Notes on Cloud Operating Systems: Workflows, Distributed Shared Memory, and Distributed File Systems

cjeongkr@gmail.com

1 Introduction

- Old Computer: HW + OS + Application
- New Computer: distributed HWs + Cloud OS + Cloud Applications

2 Workflows (Distributed Processes): Overview

3 Case Study: MapReduce

4 Distributed Shared Memory: Overview

5 Distributed File Systems: Overview

- distributed file systems:
  - allows multiple process to share data

6 Case Study: Sun Network File System (NFS)

6.1 Overview

- basic idea: **each file server provides a standardized view of its local file system**
- traditional **client-server**-based
- not actually a true file system but *a collection of protocols that together provide clients with a model of a distributed file system*

6.2 NFS architecture

- NFS model is **remote file service**
  - **clients** are offered transparent access to a file system which is managed by **remote server**
  - clients are unaware of the actual location of files
  - *clients* are offered an interface to a file system similiar to POSIX API for files and *server* implements that interface
- NFS is a **remote access model** (cf. **upload/download model**)
  - **remote access model**

- **upload/download model**: client access a file locally after having downloaded it from the server

- nowadays, NFS is incorporated into **VFS (Virtual File System)**, which provides a unified inteface to many different (local or distributed) file systems

6.3 Naming in NFS

- through **mounting** remote file system onto its local file system, transparency is achieved
- server is said to **export** its directory
- **implication!**: *clients do NOT share name spaces*
  - client A sees a file at `/remote/vu/mbox`
  - client B sees the same file at `/work/me/mbox`
  - **file sharing in NFS is hard!** since A and B have different name spaces
  - **solution**: standardize the mount point (e.g. all clients mount remote FS into `/nfsmnt`)

7 Case Study: Coda

8 Case Study: Plan 9

9 Case Study: xFS

10 Case Study: SFS

11 Case Study: GFS (Google File System)

11.1 Overview

- GFS is a scalable file system for large distributed data-intensive applications [1].
- design goals (quite conventional):
  - performance
  - scalability
  - reliability
  - availability
- GFS-specific design goals
  - **component failures are the norm rather than the exception**
    * *sources of failures* are: application bug, OS bug, human error, machine failures, network failures
    * *solution*: constant monitoring, error detection, fault tolerance, automatic recovery
  - **small number of huge files** or **huge number of small files** or both
    * **basic assumption on paramters should be revised**:
      · I/O operations
      · buffer size
      · block size
  - **special access patterns**: most files are mutated by appending new data rather than overwriting existing data

        * random writes are practically non-existent
        * files are mostly sequentially read
            · **large streaming reads**: 100s of KBs (more commonly 1MBs or more)
            · **small random reads**: a few KBs at some arbitrary offsets
    - **co-designing applications w/ the file system API benefits the overall system by increasing flexibility**
    - **high sustained bandwith is more important than low latency**

11.2 GFS API

- **create**, **delete**, **open**, **close**, **read**, **write**
- **snapshot**: creates a copy of a file or directory tree at low cost
- **record append**: allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individial client's append
    - good for merging outputs to the tempory data file (e.g. in MapReduce)
    - no extra synchronization (e.g. locking) is required

11.3 GFS architecture

- a GFS cluster consists of
    - a signle **master**
    - multiple **chunkservers**
    - multiple **clients**
- **files** are divided into fixed-size **chunks**
- **Master**:
    - store **metadata** associated with the chunks (e.g. tables mapping the 4-bit labels to chunk locations)
    - periodically receives updates from each chunkserver ("hear-beat messages")
    - permission for modificiations are handled by finite-time **leases** granted to clients
- **Chunkserver**:
    - store the files, which each individual file broken up to fixed-size chunks (about 64MB)
- **metadata**: there are three types
    - **file and chunk namespaces**: kept persistent
    - **mapping from files to chunks**: kept persistent
    - **locations of each chunk's replicas**: master does not store chunk location persistently; instead, it askes each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster
- **handling metadata (in-memory)**
    - master periodically scan through its entire state in the background for:
        * chunk garbage collection
        * re-replication in the presence of chunkserver failure
        * chunk migration to balance load and disk space usage
    - 64B metadata for each 64MB chunk
- **chunk location**:

11.4 Consistency model

- **file namespace mutation** (e.g. **file creation**) are atomic
- above is achieved by *namespace locking* by master, where atomicity and correctness is guaranteed

12 Case Study: HDFS (Hadoop File System)

12.1 Assumptions and goals

- **hardware failure**
- **streaming data access**
- **large data sets**
- **simple coherency model**: *write-once-read-many* access model for files; a file once created, written, and closed need not be changed
- **"moving computation is cheaper than moving data"**: computatino is more efficient "near" the data
- **portability across heterogeneous HW and SW platforms**

12.2 `NameNode` and `DataNodes`

-

12.3 File system namespace

13 Case Study: IBM GPFS (General Parallel File System)

References

[1] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *The Processings of 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

Contents