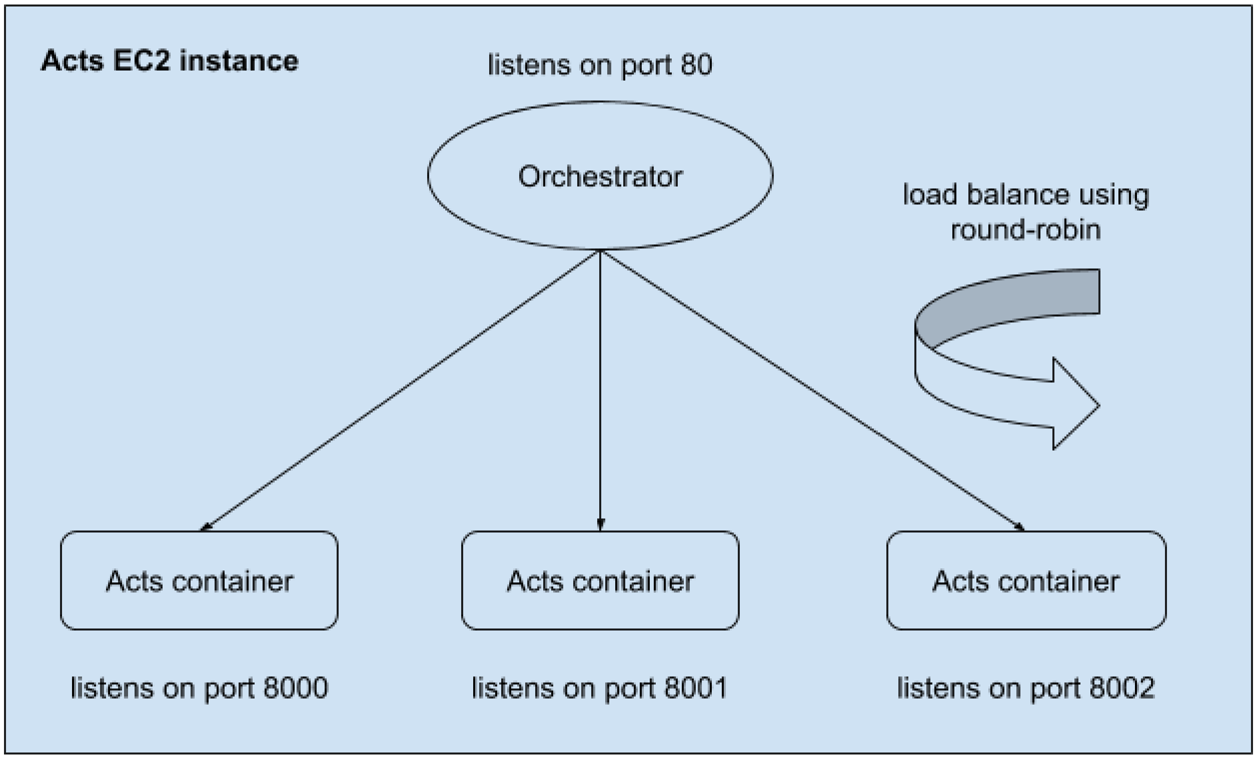
Final Project: Container Orchestration

The final project is focused on building a container orchestrator that can perform load balancing, fault tolerance, and auto-scaling. You will only be working within the Acts EC2 instance.

You will implement a custom container orchestrator engine that will:

1. Be able to start and stop Acts containers programmatically, and allocate ports for them on localhost.
2. Load balance all incoming HTTP requests (to the Acts EC2 instance) equally between all running Acts containers in a round-robin fashion.
3. Monitor the health of each container through a health check API, and if a container is found to be unhealthy, stop the container and start a new one to replace it.
4. Increase the number of Acts containers if the network load increases above a certain threshold.

The orchestrator engine will run inside the Acts EC2 instance, listen on port 80 of the public IP, and load balance all HTTP incoming requests equally to every Acts container.



The project is broken down into four tasks:

Task 1

On each Acts container, implement the following two APIs:

Health Check
Route: `/api/v1/_health`
HTTP Request Method: `GET`
Relevant HTTP Response Codes: `200, 500`
Request: `<empty>`
Response: `<empty>`
Comments:

- If the server is functioning normally, i.e., able to server all requests and read/write to database, then respond with code 200.
- If the server is disabled for some reason, such as due to not being able to read/write data, or if there’s no free memory available to allocate new objects, then respond with code 500.

Crash server
Route: `/api/v1/_crash`
HTTP Request Method: `POST`
Relevant HTTP Response Codes: `200`
Request: `<empty>`
Response: `<empty>`
Comments:

- Once this API is called on an Acts container (and a 200 response is returned), henceforth every HTTP request to every route, including the health check, must be responded with code 500.
- This API will be called by the testing script a certain point to permanently disable an Acts container and then check if its health check (and other routes) return 500 subsequently.

Task 2

The second task is to implement the load balancing feature of the orchestrator engine.

The orchestrator can be written as a simple web server in any language, which will run on the Acts instance itself. It must listen on port 80 and forward HTTP requests to healthy Acts containers. The Acts containers themselves must be listening on other ports (8000, 8001, 8002,..) on localhost. The orchestrator engine must track these active ports.

The HTTP requests must be distributed in a round-robin manner to every Acts container. Example: if there are 3 Acts containers called A, B, and C, and 6 HTTP requests were made towards the Acts EC2 instance, then the orchestrator engine must route the requests to the Acts containers in following order - A, B, C, A, B, C.

Note that you must only forward the API routes to the Acts containers, not other routes such as "`/`". The "`/`" route must be handled by the orchestrator itself.

Task 3

The third task is to implement the fault tolerance feature of the orchestrator engine.

For every 1 second interval, the orchestrator engine must poll the health check API of each running Acts container. If it detects an unhealthy container, it must stop that container and start a new container (using the same Acts docker image). The replacement container must listen on the same port that the unhealthy container was listening on.

Task 4

The fourth task is to implement the auto-scaling feature of the orchestrator engine.

The orchestrator must keep a track of the number of incoming HTTP requests in the past two minutes.

At every 2 minute interval, depending on how many requests were received, the orchestrator must increase or decrease the number of Acts containers:

- If the number of requests is less than 20, then only 1 Acts container must be running.
- If the number is >= 20 and < 40, then 2 Acts containers must be running.
- If the number is >= 40 and < 60, then 3 Acts containers must be running.
- and so on...

Note 1: Do *not* count requests made to `/api/v1/_health` or `/api/v1/_crash` for the above number.
Note 2: Count only the **Acts API requests** received by the orchestrator, *not* other requests such as for "`/`".

Example: once a 2 minute interval is over and 20 HTTP requests have been received within that interval (*and* forwarded to the sole Acts container), the orchestrator must then programatically start a new Acts container and assign a port for it to listen on localhost. Then, all subsequent requests must be load balanced between the two containers in a round-robin manner.

The first container running must be listening on port 8000 on localhost. Subsequent containers that are run must listen on port 8001, 8002, 8003, and so on. You can assume you won't run more than 10 containers at a time.

UPDATE: The orchestrator engine must begin the 2 minute timer **after** it receives the first Acts API request. So, initially, there's no autoscaling timer. And when the first API request is received by the orchestrator, it must start the regular 2 minute timer and start counting the no. of API requests received every 2 minutes. The first API request, which caused the timer to start, **must** be counted as among the requests received in the first 2 minutes.

And, before *every* run of the script, make sure the auto-scaling timer disabled until the first API request is made.

As new containers are started, the orchestrator must load balance incoming requests to them as usual, and also poll their health check every 1 second to check if they need to be stopped and replaced.

Remember that all the Acts containers must serve requests using a common database. If an act was posted by making a request to one container, then it must be retrievable by making requests to any of the other running containers. How this is implemented is up to you.

Integration with the mobile Application

You will integrate your cloud backend with the mobile application that will be given to you. This app will make POST requests to /api/v1/acts. You will be able to select the category and user in the UI.

You will collect at least a 100 different acts (real ones - no sharing allowed). An online leaderboard will connect to your instance and collect the various acts (and check for uniqueness). You can see your position on the leaderboard (the address of which we will share with you soon). If the mobile app does not work, you will need to fix the server side APIs; the mobile app will give an error message indicating which API was used and what the failure code was.

Other points

You must keep the AWS setup that you created in your last assignment still running. The Users container must still be running on the Users EC2 instance, and the AWS Application Load

Balancer must still perform the path-based load balancing before requests are ever received by the Acts container orchestrator.

As this project is fairly complex, we have outlined some of the scenarios that test script *may* check for so that you may have a clearer picture of what needs to be done:

1. In the Acts EC2 instance, the orchestrator engine must be running and listening on port 80.
2. Before the script is run, ensure that the count of HTTP requests received by the orchestrator is set to zero. And so, there must initially be only 1 Acts container running and listening on port 8000 on localhost.
3. All Acts containers must be from the same image named “acts”, with tag “latest”.
4. All requests made to port 80 of the public IP must be routed by the orchestrator to the sole running Acts container.
5. A GET request to `localhost:8000/api/v1/_health` (from within the instance) must return 200.
6. 20 HTTP requests will be made to port 80 of the public IP. After the 1st request was received, the orchestrator must have started the autoscaling timer. These 20 requests must have been counted as received within the first 2 minutes.
7. The script will sleep for 2 minutes.
8. The orchestrator engine must have started a second Acts container listening on port 8001 on localhost by then.
9. Anywhere between 1 to 10 HTTP requests will then be made towards the Acts EC2 instance.
10. The orchestrator engine must now route these requests in a round-robin manner to the two containers. Any container can be picked as the starting one.
11. A POST request will be made to `localhost:8000/api/v1/_crash`
12. The script will then sleep for 20 seconds, and check if a new healthy container (with a different container ID) is running and listening on the same port 8000. If the unhealthy container is still running, this test will fail.
13. The script will sleep for 2 minutes.
14. The orchestrator engine must have scaled down the containers by then. It must stop the container running on port 8001, the 8000 port container must still be running.

Due Date: Last working week of the semester. April 22.
Marks 20

Distribution of Marks

Item #	Item	Marks

1	Integration with Mobile App	2
2	Health check and Crash server APIs	2
3	Load Balancer - round robin	4
4	Fault Tolerance	4
5	Scale Up/Down	4
6	#unique acts uploaded (min 100)	2
7	Over and Above specification (make a generic load balancer or a deployment engine to deploy a generic application to multiple containers.)	2
	Total	20

You need to submit a project report which outlines these steps using the template in the share.