

Party Distributed Processing

JOHNSON LAM JUSTIN CHEN

Boston University

April 22, 2017

Abstract

In the past few years, machine learning and deep learning have burgeoned from theoretical research to hobbyist computing. Training algorithms require extremely large datasets and users without access to large amounts of resources must purchase them from cloud services, such as AWS, which can be expensive over time. We propose an alternative solution using commodity hardware spread across the internet to tackle this barrier to entry. This could allow any user the opportunity to do computationally intensive work.

I. INTRODUCTION

Open source software libraries, such as TensorFlow, Torch7, and Theano, have made machine learning more accessible than ever before. However, training learning algorithms require massive datasets in the order of several gigabytes to several hundred of gigabytes, which require expensive, specialized GPUs. These models, however, can be just as effectively trained on a large number of CPUs. Cloud services provide these resources at a cost and their interfaces can be difficult to navigate for new users.

The average computer is typically under-utilized or idle in a typical day. These available resources could be altruistically allocated to agreed party members for distributed computation. We propose an affordable and easy-to-use distributed data computation paradigm for processing datasets across a party of commodity hardware connected by the internet. Our goal is to lower the cost of entry and simplify the interface users interact with. We use the MapReduce [1] architecture as a guide for our own design and implement a distributed word counter as a proof-of-concept on BU's csa servers.

II. DESIGN

We propose utilizing hardware owned by a trusted network, like a set of computers owned by a group of friends or a trusted organization. This can even be extended to locally connecting a group of machines that a user accumulates over the years. Our current design utilizes a "master" cloud server that will coordinate work across the party, combine intermediate results from each worker, and return the final result to the client that submitted. Each computer in this network can act a client wanting to do work or as a server available to do work for others.

Each computer begins by stating their available resources (cores, RAM, etc.) and their restrictions (bandwidth) to the master server when it comes online. At any point in time a computer on the network can make a request to the master server that it'd like to do some work on a dataset and pass the dataset to the master. Similar to how the master node took care of distributing work in MapReduce [1], the master node in our system will distribute work among the available servers in the network based on each computer's available resources and restrictions. The bottleneck of this design will be the network speed and reliability of each computer. Given this restriction, distribution of work will be in smaller chunks such that sending data over the network will not be too cumbersome. The master server will be responsible for handling network failures and unresponsive or slow workers in the same way as if these servers were in the same data center.

Our current work achieves the above-mentioned by distributing work for counting words in a large document, except it does not take into account the available resources and restrictions for a worker. A worker can come and go at any time, and work will be distributed accordingly to this dynamic network inspite of worker failure. The master keeps track of workers currently available to it by sending periodic heartbeats and removes workers from the list who are unresponsive. When finished, the master returns the final word count to the worker that submitted the request. This implementation is currently running on csa1, csa2, csa3, and kd-sesa with csa1 being the master server.

III. ARCHITECTURE

Our program is written in Go and utilizes the "net/http" package to communicate between the worker and master. The master has four endpoints: join, heartbeat, job_request, and job_chunk. The worker has three endpoints: heartbeat, process_chunk, and finish_job.

When workers first come online, they notify the master they are joining the network. The worker then idles and waits for either a heartbeat or a job from the master and responds to the master accordingly. When a worker submits a request to the master, the user specifies a URL containing the text document and a name for the document. The worker contributes its resources for work its given to do by the master. It will then be notified of the final result of its request when the master is finished distributing work.

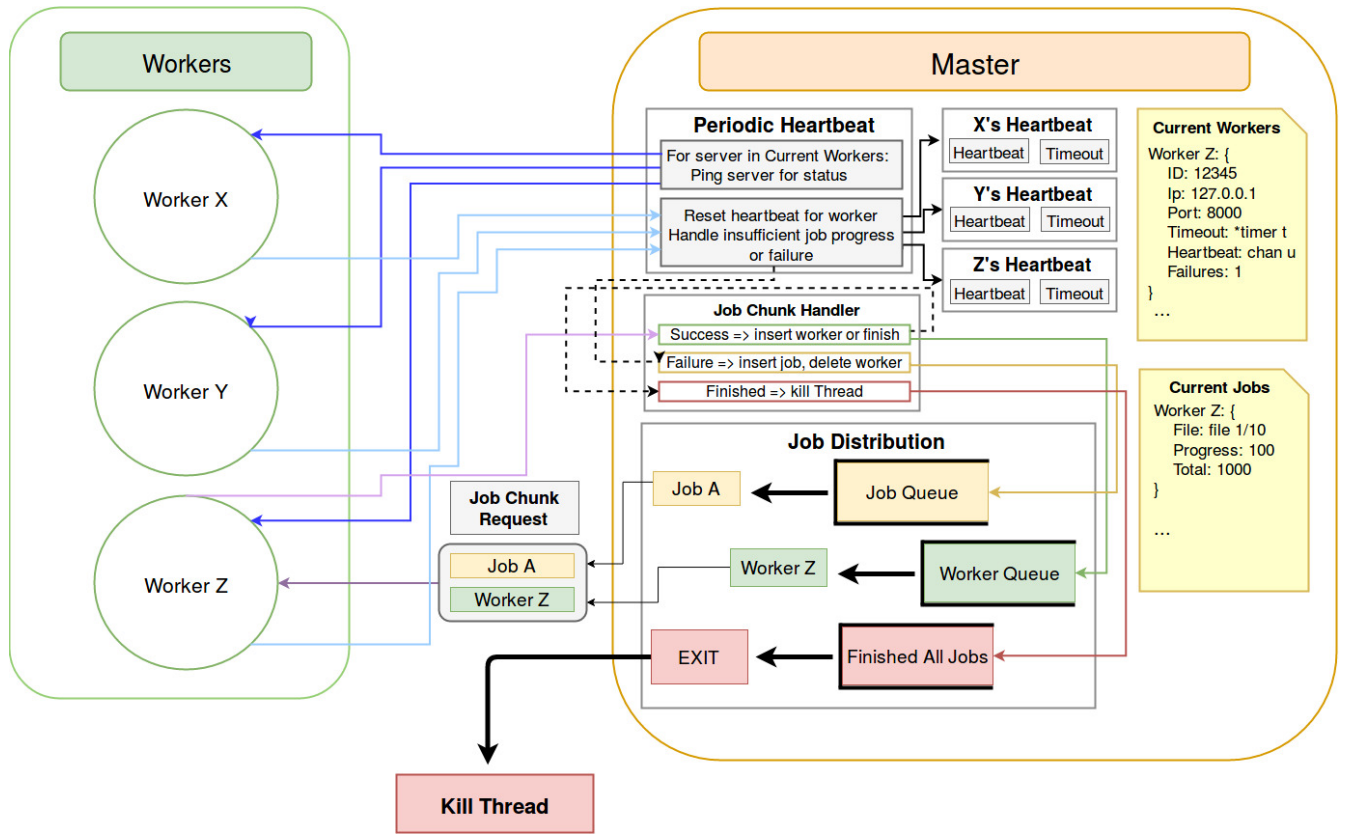


Figure 1: Heartbeating and Job Distribution Among Workers

The master handles all of the coordination on its fleet of machines. When it first boots up, it starts a thread to send periodic heartbeats to workers that have joined the party. When the master sees a new worker joining, it adds it to its map of current workers and starts a thread responsible for making sure it's responsive by using a timer. If

a worker does not respond to a heartbeat, this timer will not be reset and will eventually fire. When the master receives a job request, it makes a GET request on the URL provided, downloads the page onto its machine, and splits the file up into 1000 line file chunks. It then distributes the file chunks to the available workers, combines their results, and returns it to the submitter. The key components of our work are heartbeating and distribution of work for a request. We use heartbeating as a form of checking on the availability of workers and checking on worker progress on a file chunk. The former case is simple: if the worker is unresponsive after three timeouts, delete the worker. In the latter case, if the worker is unresponsive or fails to make satisfiable progress, delete the worker and add the file chunk back into the queue of jobs that need to be processed. The progress check has not been implemented yet.

We distribute work by creating a separate thread with two channels and treat them as queues, one for jobs to be worked on and one for available and idle workers to process these jobs. A third channel is used to exit this thread when all jobs have been completed. We insert the file chunks and workers into their respective queues at the beginning of a request. The thread blocks until it either receives a job or notification that all jobs have finished. In the first case, it then blocks until a worker becomes available, at which point a request is sent and the thread loops back to the beginning. In the second case, a handler for job results will indicate to the thread that all jobs have finished and the thread will proceed to exit. As mentioned earlier, a job is reinserted back into its queue if unsatisfactory progress is made by the worker and the worker is deleted. If a new worker joins in the midst of a request being processed, it adds itself to the worker queue and will be given a job when one becomes available. When a worker returns the result for a job, the master adds it back into the worker queue. The insertion of jobs and workers into the queue is done on separate thread to ensure that it does not block the master. Figure 1 details the architecture of heartbeating and job distribution for a job request.

IV. FUTURE WORK

Although we use the MapReduce [1] architecture as a guide for building out our distributed word counter, much of what is described below is borrowed from the Borg [2] paper from Google. The overarching goal is to distribute the processing of large datasets over a set of machines and achieve the highest efficiency to enable hobbyist to do computationally intensive work. Benchmarks need to be conducted to show the efficiency and savings of this.

i. Functionality

Our current implementation handles only one request at a time. If an active job is being processed, any request submission will be rejected until it is finished. Future work can be done to handle concurrent requests, or a queue can be made such that requests can be processed one at a time in the order they were received. Introducing this complexity will lessen the burden for application users to handle job submit rejections.

The system also does not take into account the progress workers make- it only checks whether a worker is unresponsive when given a job. Worker progress will be essential in making sure workers finish in a timely manner. Further discussion for what to do with slow workers will be needed. Finally, this project is only functional for counting words. Processing large datasets will be the next steps in accomplishing the motivation behind the project.

ii. Architecture

Our system still has major flaws to be worked on. At the basic level, network errors are not handled properly and it is assumed that packets make it to their target. The project was initially written using the http package, but future iterations may use the RPC package to more easily handle such errors. When the functionality of processing large datasets is built out, the system will need to handle the requirement of available resources and restrictions to choose which workers are most suitable for handling the job. Although our system is built on the idea that workers come and go, the master must be fault-tolerant. One way to achieve this is to use replicated state machines and build the master on top of the Raft system done in Lab 2. If the master goes down, another

server, that is not a worker, can become the master. This introduces a new range of complexity, such as workers being aware of where the new master is and how the new master can pick up where the last one left off and continue work for a given job. This discussion will be left for the next iteration. Finally, for this system to be used in production, security features will need to be added such that only authenticated users can join a party.

REFERENCES

- [1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [2] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.