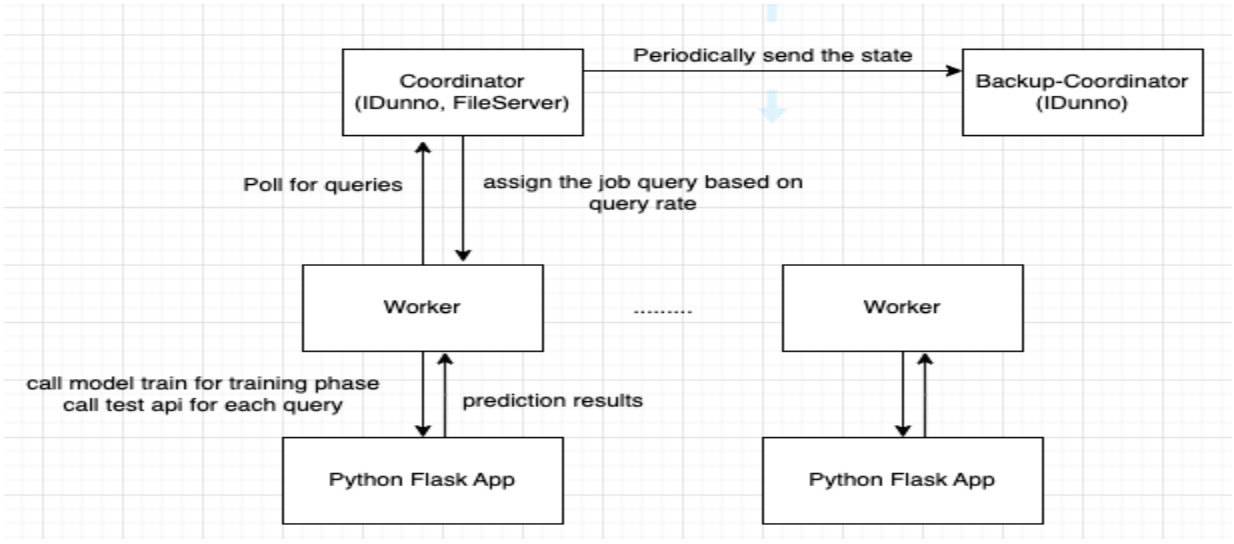
IDunno - Distributed Learning Cluster
CS 425 - MP4 - Group 70 (@sushmam3 and @jadonts2)

Design:



Our IDunno system is built on top of the same SDFS and Full membership list from previous assignments, with minor modifications and safety/performance improvements as needed. At each node, we keep the 5 previous threads (TCP listener, UDP listener, Ping/ACK thread, leader election thread and command thread), with the addition of a worker thread which polls the IDunno coordinator for queries, and said IDunno coordinator protocol thread for the coordinator node (but importantly not the backup coordinator, until it is promoted). Additionally each node runs an HTTP server for localhost-only as a level of abstraction to facilitate communication between Python code (for training models and running inference) and the Java code (all other functionality). All messages between VMs flow through a single port, and are dispatched to the right message handler based on type (differentiated between IDunno worker, IDunno coordinator, file server, file server coordinator, and member). IDunno commands are relayed to the coordinator by the main command handler thread seen by the user.

It currently supports image classification using Resent101 and Inception V3 neural nets. We are using a pre-trained pytorch model deployed using Python Flask Application exposing RESTful API (Port 8080) to perform the inference. The system is scalable to accommodate any number of models, by adding the model details in constants.java of IDunno folder.

All machines run the worker thread which polls the coordinator for queries and sends the result (computed through Flask inference API) back, until there are no pending queries remaining. The coordinator periodically compiles the prediction results and writes it to SDFS and sends the current state (Jobs running and their progress) to the back-up coordinator. All test datasets and results are stored in SDFS. All communication between worker, coordinator and backup coordinator is through TCP.

We have implemented commands

1. ‘train-start’ - Coordinator instructs all workers to load the pretrained model
2. ‘add-job <modelname>’ - Adds the job details to the coordinator.

3. ‘set-batch-size <modelname>’ - Informs the coordinator about the query batch size for the model. Queries are split up among VMs according to this batch size.
4. ‘start-job <modelname>’ - Coordinator schedules the VMs which are free to begin processing the queries. Workers fetch the SDFS files in the Query list and perform inference on those data files. The coordinator schedules queries in the order that they are added to the schedule queue
5. ‘stop-job <modelname>’ - Coordinator instructs all VMs to stop processing the queries for that model. It removes the queries for a given model from the scheduling Queue.
6. ‘get-job-assigned-VMs’ - Lists the VMs currently assigned to process the queries of a given job
7. ‘get-job-stats’ - Computes and displays the statistics of ongoing jobs such as moving average query rate, total number of queries processed so far etc.,

MP3 SDFS is used to store test data files and results of inference jobs. Mp2 Failure detector is used to detect failure of workers and rebalance the query rate by reassigning the job to a healthy VM. Failure detector is also used to detect coordinator/backup coordinator failure and elect the new coordinator/backup coordinator and maintain the job processing rate of the system. MP1 distributed logging is used to debug the issues in the system and measure performance metrics.

Performance Metrics:

	Time taken for second job to begin (in seconds)	**Time taken to resume normal operation after coordinator failure (in seconds)**
	5.72	5.53
	6.87	7.57
	7.13	6.29
	5.25	8.63
	6.43	5.95
Average	**6.28**	**6.794**
StdDev	**0.7**	**1.14**

1a) When a job is first added to the cluster, initially half of the VM’s are assigned to each job. This is guaranteed, as the code automatically distributes VMs this way on job startup (since we cannot guarantee anything about the rate of completion of the second job). Both our models take an average of 1.2 seconds per query, hence the ratio of resources that the system decides is 50:50.

Batch size - Job 1	Batch size - Job 2	Ratio
3	3	50:50'
5	3	70:30'
3	5	30:70'
4	2	80:20'

1b) It takes an average of 6.28 seconds for the second job to begin processing queries, measured through MP1 logs (in the program, subtracting the time of job request from the time of first query completion and reception).
2) After a single worker VM fails, normal operations are not affected. This is expected, as our design is worker focused rather than coordinator focused.
3) After coordinator failure, the system takes an average of 6.79 seconds to resume normal operations, measured from the time the first coordinator fails to the time the backup coordinator's protocol thread begins execution. Importantly, queries are still being processed during this time. The only functionality that is suspended is sending the state to the back-up coordinator.