For Developers: Making Changes to the System

Getting Started

- Begin by familiarizing yourself with the codebase. There are seven folders you'll want to explore:
    - In *Connector*, HTTPClientExample.java contains the logic for a node to join itself to the swarm. The IP address of the manager node is hard-coded and the worker simply needs to send the manager its IP address over HTTP to tell the manager that it exists.
    - In *Scaler*, Scaler.java contains logic for changing the number of containers running a service depending on a value. In Main.java, the number of tasks on a RabbitMQ instance is used as that value. It gets this number using RabbitMQHandler.java, which provides an easy way to see how many messages are on the queue. Scaler.java also changes the number of nodes depending on the number of running containers. Finally, HttpConnectionThread.java constantly listens for HTTP requests from nodes trying to join the swarm and handles them.
    - In *RabbitMQSender*, RabbitMQSender.java contains the logic for publishing transcoding tasks to the RabbitMQ instance. This class is used for testing and shouldn't be needed by the full system, as it will be receiving requests from a third party.
    - In *RabbitMQReceiverAndPuller*, Main.java contains the primary logic of the system. It connects to RabbitMQ, dequeues a task, pulls the necessary file from the storage service, transcodes it, and sends the result back to the storage service.
    - *Interfaces* contains key interfaces, implementations, and enums used by our other code:
        - FilePuller.java is an interface for pulling sets of files from a storage service; it is implemented by S3FilePuller.java.
        - FileUploader.java is an interface for uploading files to a storage service; it is implemented by S3FileUploader.java.
        - NameManager.java is an interface for creating a consistent naming scheme for video partition files; it is implemented by VideoNameManager.java.
        - TaskJSONConversionUtility.java is an interface for converting between task objects and JSON strings; it is implemented by TaskJSONConverter.java.

        - TaskPublisher.java is an interface for publishing a task object to a queue; this is used by V3 for sending tasks to our queue.
        - TaskSaver.java is an interface for saving a task object to a database; it is implemented by TaskSaverImpl.java.
        - TranscodingTask.java is an interface for storing all of the details associated with a given task; it is implemented by Task.java.
        - UtilityCalculations.java is an interface for performing basic calculations related to partitions and segments; it is implemented by PartitionCalculator.java.
        - VideoInfo.java is an interface for getting basic information about a video file; it is implemented by VideoInfoImpl.java.
        - VideoTranscoder.java is an interface for segmenting and transcoding a video file; it is implemented by VideoTranscoderImpl.java.
        - EventType.java is an enum storing whether an event is normal or an error.
        - EventName.java is an enum describing what particular event occurred.
        - Resolution.java is a record storing a resolution name and bitrate.
    - In *docker-swarm*, deployment.yml is used to deploy the *sender*, *receiver*, and *scaler* images, while rabbitmq.yml is used to deploy the RabbitMQ "middleman."
        - It's worth noting that we're using RabbitMQ's default Docker image. There are many aspects of it that can be configured in a custom manner, such as changing the host, changing the username and password, and adding a health check. See here for more information.
    - In *fastApi*, main.py is all the code needed to start up a FastAPI service on a worker node allowing it to easily join the swarm when needed.

Making Changes

1. Pull the codebase from GitHub.
2. Make any desired changes to the Scaler, RabbitMQSender, and/or RabbitMQReceiverAndPuller code.
3. For each folder that you edit, package its code by running: `mvn clean package -DskipTests`
4. Make a Docker Hub account if you don't already have one. Log in to Docker using your credentials and then build each image by running: `docker build -t <YOUR_USERNAME>/<SERVICE_NAME>:<VERSION_NUMBER> .`

SERVICE_NAME is *scaler*, *sender*, and *receiver* for the folders mentioned above.

5. Push your Docker image to DockerHub by running: `docker push <YOUR_USERNAME>/<SERVICE_NAME>:<VERSION_NUMBER>`
6. In *docker-compose.yml*, under whichever services you changed, change *lsnow987* to your Docker username.
7. You are now ready to deploy your edited system! Odds are that you'll have some trial-and-error getting your changes to work, so you'll have to repeat these steps. It gets easier after a few times 🙂

Deploying the System

Requirements

- Obtain a Ubuntu computer to use as the manager of the Docker Swarm. It should have Docker installed.
- All of your devices need to be on the same network and subnetwork. This can be achieved by using a WiFi hotspot or by connecting all devices to a particular router.
- All the worker machines should have Python, Uvicorn, and Docker installed.

Deployment Process

1. On the manager, configure the firewall by running the following commands:
    a. `sudo ufw enable`
    b. `sudo ufw allow 22/tcp`
    c. `sudo ufw allow 2376/tcp`
    d. `sudo ufw allow 2377/tcp`
    e. `sudo ufw allow 7946/tcp`
    f. `sudo ufw allow 7946/udp`
    g. `sudo ufw allow 4789/udp`
    h. `sudo ufw reload`

    The next steps are all run on this machine unless otherwise specified.
2. Run: `nmcli dev status`.
3. Locate the first device name on the list outputted by the previous step and run: `nmcli dev show <FIRST_DEVICE_NAME>`
4. Use the IP address and look for `IP4.ADDRESS[1]`. This is the IP address of this device. (Ignore the slash and numbers that follow it.) Run: `sudo docker swarm init --advertise-addr=<IP_ADDRESS>`

`--listen-addr=<IP_ADDRESS>:2377`
You should get back a `docker join` command.

5. Open *deployment.yml*. Find the *DOCKER_TOKEN* environment variable under the *scaler* service. Paste in the full command you got from the previous step between double quotes as the value for this variable.
6. In the same file, find the *RABBITMQ_HOST* environment variable under the *receiver*, *sender*, and *scaler* services and set it to the IP address of the computer which will host the RabbitMQ middleman. You can determine the IP address of a computer using the `ipconfig` command on Windows or `ifconfig` on Linux.
7. Under *scaler*, there's another environment variable called *IP_ADDRESSES.* For this variable, input the IP addresses of the worker computers separated by commas. Use the same commands mentioned above to find the IP addresses. If you don't know all of the IP addresses, don't worry; workers can dynamically join (see Step 11).
8. Under *sender* and *receiver*, change the environment variable named *IS_DB_SETUP*, setting it to *true* or *false* depending on whether you want to enable logging to the database. Under both of those, also set *MONGODB_HOST* to the IP address of the node on which the database is running. Use the same commands mentioned above to find its IP address.
9. Under *receiver*, set *DB_HOST* to the IP address of the node running the object store and *DB_PORT* to the port on which it is running.
10. On the machine you'll be using to host RabbitMQ, start up RabbitMQ by running in the *docker-swarm directory*: `docker compose -f rabbitmq.yml up`
11. On Ubuntu worker machines, you may need to run the following commands to allow the next step to work:
    a. `sudo ufw enable`
    b. `sudo ufw allow 8000/tcp`
    c. `sudo ufw reload`
12. For worker machines which aren't in the initial list of pre-configured IP addresses (see Step 7): Navigate to the Connector directory and run: `mvn clean test -DmyIpAddress=<YOUR_IP_ADDRESS> -DtargetIpAddress=<MANAGER_IP_ADDRESS>`
13. On all the worker machines, download *main.py* from the *fastApi* folder and run it using this command: `uvicorn main:app --reload --host 0.0.0.0`
14. Back on the Ubuntu manager computer, run in the *docker-swarm* directory: `sudo docker stack deploy -c deployment.yml myapp`
    Congratulations! The system is now up and running and will auto-scale as needed.

15. When you're done using the system, run: `sudo docker stack rm myapp` on the manager and `docker compose down` on the RabbitMQ node. You can now kill Docker and Uvicorn on all nodes.

Running the Full System

Above was a description of how to deploy our subsystem. It assumes that:

- There is already a task orchestrator for creating and sending tasks as well as a cache for efficient streaming. See here for how to deploy that subsystem.
- There is already an object store for storing the original video files as well as the resulting processed video files. See here for how to deploy it.