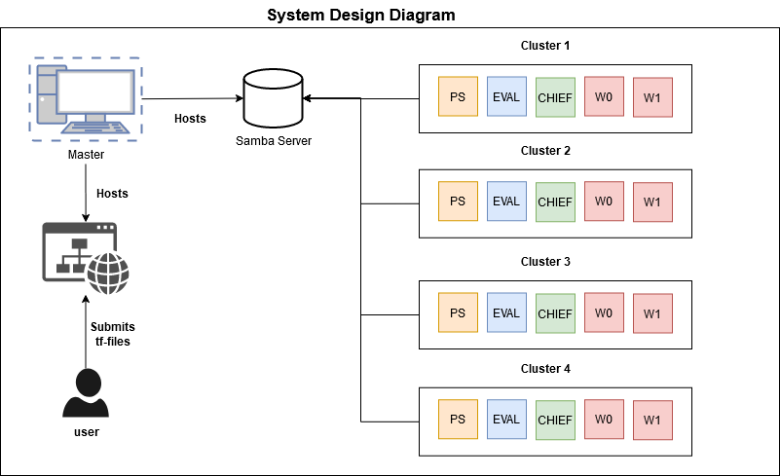
Research Question Number 1:

**1.2 Cluster System Architecture, setup and communication.**

****

***Figure 1.2 System Design Diagram***

As stated previously, the aim of this research is to allow researchers to be able to use this platform as a service to be able to not only train their neural network model concurrently but also be able to concurrently train multiple neural networks models by submitting their code to the system.

Twenty low-to-medium end computers are distributed into four clusters as shown in figure 1.2 and utilises a main-master node that is used to be able to coordinate job tasks with each cluster while also accepting user input using a front-end website.

For code execution isolation, we have opted to use Docker containers as (reference authors) stated that apart from virtual machines, Docker containers also present a level of isolation which also allowing containers to directly communicate with the operating systems kernel. This will be demonstrated later when explaining how containers are deployed.

As can be seen in pipeline diagram, different services will be communicating with each other to be able to complete a task. These services apart from the Samba server and TensorFlow are REST API services created using Java Spring. This approach enabled the master node to send concurrent job requests to each cluster and await each cluster to return a response.

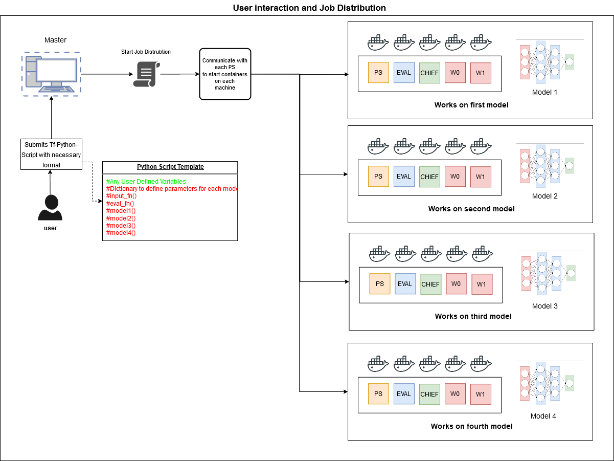
****

Figure 1.3: User Interaction and Model Distribution to each cluster.

**Python Template file and TensorFlow code configuration**

Firstly, for the researcher to submit a job, they need to upload their code into their system. However, not only does the user need to have the TensorFlow python file code, the system needed to be able to extract the required model functions and datasets that these models will use. A solution to this issue was approached as depicted in figure 1.3. The user will be required to submit a python file that follows a coding standard that matches the requirements needed. Thus, we created a python Template file that will requires model and dataset functions to follow a specific naming convention. Also, we created a dictionary that will require the user to specify the model function name and the number of steps required for that specific model to be trained.

The reason behind the python template file is that as Distributed TensorFlow with the Parameter Server Strategy utilizes estimators to train a neural network. When initializing the estimator, it requires the model name, input function and evaluation function and then starts training the neural network. So, with the use of naming conventions, the python file is then parsed to extract the model names and functions. Then, by using our custom TensorFlow python runner script we can import the user file alongside the script and pass the function names as command line arguments when executing it.

**Main-Master node and services.**

Secondly, as can be seen in figure 1.2 and in the pipeline, the Main-Master node will handle three services : 1) The front-end website where the user will be able to submit their job request, view cluster progress, view model progress and the ability to download the trained model; 2) A samba storage service that will allow each cluster to access the same storage medium; 3) The back-end service that will be responsible with provide output to the website and that is responsible to send and receiving REST API calls between each cluster. It is with this service that the master can concurrently communicate with each cluster.

When a job is submitted by a user as seen in figure 1.3, the back-end service is responsible to handle the following tasks to be able to distribute Neural networks on each cluster:

1. Check user file template validity and file naming conventions to avoid file naming exploits as stated previously.
2. Parse user file to extract neural network models the python file contains.
3. Create a new job directory in the shared storage and create sub-directories which will contain specific files and configurations for each cluster to handle.
4. Concurrently send A REST API request to each cluster using a Plain old Java Object (POJO) that contains cluster job specifications.
5. Wait for each cluster to return a response about job deployment progress and return an output.

**Master node in a cluster and services**

Thirdly, each master node in a cluster will await a request that is done by the master node via a REST API request. This service will handle the follows tasks, the file directories can be further demonstrated in figure 1.4:

1. Create sub directories in the directory that was specified by main-master node that will contain the configurations that will assign each node which docker container configuration will deploy.
2. Set TF\_CONFIG in each sub directory, this file is needed by TensorFlow to recognise the role of the current node when it is executed.
3. Create and place a .env file for each node to specify how the docker containers will be composed and what files to mount unto the containers.
4. Send a concurrent request to each node in the cluster to start deploying docker containers, check if containers deployed successfully and return a response to the main master.
5. Await each worker node to respond on deployment status and return a response to the Main-Master node.

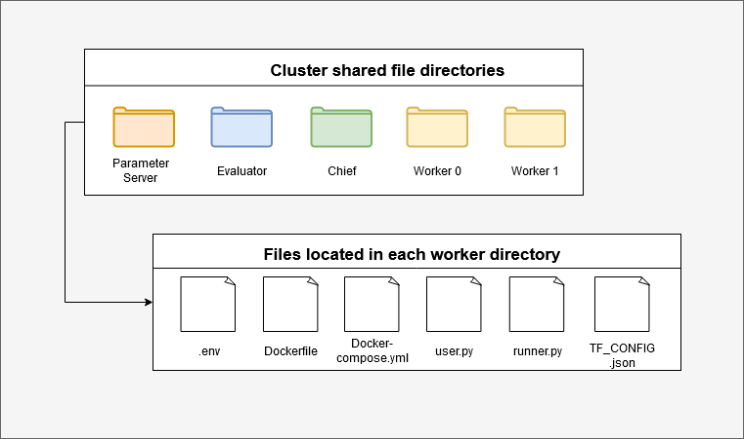


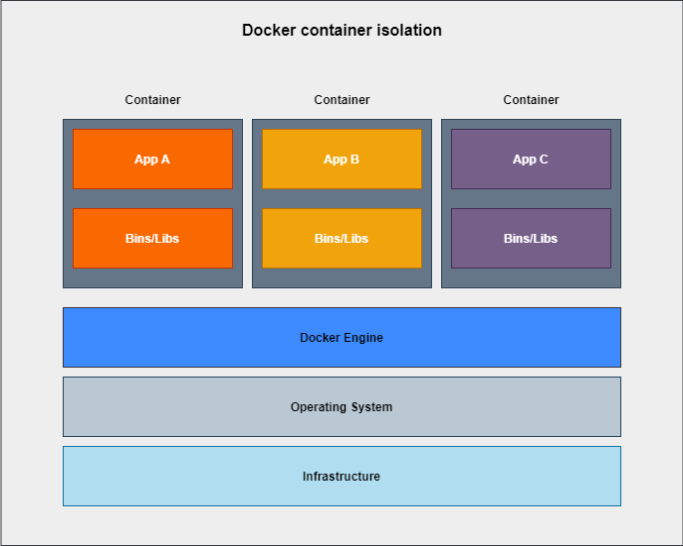
Figure 1.4: Cluster shared file directory and node files.

**Cluster worker nodes**

The last most important component of this platform is the service that will handle the automation of the docker container deployment. Each node in the cluster will await a request that is done by the master node of the cluster (The Parameter Server node). This service will handle the docker-container deployment on the machine by taking the parameters sent by the master node in the cluster and execute a command line argument to deploy the container on the host machine itself. After deploying the container, it executes another command line argument to check if the container successfully deployed and return a response to the master node regarding if the container deployment was a success or a failure.

* **Docker containers and code execution isolation.**

Docker containers will allow a level of isolation between the host and the container when executing any type of python script. Thus, allowing the code to be executed in an isolated environment. If the user decides to incorporate any malicious python code into the template file, it will only affect the container and it will not the host machine itself even if containers utilize the host’s kernel space as can be seen in figure 1.5.

****

*Figure 1.5: Docker container isolation from the operating system*

The docker container that will be used to execute the user’s python file. Thus, a custom build image was created by firstly, pulling the ubuntu:18.04 base image from docker-hub and secondly, install the necessary software python3, python3-pip, smbclient and cifs-utils (these are used to make the container connect with the samba server on the main-master node) and TensorFlow.

Finally, it will create the required directories to mount the samba server and where the TensorFlow program will be executed from by using the specified directory for that node as can be seen in figure 1.4 that was created by the master node in that cluster. After the container has deployed successfully and connected with the samba server, a python command will be executed to run the runner.py together with the user’s template file.

In the python command passed through as arguments, the model function that this cluster will process, model directory and the path to the TF\_CONFIG.json will be added for the program to process. Then, Distributed TensorFlow training process can start training once all nodes in the cluster connect with each other as previously stated when explaining the Parameter server strategy requirements.

System Limitations :