IDunno: a Fault-tolerant Distributed Learning Cluster

Project Overview

Our team implemented the IDunno, a Distributed Machine Learning Cluster, with a unidirectional, ring-based, ping-ack style Simple Distributed File System with **Golang, gRPC, Protobuf, PyTorch, and React.** We use **gRPC** and **Protobuf** as the means of serialization and remote procedure call because they are both efficient and idiomatic for **Golang**. We utilize **PyTorch** for the ML task. In summary, our IDunno consists of seven parts: 1) a DNS server, 2) a coordinator server (and a backup coordinator server), 3) 10 SDFS servers, 4) 10 Golang workers, 5) 10 Python Runners, 6) A HTTP server for serving real-time statistics, 7) A frontend UI built with **React** that can view real-time updates on IDunno jobs.

IDunno Architetcure

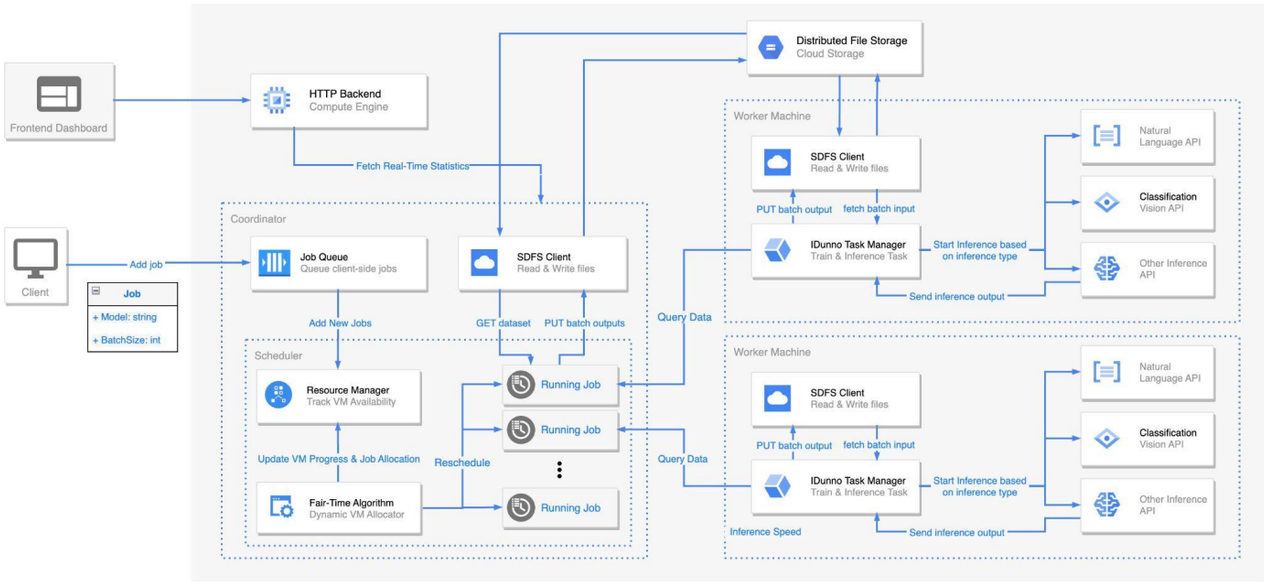


Figure 1. IDunno Design Schema

In our design architecture, every machine is simultaneously a Idunno worker, a SDFS server, and a Idunno client. An Idunno coordinator is elected based on the earliest join time. Coordinator has three three major components: job queue (**JQ**), resource manager (**RM**), and a fair-time scheduler (**FTS**). **JQ** queues incoming model inference request and dispatch requests in FIFO order. **RM** stores information about each worker machines regarding their scheduled job ID, currently running batch input, last query time, etc. **FTS** is responsible for schedule each job in a set of worker machines. FTS adopts 2 scheduling modes: local and global.

Local scheduling mode keep track of the **average second per query** of each job during the last 10 seconds, say $S_1, S_2, \cdots, S_n$, and calculate number of workers to assign to each job by formula

$$\frac{n \cdot S_i}{\sum_{j=1}^{n} S_j}, \forall i = 1, \cdots, n$$

where n is total number of workers, and i is a given job i.

On the contrary, global scheduling mode applied a more globally-aware algorithm that allocate workers based on the **expected query rate** of each job throughout the entire lifetime. The expected query rate of a job, given C worker runs in parallel, is calculated as

$$qps_i = \frac{Q_i}{e_i + S_i \frac{R_i}{C}}$$

where Q_i is total query count, e_i is time that has already elapsed for job i, S_i is model's second per query, and R_i is the remaining unfinished query count. The algorithm trys to find a set of machine allocation count $C_1, \cdots, C_n$ to each job such that we want to

$$\begin{aligned} &\underset{(C_1,\ldots,C_n,x)\ \in\ \mathbb{R}^{n+1}}{\text{minimize}} && x \\ &\text{subject to} && |qps_i - qps_j| \leq x, \forall i, j \in \{1, \ldots, n\}, i \neq j \\ & && \sum_{i=1}^{n} C_i = n,\ C_1, \cdots, C_n \geq 0 \end{aligned}$$

In other words, we want to find the optimal number of machines to allocate to each job that minimizes the pairwise expected query rate difference between each two distinct jobs. In above expression, x is the minimized maximum pairwise query rate differences. We solved this problem using dynamic programming in $O(mn^2)$ time where m is total running jobs and n is total number of active worker machines.

Both algorithms support **an arbitrary number of simultaneous running jobs**, and would work better as total number of worker machines increases. **FTS** periodically re-run the scheduling algorithm to see if any reschedule operations need to be done in order to satisfy the fair-time inference of IDunno system.

Once a worker is received a inference rpc call, it starts querying batch data from coordinator. Coordinator would then send a new batch input (consisting of a batch of SDFS filenames) to worker. Worker then fetch the actual files from SDFS, which could be a list of images. After that, worker run inference on those images and send inference batch output as well as accuracy metrics back to coordinator. This query-reply cycle is repeated until worker is rescheduled to some other jobs or the current job is completed. Between each query data request from worker to coordinator, we added a **500ms backoff** so that coordinator will not be overwhelmed by massive requests sent by lots of workers at the same time. Since each worker gets a different batch of data, it allows our system to fully take advantage of the parallelism of multiple workers.

Once a job is finished, coordinator would merge all batch outputs and calculate the average accuracy scores, and then send the resulting inference output to SDFS.

Whenever a worker is crashed, failure detector will detect that and notify the **RM**. **RM** would then refresh its worker list to exclude failed workers, and restore the running batch from this worker back to “available” state. Then, rescheduling will be executed to generate the new fair-time allocation schedule.

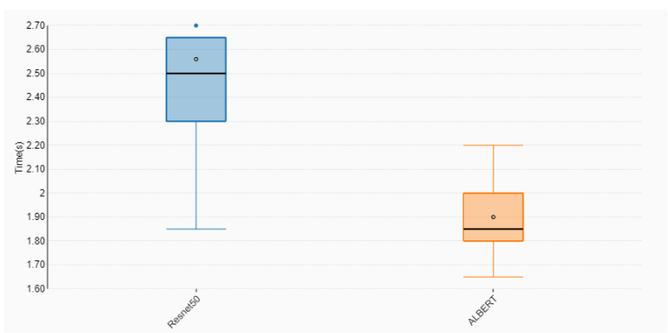
Coordinator periodically backup its data to its successor in the ring so that, whenever coordinator crashes, its successor will restores its data structure from backup data, and become the main coordinator again.

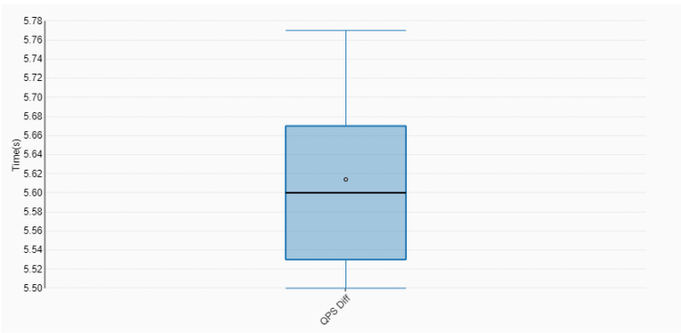
Past MP Use

We heavily utilize both MP2 - Distributed Group Membership and MP3 - Simplified Distributed File System in this MP. MP2 is an essential part of this MP since we leveraged the same ring-based, ping-ack membership list. MP2 is extremely important to correctly detect server failures. At the same time, SDFS is critical for uploading datasets and serving ML inference tasks. The failure tolerance nature of our MP3 guarantees that all uploaded datasets will be correctly processed. In this MP4, we further ensured that all tasks will be completed regardless of any worker/coordinator failure.

Measurements

Fair-Time Inference



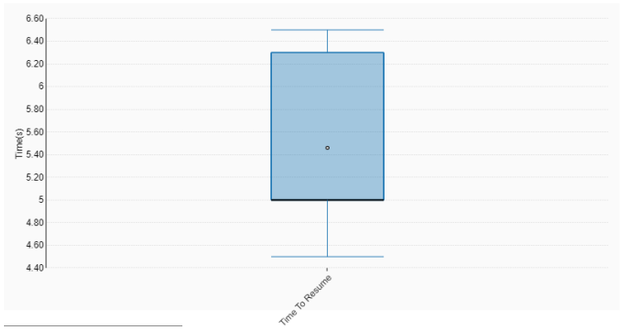


a) We started one jobs: Resnet50 with batch size 4, and add another job ALBERT with batch size 8. Based on our measurements, resnet50 and ALBERT take roughly 2.56 and 1.88 second to run one query respectively. We know that slower model should be given more machines, and the ratio of second per query should be proportional to the ratio of resources. By the formula of the local mode scheduling algorithm above, we get the number of machines to assign to Resnet 50 and ALBERT are 5.77 and 4.23. This result is expected from our observation, as our scheduler indeed assigned 6 VM to Resnet50 and 4 VM to ALBERT, which is extremely close to the theoretically optimal values we get. The ratio of second per query is 2.56/2.88 = 1.36, which is approximately equal to the ratio of resources assignment 6/4 = 1.5. In addition, we measured that the relative query rate difference between these two jobs are around 5.614% on average within 10%, which satisfies fair-time inference.

b) When a new job is added to the cluster, it takes about **3 seconds** to fetch dataset, reschedule, and start executing queries of the second job. Note that, The resource allocation is a dynamic process, meaning that the machine allocation is not fixed over the lifetime. At the same time, the batch size of each job is also a critical hyperparameter that affects the resource allocation.

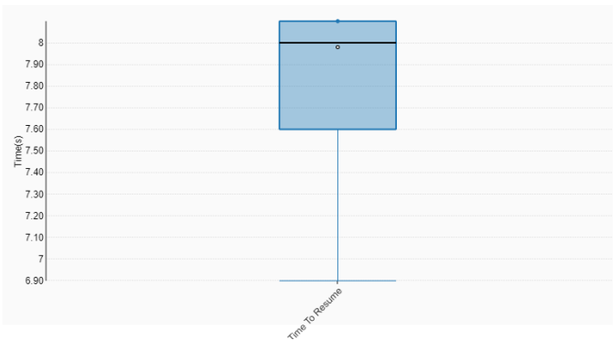
Worker Failure and Resumption

In our design, a worker failure will not block the normal inference operation on other workers. Other workers queries data regardless of worker failure. However, it takes roughly **5.43 seconds** for the ring to timeout and delete the failed worker and reschedule the specific query assigned on the failed worker to other server.



Coordinator Failure, Backup and Resumption

After the failure of the coordinator, it takes about **8 seconds** for the cluster to resume the normal inference operation. It takes 5 seconds to timeout the coordinator, 1 to 2 seconds to transfer files in SDFS, and another 1 second to reschedule the jobs.



Here's the nice dashboard we built using **React** and **TypeScript**: