CS434/534: Topics in Network Systems

Cloud Storage: GFS; Distributed Processing using MapReduce

Yang (Richard) Yang
Computer Science Department
Yale University
208A Watson
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/

Acknowledgement: slides include content from Ennan Zhai
Outline

- Admin and recap
- Basic cloud data center (CDC) network infrastructure
  - Background, high-level goals
  - Traditional CDC vs the one-big switch abstraction
  - VL2 design and implementation: L2 semantics, VLB/ECMP
  - Load-aware centralized load balancing (Hedera)
  - Distributed load balancing by end hosts: MP-TCP
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow (e.g., Web apps)
  - Coarse-grained dataflow (e.g., “big” data analytics)
Admin

- PS1 submission

- Office hours on projects
  - Thursday: 1:30-3:30 pm
Algorithm

- Each ACK on subflow $r$, increase the window $w_r$ by $\min(a/w_{total}, 1/w_r)$.
- Each loss on subflow $r$, decrease the window $w_r$ by $w_r/2$.

Here

$$a = \hat{w}_{total} \frac{\max_r \hat{w}_r/RTT_r^2}{\left(\sum_r \hat{w}_r/RTT_r\right)^2},$$  \hspace{1cm} (5)$$

$w_r$ is the current window size on path $r$ and $\hat{w}_r$ is the equilibrium window size on path $r$, and similarly for $w_{total}$ and $\hat{w}_{total}$.  

Recap: MPTCP at Different Load

![Diagram showing Relative MPTCP Throughput vs. Connections per host, with shaded areas indicating Underloaded, Overloaded, and Sweet Spot regions.].
Recap: Fine-grained Dataflow

FB architecture

Noria program
Outline

- Admin and recap
- Basic cloud data center (CDC) network infrastructure
  - Background, high-level goals
  - Traditional CDC vs the one-big switch abstraction
  - VL2 design and implementation: L2 semantics, VLB/ECMP
  - Load-aware centralized load balancing (Hedera)
  - Distributed load balancing by end hosts: MP-TCP
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow (e.g., Web apps)
  - Coarse-grained dataflow (e.g., “big” data analytics)
Discussion: Examples of "Big" Data Analytics
Challenges of “Big” Data Analytics

- How to store a huge amount of data?
- How to process and extract something from the data?
- How to handle availability and consistency?
- How to preserve the data privacy?
- ...

...
Discussion: DA Software Architecture

- What may the software architecture look like?
Basic Google DA Software Architecture

- How to store a huge amount of data?
  - Google File System & BigTable

- How to process and extract something from the data?
  - MapReduce

- How to handle multiple availability and consistency?
  - Paxos

- How to preserve the data privacy?
Basic Google DA Architecture

Google Applications, e.g., Gmail and Google Map

MapReduce

BigTable

Google File System (GFS)
Another Open Source Example: Apache

Apache Applications

Apache MapReduce  Apache HBase

Hadoop Distributed File System (HDFS)
Outline

- Admin and recap

- Basic cloud data center (CDC) network infrastructure
  - Background, high-level goals
  - Traditional CDC vs the one-big switch abstraction
  - VL2 design and implementation: L2 semantics, VLB/ECMP
  - Load-aware centralized load balancing (Hedera)
  - Distributed load balancing by end hosts: MP-TCP

- Cloud data center (CDC) applications/services
  - Fine-grained dataflow (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics -- aka big data)
    - Data storage: Google File System (GFS)
The Google File System [SOSP’03]: Goals

- GFS aims to offer file services for google applications:
  - Scalable distributed file system
  - Designed for large data-intensive applications
  - Fault-tolerant despite running on commodity hardware
Workload Assumptions

- GFS was originally built for storing Web pages of a Web site as a file for Web indexing:
  - Files are huge. n-GB/TB files are norm
  - Files are mostly read, often sequential
  - Random writes in a file are quite rare. Most files are appended, not overwritten
  - Hundreds of processes append to a file concurrently

- Workload in which component failures are norm:
  - File system = thousands of storage machines
GFS Architecture

**GFS cluster:** \( N \) **Chunkservers** + 1 **Master**

![Diagram showing GFS architecture with chunk servers and master node]

- **Chunk servers:** Store data chunks.
- **Master node:** Manages file metadata and chunk locations.
- **File system metadata:** Maps files to chunks.
- **Data Storage:** fixed-size chunks. Chunks replicated on several systems.
Each rack has a master
GFS Architecture
GFS Architecture
GFS Architecture
GFS Architecture

Metadata 1 2 3

Master Chunkserver1 Chunkserver2 Chunkserver3 Chunkserver4 Chunkserver5

GFS
GFS Architecture
GFS Architecture
A File in GFS

A File is made of 64MB (why so big?) chunks

Each chunk is assigned a 64-bit global Id by master, replicated for fault-tolerance: 3 default, more for popular content

Chunks live as Linux files on chunkservers

The master manages the file system namespace

A File in GFS

A File is made of 64MB (why so big?) chunks

Each chunk is assigned a 64-bit global Id by master, replicated for fault-tolerance: 3 default, more for popular content

Chunks live as Linux files on chunkservers

The master manages the file system namespace
Outline

- Admin and recap
- Basic cloud data center (CDC) network infrastructure
  - Background, high-level goals
  - Traditional CDC vs the one-big switch abstraction
  - VL2 design and implementation: L2 semantics, VLB/ECMP
  - Load-aware centralized load balancing (Hedera)
  - Distributed load balancing by end hosts: MP-TCP
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics -- aka big data)
    - Data storage: Google File System (GFS)
      - Architecture
      - Read
**GFS Read**

- The ID for each chunk
- IP address for each chunk
- Master can choose the nearest replicas for the client

Metadata:
- /usr/data/foo -> 1, 2, 4
- /usr/data/bar -> 3
GFS Read

Metadata:
/usr/data/foo -> 1,2,4
/usr/data/bar -> 3
GFS Read

Metadata:
/usr/data/foo -> 1, 2, 4
/usr/data/bar -> 3
Outline

- Admin and recap
- Basic cloud data center (CDC) network infrastructure
  - Background, high-level goals
  - Traditional CDC vs the one-big switch abstraction
  - VL2 design and implementation: L2 semantics, VLB/ECMP
  - Load-aware centralized load balancing (Hedera)
  - Distributed load balancing by end hosts: MP-TCP
- Cloud data center (CDC) applications/services
  - Fine-grained dataflow (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics -- aka big data)
    - Data storage: Google File System (GFS)
GFS Write

- Less frequent than reading
- GFS write focuses on appending (what if you want to just overwrite a part?)

Discussion: how you may design the write (key issues to address in write)?
GFS Write

- **Two phases**
  - Data flow (phase 1): to replicate the chunk
  - Control flow (phase 2): to order and commit the write
GFS Write

• Phase 1: data flow (Deliver data but do not write to the file)

• A client is given a list of replicas
• Client writes to the closest replica
  - Pipeline forwarding (why not client sends to all replicas?)
• Chunkservers store this data in a cache
• Client waits for replicas to ack. receiving data
GFS Write

- Phase 2: Write data (Commit it to the file)

- Commitment done by the primary chunk server

- Master grants a chunk lease to one of the replicas, who becomes the primary replica chunkserver
  - Primary can request lease extensions, if needed
  - Master increases the chunk version number and informs replicas if primary changes
GFS Write

- Phase 2: Write data (Commit it to the file)

- After getting ack from phase 1, client sends a write request to the primary (master provides primary in the write request reply)

- The primary is responsible for serialization of writes (applying then forwarding)

- Once all ack. have been received, the primary ack. the client
GFS Write: Summary

- **Data flow (replication)**
  - Client to chunkserver to chunkserver to chunkserver ...
  - Ordering does not matter

- **Control flow (commit):**
  - Client to primary (sequencer) to all secondaries
  - Ordering maintained
Summary of GFS Insights

- Separation of data and metadata
- Huge files -> 64 MB for each chunk -> fewer chunks
  - Reduce client-master interaction and metadata size
- Replicas are used to ensure availability
- Master can choose the nearest replicas for the client
- Read and append-only makes it easy to manage replicas
- Separation of data flow and control flow in write to achieve high concurrency, but still ordering

- Adopted by other distributed file systems such as Hadoop Distributed File System (HDFS)
Outline

- Admin and recap

- Basic cloud data center (CDC) network infrastructure
  - Background, high-level goals
  - Traditional CDC vs the one-big switch abstraction
  - VL2 design and implementation: L2 semantics, VLB/ECMP
  - Load-aware centralized load balancing (Hedera)
  - Distributed load balancing by end hosts: MP-TCP

- Cloud data center (CDC) applications/services
  - Fine-grained dataflow (e.g., Web apps)
  - Coarse-grained dataflow (e.g., data analytics)
    - Data storage: Google File System (GFS)
    - Data processing: MapReduce
Processing Example Problem

- A toy problem: The word count
  - ~ 10 billion documents
  - Average document’s size is 20KB \(\Rightarrow\) 10 billion docs 
    = 200TB (how many chunks?)
Processing Example: Naïve Solution

- A toy problem: The word count
  - ~ 10 billion documents
  - Average document’s size is 20KB => 10 billion docs = 200TB

```python
for each document d
    for each line in d
        for each word w in line
            word_count[w]++;
```
A toy problem: The word count

- ~ 10 billion documents
- Average document's size is 20KB => 10 billion docs = 200TB

```c
// parallel for each chunk
for each chunk c
  for each document d in c
    for each line l in d
      for each word w in line
        word_count[w]++;
```

Problem: need to merge results from each chunk.
How should the result from each chunk be partitioned (called shuffled) to mergers?
MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming language like LISP

\[
\text{Map}(k,v) \rightarrow (k',v') \\
\text{Reduce}(k',v'[]) \rightarrow v''
\]

- Users implement interface of two primary methods:
  - 1. Map: \(<key1, value1> \rightarrow <key2, value 2>\)
  - 2. Reduce: \(<key2, value2[ ]> \rightarrow <value3>\)

- After map phase, all the intermediate values for a given output key are combined together into a list and given to a reducer for aggregating/merging the result.
Discussion: MapReduce Realization

MapReduce System Components:
- **Master**: Controls the overall execution.
- **Chunkservers**: Store and process data chunks.
- **Input**: Data source for MapReduce.
- **Map**: Processes input data, producing intermediate output.
- **Reduce**: Groups and aggregates intermediate output.
- **Output**: Final output of processed data.

**GFS** (Google File System) provides scalable, distributed storage for MapReduce tasks.

**MapReduce Phases**:
- **Map**: Converts input data into key-value pairs.
  \[ \text{Map}(k,v) \rightarrow (k',v') \]
  - Groups by key \( k' \)

- **Reduce**: Aggregates data by key.
  \[ \text{Reduce}(k',v'[]) \rightarrow v'' \]

**Execution Flow**:
1. Input data is processed by the **Map** phase.
2. Intermediate results are grouped by key by the **Reduce** phase.
3. The aggregated data is written to the **Output**.

**Master and Chunkserver Setup**:
- **Master** initializes the job and distributes tasks.
- **Chunkservers** (Chunkserver1 to Chunkserver5) execute MapReduce tasks on data blocks.
MapReduce Architecture

• Two core components
  - JobTracker: assigning tasks to different workers
  - TaskTracker: executing map and reduce programs
MapReduce Architecture

HDFS + Hadoop MapReduce
MapReduce Architecture

- Document
  - Job Tracker
  - Task Tracker
  - Task Tracker
  - Task Tracker
  - Task Tracker
  - Task Tracker

GFS

- Master
- Chunkserver1
- Chunkserver2
- Chunkserver3
- Chunkserver4
- Chunkserver5
MapReduce Architecture
Map Phase (On a Worker)

Count the # of occurrences of each word in a large amount of input data

```
Map(input_key, input_value) {
    foreach word w in input_value:
        emit(w, 1);
}
```
Map Phase (On a Worker)

Input to the Mapper

(3414, ‘the cat sat on the mat’)
(3437, ‘the aardvark sat on the sofa’)

Count the # of occurrences of each word in a large amount of input data

Map(input_key, input_value) {
    foreach word w in input_value:
        emit(w, 1);
}

Output from the Mapper

(‘the’, 1), (‘cat’, 1), (‘sat’, 1), (‘on’, 1),
(‘the’, 1), (‘mat’, 1), (‘the’, 1), (‘aardvark’, 1),
(‘sat’, 1), (‘on’, 1), (‘the’, 1), (‘sofa’, 1)
Grouping/Shuffling

- After the Map, all the intermediate values received at a reducer for a given intermediate key are combined together into a list (how to efficiently combine?)

Mapper Output received at a reducer

- (‘the’, 1),
- (‘cat’, 1),
- (‘sat’, 1),
- (‘on’, 1),
- (‘the’, 1),
- (‘mat’, 1),
- (‘the’, 1),
- (‘aardvark’, 1),
- (‘sat’, 1),
- (‘on’, 1),
- (‘the’, 1),
- (‘sofa’, 1)

Reducer Input

- aardvark, 1
- cat, 1
- mat, 1
- on [1, 1]
- sat [1, 1]
- sofa, 1
- the [1, 1, 1, 1]
Reduce Phase (On a Worker)

Add up all the values associated with each intermediate key:

```plaintext
Reduce(output_key, intermediate_vals) {
    set count = 0;
    foreach v in intermediate_vals:
        count += v;
    emit(output_key, count);
}
```
Reduce Phase (On a Worker)

Input to Reducer

```
(aardvark, 1)
cat, 1
mat, 1
on [1, 1]
sat [1, 1]
sofa, 1
the [1, 1, 1, 1]
```

Add up all the values associated with each intermediate key:

```
Reduce(output_key, intermediate_vals) {
    set count = 0;
    foreach v in intermediate_vals:
        count += v;
        emit(output_key, count);
}
```

Output from Reducer

```
(‘the’, 4), (‘sat’, 2), (‘on’, 2), (‘sofa’, 1),
(‘mat’, 1), (‘cat’, 1), (‘aardvark’, 1)
```
Map + Reduce

Fig assumes a single mapper, single reducer.
What if multiple mappers and multiple reducers?
High-Level Picture for MR
Let’s use MapReduce to help Google Map

We want to compute the average temperature for each state
Let’s use MapReduce to help Google Map

We want to compute the average temperature for each state
Let's use MapReduce to help Google Map
Let's use MapReduce to help Google Map
Let's use MapReduce to help Google Map

MAP OUTPUT: <K2, V2>

- Key: MP, Value: 77
- Key: MP, Value: 76
- Key: CG, Value: 70
- Key: CG, Value: 72
- Key: CG, Value: 75
- Key: OR, Value: 69
- Key: OR, Value: 71
- Key: OR, Value: 76
Let’s use MapReduce to help Google Map

SHUFFLE OUTPUT: <K2, List<V2>>
Let's use MapReduce to help Google Map

REDUCE INPUT: <K2,List<V2>>

- **MP**: 77, 72, 76
- **CG**: 72, 75, 70
- **OR**: 69, 71, 76
Let's use MapReduce to help Google Map

REDUCE OUTPUT: <K3, V3>
Exercise: PageRank
[Sergey Brin and Larry Page, 1998]

- Problem: many Web pages may contain the searched key word (e.g., Yale), how to rank the pages when displaying search results?

- Basic PageRank™ idea
  - The 10-90 rule
    - 10% of the time surfer types a random page
    - 90% of the time surfer clicks (follows) a random link on a given page
  - PageRank ranks pages according to frequencies (we call the pageranks) surfer visits the pages
Round-Based PageRank

- Initialize arbitrary page ranks
- Iterative algorithm to simulate visit redistribution
  - Assume current round page rank of page $p_i$ is $PR_c(p_i)$
  - Update next round

$$PR_{new}(x) = 0.1 \left( \frac{1}{N} \right) + 0.9 \sum_{i=1}^{n} \frac{PR_{pre}(p_i)}{C(p_i)}$$
Offline Exercise: PageRank

- What a mapreduce for PageRank looks like?
Discussion

- Issues of MapReduce
Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Hadoop [1] are widely used for processing large datasets. However, these frameworks do not provide fault tolerance, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (e.g., looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, e.g., to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called resilient distributed datasets (RDDs) that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance efficiently. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-