COMP5349 – Cloud Computing

Week 6: Distributed Execution: HDFS and YARN

Dr. Ying Zhou
School of Computer Science

Outline

- Distributed Execution Basics
- GFS and HDFS
- MapReduce on YARN

Last Two Weeks

- In Last two weeks we cover basic APIs of MapReduce and Spark framework
  - We run them on single node without much parallelisation
- Spark is a data flow system, data analytic workload is designed as *data* flowing along a sequence of *operations*
  - The original *data structure* and *data flow operations* closely follow the MapReduce design principles
  - But MapReduce's workflow only contains two operations: map and reduce
  - Spark has larger set of APIs
    - A number of map like operations: map, flatMap, mapToPair, filter, etc.
    - A number of reduce like operations: reduceByKey, groupByKey, aggregateByKey, join, etc…
    - Spark is memory based, MapReduce is storage based

High level view of Distributed Execution

- Assumptions:
  - Very large input data
  - A lot of machines that can execute operations in parallel each on a partition of the data
- MapReduce Distributed Execution
  - A lot of machines run mappers on input data in parallel
  - Mapper outputs are shuffled to multiple reducers
  - A lot of machines run reducers on mapper output in parallel
- Spark Distributed Execution
  - A lot of machines run *some* operations in parallel
  - When intermediate result regrouping is needed, outputs are shuffled to machines running next sequences of operations
  - Repeat until an action is executed to return data to driver program
- Large workload is expected to run on multiple machines for an extended period of time

Basic Requirements and Supports

- A way to divide large input data into smaller partitions
- A mechanism to allocate machines to run those operations in parallel and to organize data transfer when needed
- A mechanism to support continuation of the execution when something goes wrong
  - Data reading error
  - Data transmission error
  - Machine execution error
- Basic Supports:
  - A distributed file system that will
    - automatically partition and replicate data
  - A resource scheduling system that will
    - monitor workload of each machine in the system and allocate workload accordingly
    - monitor execution of application and kick in fault-tolerance or optimization mechanism when needed

Systems that are in use

- Distributed File System
  - **Google File System / HDFS**
    - Supported by MapReduce and Spark
  - Many other storage systems supported by various framework
    - HBase, Azure Blob Storage, S3, etc…
- Resource Scheduling System
  - **YARN** (Yet Another Resource Negotiator)
    - Can schedule MapReduce and Spark applications
  - Other scheduling system
    - Early MR1 job tracker for MR
    - Spark standalone, Mesos, etc…

Outline

- Distributed Execution Basics
- **GFS and HDFS**
  - **Overall Architecture**
  - **Read/Write Operation**
  - **HA Strategy**
  - **Chunk Placement Strategy**
- MapReduce on YARN

Distributed File System Basics

- A distributed file systems allow clients to access files on remote servers "*transparently*".
- We are using at least two popular ones in SCS
  - Your home directory is physically located on some file servers.
  - Your home directory is mounted on all Linux servers using NFS
  - Your home directory is mapped as U drive in Windows using Samba
- Constrains
  - No sufficient mechanism in terms of reliability and availability
    - Replica, migration, etc
    - You may have encountered login hiccups like your U drive is not mapped when you login to one of the lab machine…

GFS Design Constraints

- Component failures are the norm
  - 1000s of components (servers)
  - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
  - Monitoring, error detection, fault tolerance, automatic recovery
- Files are huge by traditional standards
  - Multi-GB files are common
  - Billions of objects
- Workload features (Co-design of application and the file system)
  - Most files are mutated by appending new data rather than overwriting existing data.
    - E.g. log data, web crawling data, etc…
  - Files are often read sequentially rather than randomly.

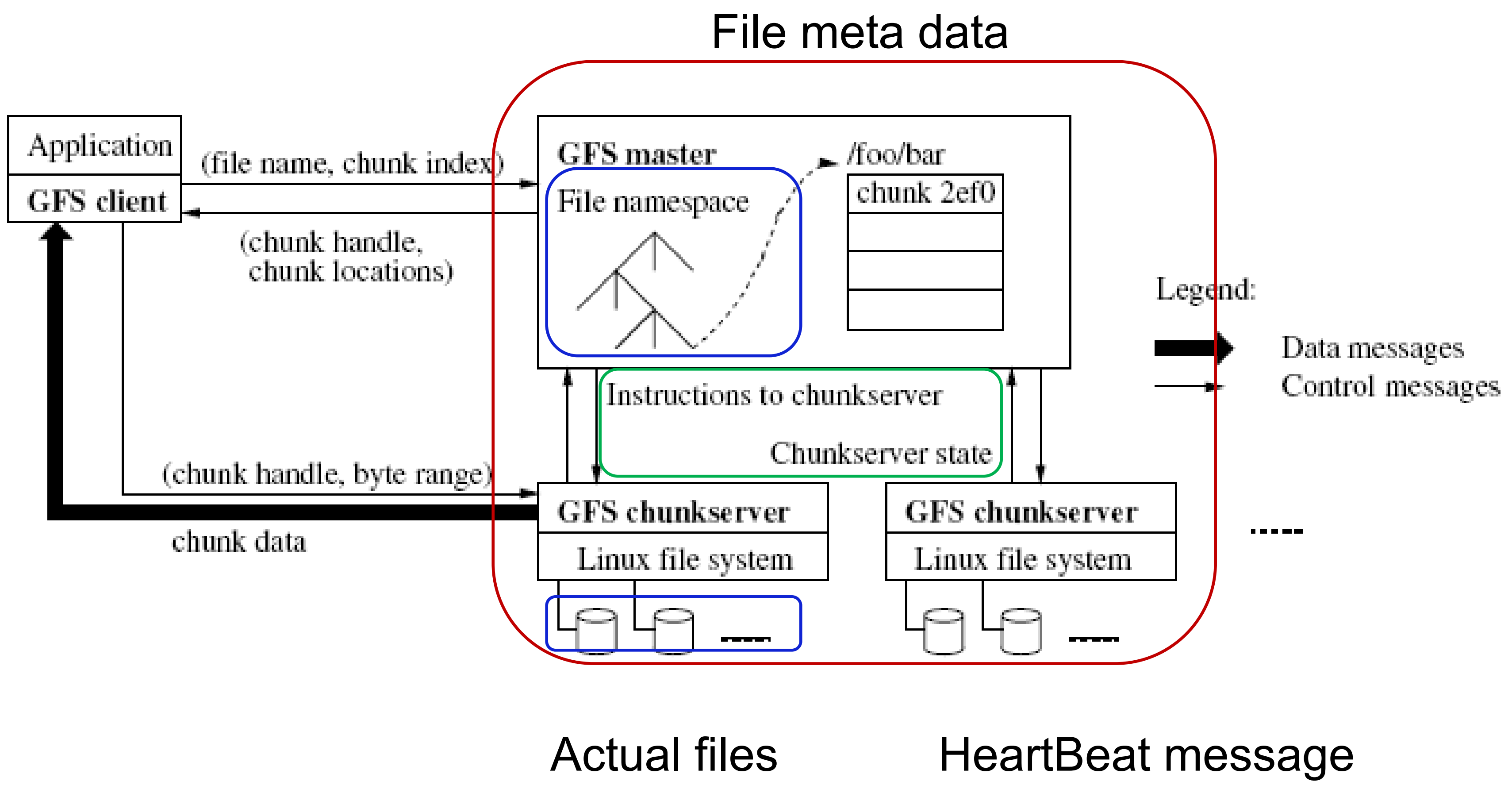
Master/Slave Architecture

- A GFS cluster
  - A **single** *master*
    - Maintains all **metadata**
      - Name space, access control, file-to-chunk mappings, garbage collection, chunk migration
    - Periodically communicates with chunkservers in *HeartBeat* messages.
  - **Multiple** *chunkservers* per master
    - Store actual file data
  - Running on commodity Linux machines

Master/Slave Architecture (cont'd)

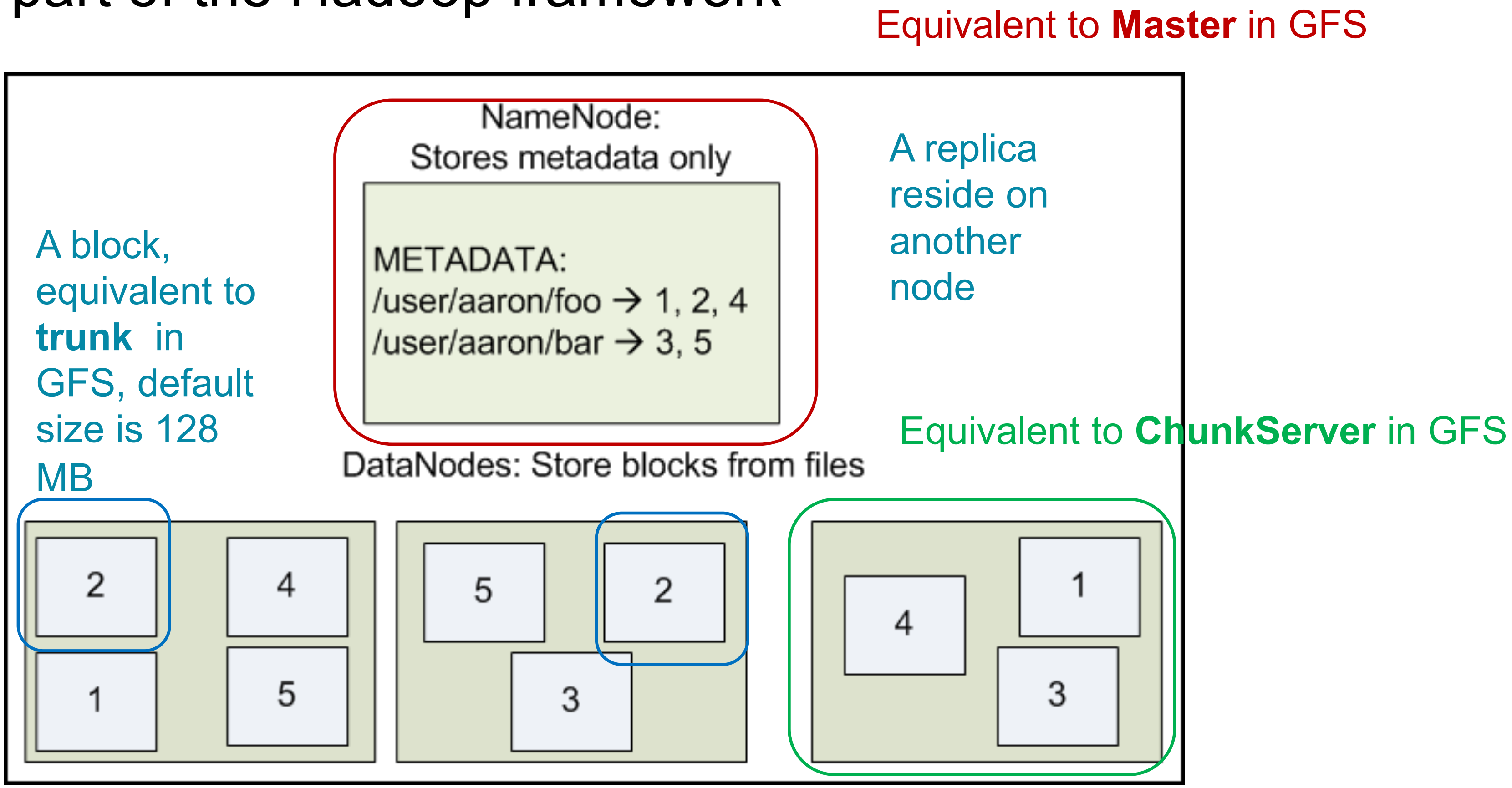
- A file
  - Divided into fixed-sized *chunks*
    - Default chunk size is 64 MB
    - Labeled with 64-bit unique global IDs (chunk handle)
    - Stored at chunkservers
    - Chunk is replicated on multiple chunkservers (by default 3 replicas
      - Files that needs frequent access will have a high replication factor
- GFS clients
  - Consult master for **metadata**
  - Access data from **chunk servers**
  - The equivalent client in HDFS is the hdfs shell script
    - `hdfs dfs –put <sourc> <destination>`

Architecture diagram



HDFS

- Hadoop Distributed File System is the open source implementation of GFS
- It is part of the Hadoop framework



Equivalent to **Master** in GFS
NameNode:
Stores metadata only
METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5
A replica reside on another node
A block, equivalent to **trunk** in GFS, default size is 128 MB
Equivalent to **ChunkServer** in GFS
DataNodes: Store blocks from files
2
4
1
5
5
2
3
4
1
3

http://developer.yahoo.com/hadoop/tutorial/module2.html

GFS v.s. HDFS

GFS	HDFS
Naming	
Master	NameNode
Chunk Server	DataNode
Chunk	Block
Functionality	
Support random write and record append. In practical workload, the number of random write is very small	Support Write-Once-Read-Many workload
HA strategy requires an external monitoring services, such as Chubby lock service for Master recovery	HA configuration is only available in the latest version Secondary name node does part of the job GFS master is supposed to do. It is not a shadow master

http://soit-hdp-pro-1.ucc.usyd.edu.au:50070

Single-Master Design

- Benefits:
  - Simple, master can make sophisticated chunk placement and replication decisions using **global** knowledge
- Possible disadvantage:
  - Single point of failure (***availability***)
  - Bottleneck (***scalability***)
- Solution
  - Replicate master states on multiple machines; Chubby is used to point a master among several shadow masters
  - Fast recovery;
  - Clients use master only for metadata, not reading/writing actual file

Metadata

- Three types:
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of chunk replicas
- All metadata is in memory.
  - Large chunk size ensures small meta data size
  - First two are kept persistent in an *operations log* for recovery.
  - Third is obtained by querying chunkservers at startup and periodically thereafter.
    - Chunkservers may come and go or have partial disk failure
- ***Ask yourself***: In a small Hadoop cluster, you may be able to create directory on HDFS but cannot upload files, what could be the issue?

Operation Log and Master Recovery

- Metadata updates are logged
  - Example updates: create new directory, new files
  - Log replicated on remote machines
- Take global snapshots (checkpoints) to truncate logs
  - Memory mapped (no serialization/deserialization)
- Recovery
  - Latest checkpoint + subsequent log files

Scalability

- Single Master is the only possible bottleneck
  - Minimize Master storage requirement for each file
  - Minimize Master involvement in read/write operation
    - Number of messages
    - Size of messages
- Design
  - Master is only involved at the *beginning* of Read/Write operation
  - Only limited chunk information is transferred
  - Actual data movement does not involve master
- A single master is able to server thousands of chunk servers

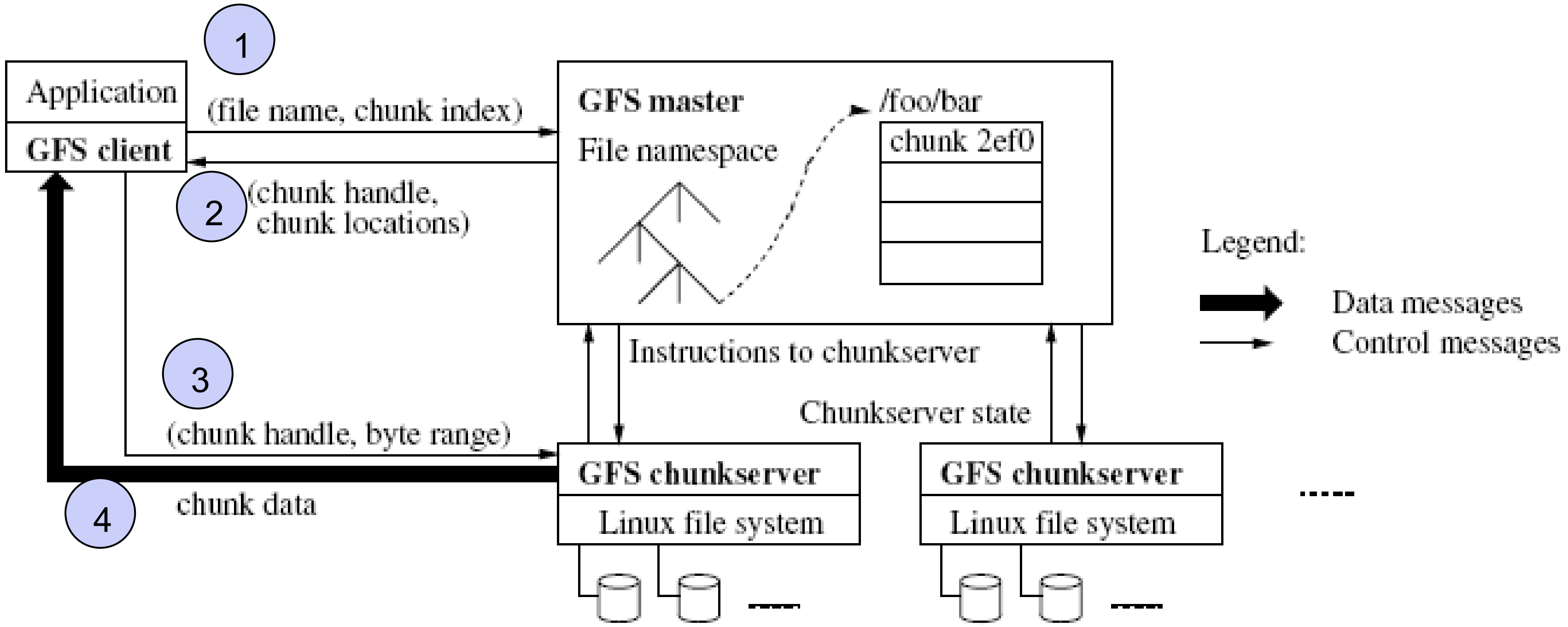
Read and Write in storage system with replication

- Consistency and many other issues
  - Freshness of data, request latency, …
- Many design options depending on targeting apps' requirements
  - How many copies to contact during read/write?
  - How many acknowledgements from copies to wait before replying to client during read/write?
  - Order of applying concurrent write requests
  - Fault tolerance mechanism
  - Recovery mechanism

Overview of GFS read/write operation

- Read from a **single** replica, the **client** *choses* the replica closest to it
- **Synchronous** write to all **replicas**
  - Coordinated by a primary replica
    - Determine the sequence of concurrent writes
    - Request and **wait** for secondary replicas to apply the writes
  - The primary replica is not a fixed role
    - Any replica may become the primary

GFS Read



› Client translates file name and byte offset to chunk index.

› Sends request to master.

› Master replies with chunk handle and ***location of replicas***.

› Client caches this info.

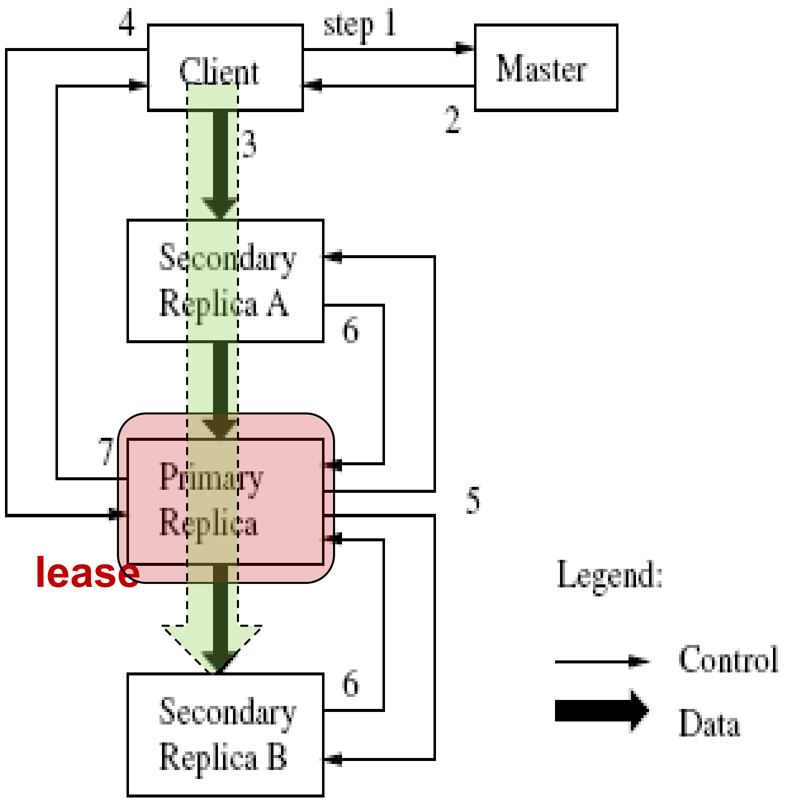
› Client sends request to the ***closest replica***, specifying chunk handle and byte range.

› The chunk server sends chunk data to the client

Lease and Mutation Order

- A mutation is an operation that changes the contents or metadata of a chunk (e.g. a write)
- The master grants a *chunk lease* to a replica
- The replica holding the lease becomes the primary replica; it determines the order of updates to all replicas
- Lease
  - 60 second timeouts
  - Can be extended indefinitely
  - Extension request are piggybacked on heartbeat messages
  - After an old lease expires, the master can grant new leases
- *Any replica may become primary to coordinate the mutation process*

GFS write



1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas.

   Grant a new lease if no one holds one

2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas

   Cached in the client

3. The client pushes the data to all the replicas

4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization.

5. The Primary forwards the write request to all secondary replicas.

6. Secondaries signal completion.

7. Primary replies to client. Errors handled by retrying.

Data Flow

- Data flow is *decoupled* from control flow
- Data is pushed from the client to a chain of chunkservers
  - Each machine selects the "closest" machine to forward the data
    - Client pushes data to chunkserver **S1** that has **replica A** and tells it there are two more chunkservers: **S2** and **S3** in the chain
    - **S1** pushes the data to **S2**, which is closer than **S3** and tells it **S3** is in the chain. **S2** has primary replica.
    - **S2** pushes the data to **S3** and tells it you are the last one. **S3** has **replica B**
- The network topology is simple enough that "distances" can be accurately estimated from IP addresses
- Data is pushed in a pipelined fashion
  - Each trunk server starts forwarding immediately after it receives some data

High Availability (HA) Strategies

- Any server may be down/unavailable at a given time
- HA of the overall system relies on two simple strategies
    - fast recovery
        - Restore states quickly when a server (master or chunk) is back to life
    - Replication
        - Putting extra copies of data

Fast Recovery

- **Key Principle**: Small meta data
- Chunkservers maintain
  - Checksums for 64kb blocks of data for **data integrity check**
  - Replica/chunk version number for **data freshness check**
- Master maintain
  - Data information (owner, permission, mapping, etc)
  - Memory image and short operation log enable fast recovery
- It takes only a few seconds to read this metadata from disk before the server is able to answer queries.
- Master may wait a little longer to receive all chunk location information.

Master Replication

- Master operation log and checkpoints are replicated on multiple machine.
- If the master fails, monitoring infrastructure outside GFS starts a new master process elsewhere (Chubby lock service which utilizes Paxos algorithms to elect a new master among a group of living shadow masters).
- Shadow masters provides read-only access when the primary master is down.

Chunk replica

- Chunk replica is created within the cluster
- Master is responsible for chunk replica creation and placement
  - Decide where to put replica
  - Decide if a chunk needs new replica
  - Decide if a replica needs to migrate to a new location

Chunk Replica Placement

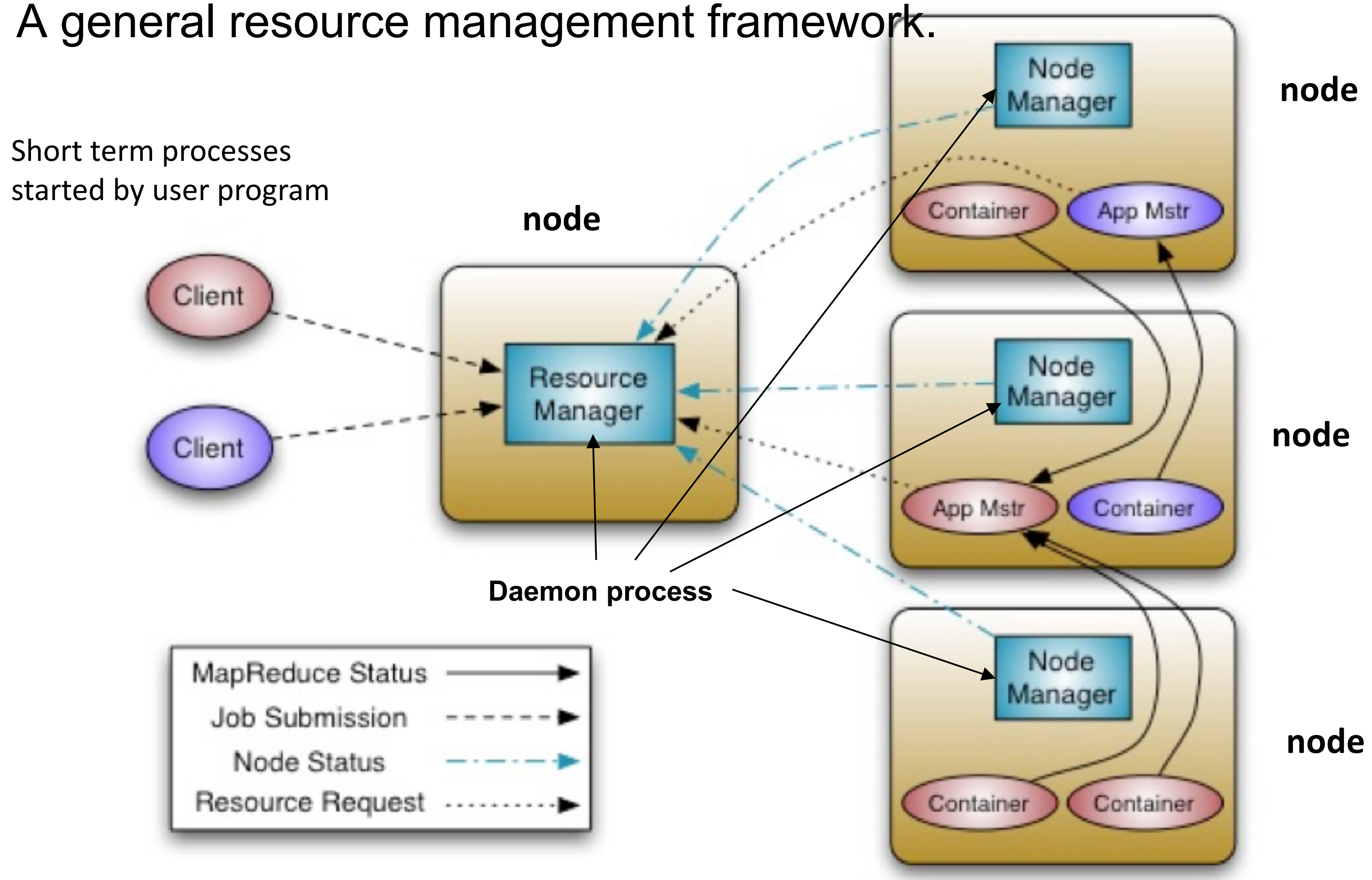
- Goal
  - Maximize data reliability and availability
  - Maximize network bandwidth
- Need to spread chunk replicas across machines and racks
  - Read can exploit the aggregate bandwidth of multiple racks (read only needs to contact one replica)
  - Write has to flow through multiple racks (write has to be pushed to all replicas)

Outline

- Distributed Execution Basics
- GFS and HDFS
- YARN
  - Basic component of YARN
  - YARN scheduling
  - Data locality constraints
  - Hadoop Straggler handling mechanism

YARN Framework

- Yet Another Resource Negotiator
  - A general resource management framework.



YARN Concepts

- YARN's world view consists of *applications* requesting *resources*
- **Resource Manager** (one per cluster)
  - "Is primarily, a *pure scheduler*. In essence, it's strictly limited to arbitrating available resources in the system among the competing applications"
- **Node Manager** (one per node)
  - Responsible for managing resources on ***individual*** node
- **Application Master** (one per application)
  - Responsible for requesting resources from the cluster on behalf of an application
  - Monitor the execution of application as well
  - Framework specific
    - MapReduce application's AM is different to Spark application's AM

YARN Resource Concepts

- Resource Request
  - In the current model, resource requirements are expressed mainly in Memory and/or CUP cores
  - Depends on configured scheduler
- Container
  - Resource allocation is in the form of container
  - A Container grants rights to an application to use a specific amount of resources (memory, cpu etc.) on a specific host.
  - MR framework uses containers to run map or reduce tasks
    - Each container runs **a** map or **a** reduce task
  - Spark users containers to run operations.
    - Each container may run many operations
  - Container is launched by **NodeManager**

Resource Allocation: MapReduce

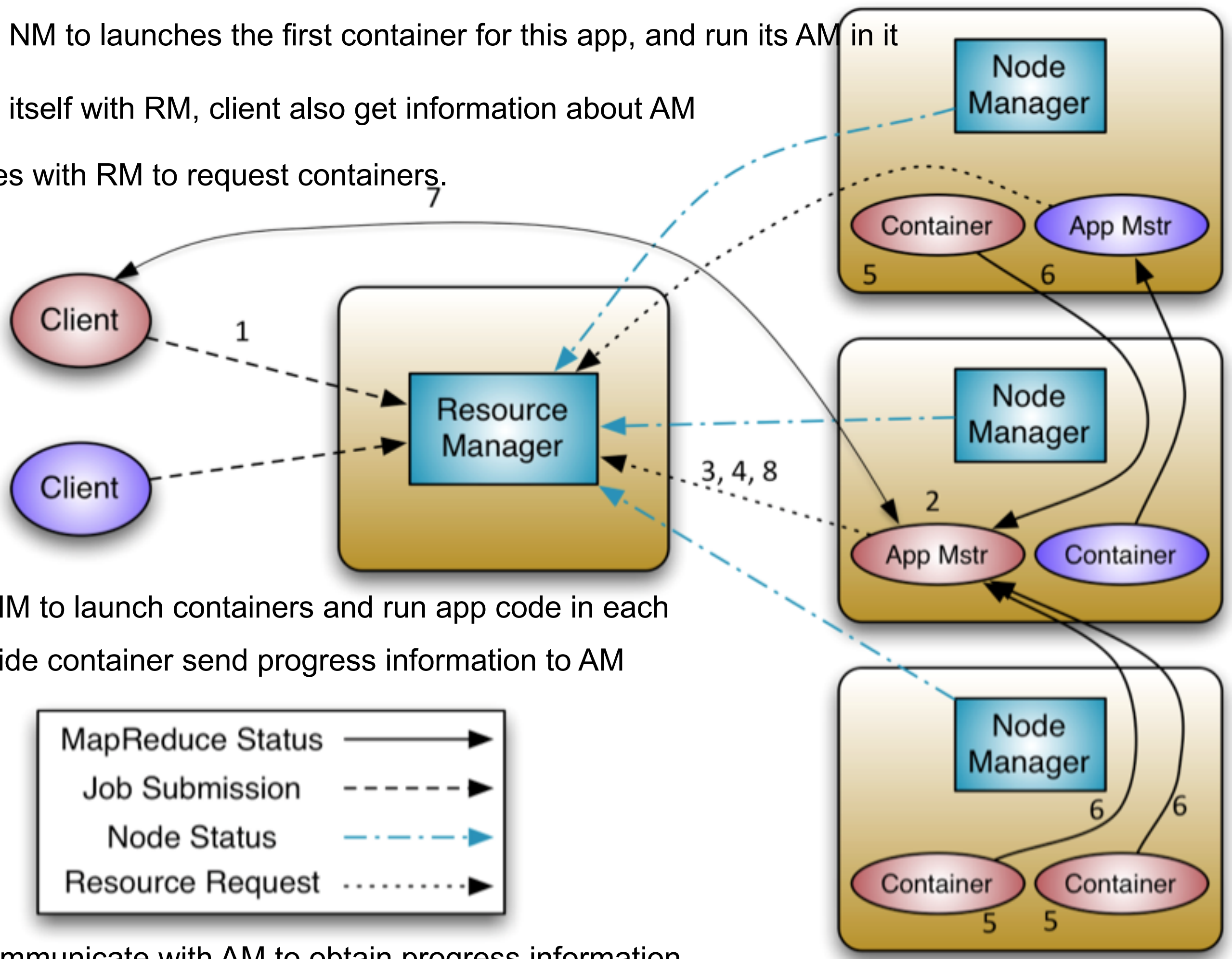
- MapReduce Framework's resource requirements can be worked out based on workload features
  - The same application running on the same data set has the same resource requirements
- The number of map tasks depends on the number of partitions of the input data
- The number of reduce tasks is specified by developer based on estimation of the map output size
- There is always an AM that need to run in a container
- Example: A MapReduce Job with 10 map tasks and 3 reduce tasks would need 14 containers. The first container runs its AM, which would request 10 containers for map tasks (mappers) and 3 containers for reduce tasks (reducers).
- Containers are requested at different time.
  - Containers for map task will be requested early and finishes early
  - Containers for reduce task will be requested late and also finishes late

Resource Allocation: Spark

- Spark's resource requirements are more flexible and are specified by developers
- Containers are requested at the same time and stay through out the life time of the application
- Each container can run different number of operations
- Developers can experiment with various setting to optimize performance

YARN Walkthrough

1. Client submit application and resource requirement
2. RM contacts NM to launches the first container for this app, and run its AM in it
3. AM registers itself with RM, client also get information about AM
4. AM negotiates with RM to request containers.



5. AM contact NM to launch containers and run app code in each
6. App code inside container send progress information to AM

7. Client can communicate with AM to obtain progress information

8. AM deregisters with RM and all containers can be removed

YARN and MR memory configuration

- YARN needs to know
  - The total amount of memories YARN can use
  - How to break up the total resources into containers
    - Minimum container size, maximum container size, etc.
- **ApplicationMaster** for MapReduce application needs to know
  - Memory requirement for Map and Reduce tasks
    - We usually assume Reduce tasks need more memory
  - Each task is running on a JVM, the JVM heap size needs to be set as well

Memory Configuration Example

- Node capacity
  - 48G Ram, 12 core
- Memory allocation
  - Reserve 8G Ram for OS and others (e.g. HBase)
  - YARN can use up to 40G
  - Set Minimum container size to 2G
    - Each node may have at most 20 containers
  - Allow each map task to use 4G and reduce task to use 8G
    - Each node may run 10 map tasks or 5 reduce task or a combination of the two

Resource Scheduling

- Resource Scheduling deals with the problem of which requests should get the available resources when there is a queue for resources
- Early MRv1 uses FIFO as default scheduling algorithm
  - Large job that comes first may starve small jobs
  - Users may experience long delay as their jobs are waiting in queue
- YARN is configured to support shared multi-tenant cluster
  - Capacity Scheduler
  - Fair Scheduler
  - FIFO Scheduler

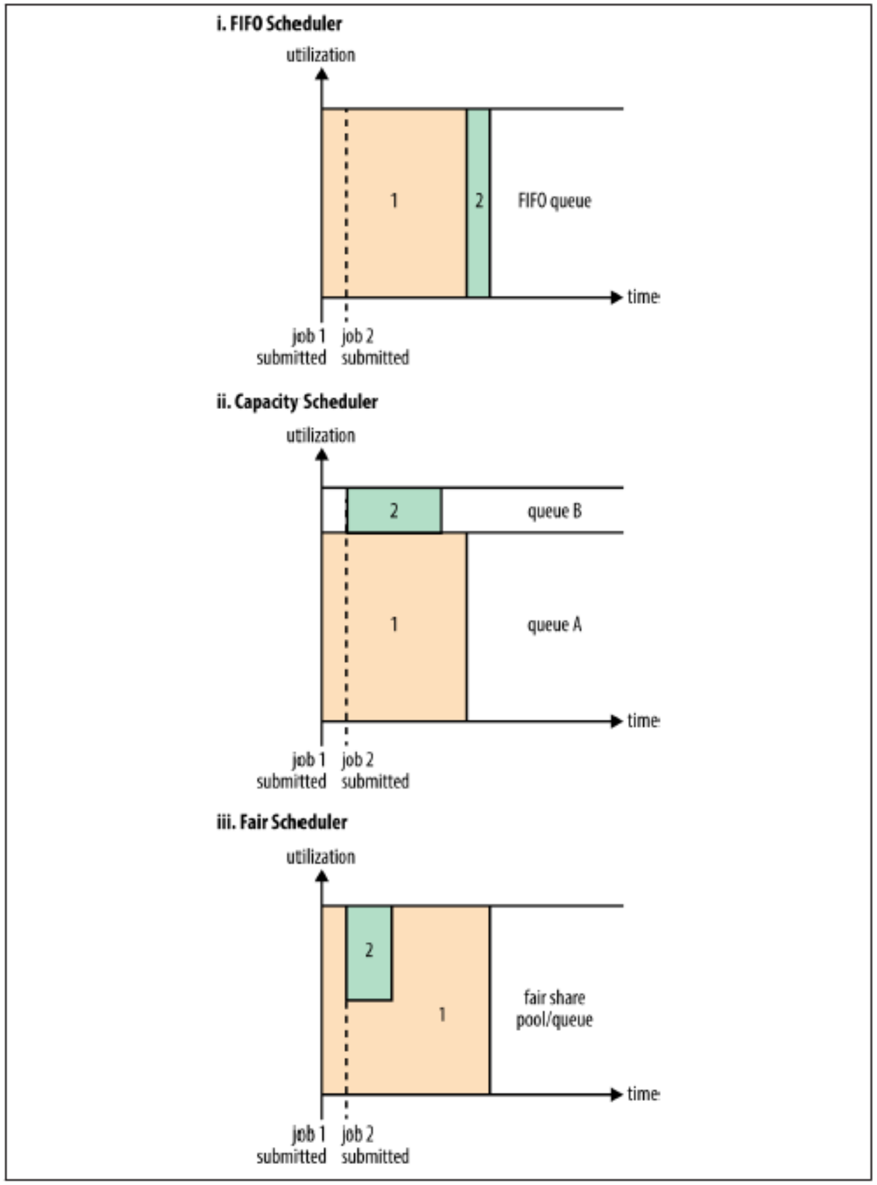


Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

Hadoop: the definitive guide, 4th edition, page 87

Capacity Scheduler

- Capacity Scheduler is designed for multi-tenancy situation
  - Each organization has a dedicated queue with a given fraction of the cluster capacity
    - Queues may be further divided to set up capacity allocation within organization
    - A single job does not use more resources than the queue capacity, but if there are more than one job in the queue, and there are idle resources, spare resources can be allocated to jobs in the queue
    - How much more resources can be allocated to a given queue is configurable
    - Which users can submit jobs to which queue is configurable

http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

Fair Scheduler

- The principle of fair scheduling is to allocate resources so that all running applications get similar share of resources
  - E.g. if there are R units of resources and J jobs, each job should get around R /J units of resources.
- YARN fair scheduler works between queues: each queue gets a fair share of the cluster resources
  - Within queue, the scheduling policy is configurable: FIFO or Fair

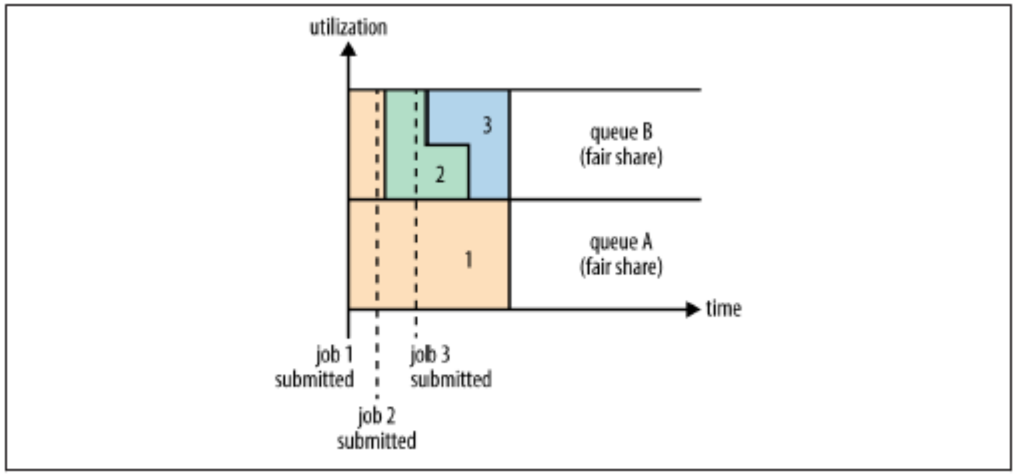


Figure 4-4. Fair sharing between user queues

Hadoop: the definitive guide, 4th edition, page 91

Fair Scheduler – Queue Placement

- Fair scheduler uses a rule based system to determine which queue an application should be placed in
  - Example queue placement policy

```
<queuePlacementPolicy>
        <rule name="specified" />
       <rule name="user" />
</queuePlacementPolicy>
```

- All queues in capacity scheduler needs to be specified beforehand by cluster administrator; Queues in fair scheduler can be created on the fly

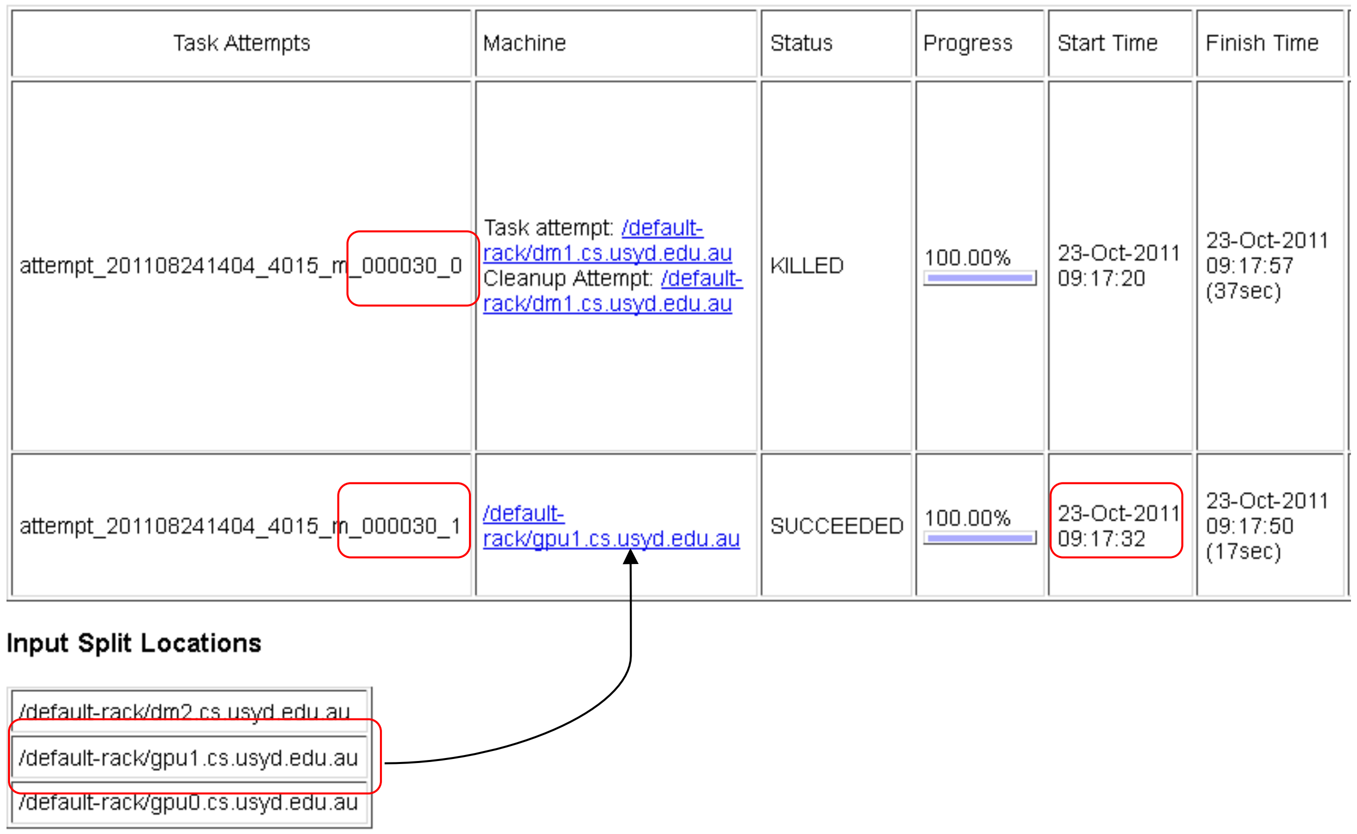
Data Locality Constrains

- Data intensive workloads (MapReduce/Spark)
  - have storage attached to computers.
  - Scheduling tasks near data improves performance.
- The requirements of fairness and locality often conflict
  - A strategy that achieves optimal data locality will typically delay a job until its ideal resources are available
  - Fairness benefits from allocating the best available resources to a job as soon as possible after they are requested
- General third party resource management system, e.g. YARN, might not follow the data locality preference

Hadoop's straggler handling mechanism

- Speculative execution
  - If a node is available but is performing poorly, this is called a straggler
  - MapReduce has a build-in mechanism to run a speculative copy of its task on another machine to finish the computation faster.
  - Speculative task attempts could be successful or just a waste
- Who manages this speculative mechanism
  - YARN resource manager
  - YARN node manager
  - Application Master?

A successful speculative task attempt

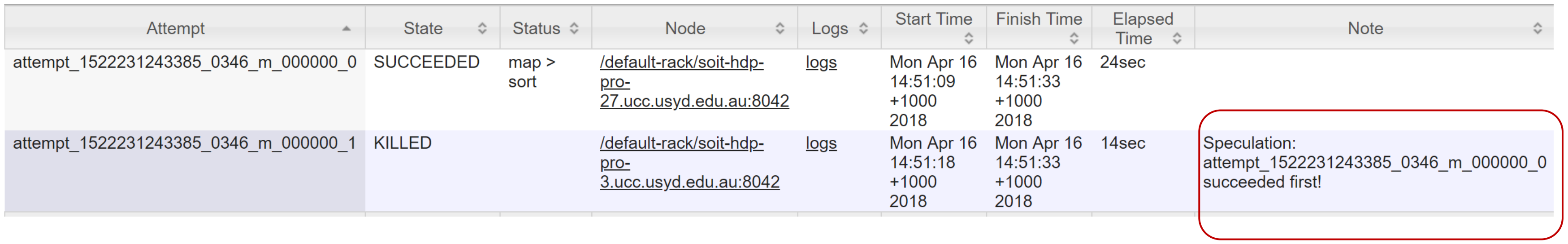


Task Attempts	Machine	Status	Progress	Start Time	Finish Time
attempt_201108241404_4015_m_000030_0	Task attempt: /default-rack/dm1.cs.usyd.edu.au Cleanup Attempt: /default-rack/dm1.cs.usyd.edu.au	KILLED	100.00%	23-Oct-2011 09:17:20	23-Oct-2011 09:17:57 (37sec)
attempt_201108241404_4015_m_000030_1	/default-rack/gpu1.cs.usyd.edu.au	SUCCEEDED	100.00%	23-Oct-2011 09:17:32	23-Oct-2011 09:17:50 (17sec)

Input Split Locations

/default-rack/dm2.cs.usyd.edu.au
/default-rack/gpu1.cs.usyd.edu.au
/default-rack/gpu0.cs.usyd.edu.au

A not useful speculative task attempt

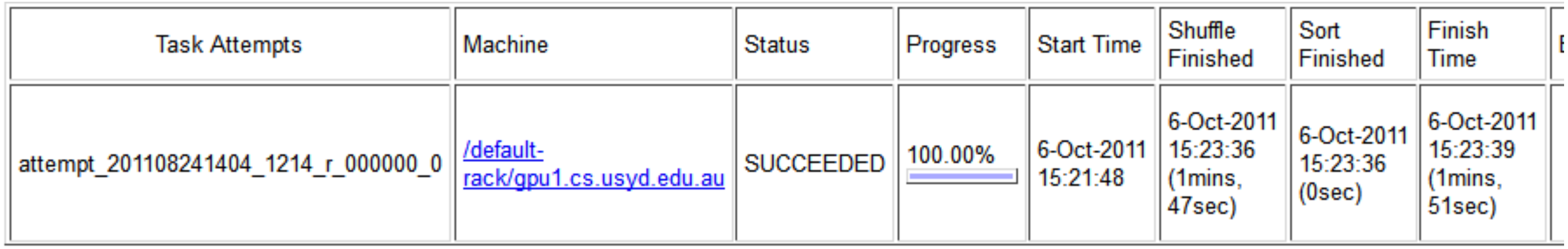


Attempt	State	Status	Node	Logs	Start Time	Finish Time	Elapsed Time	Note
attempt_1522231243385_0346_m_000000_0	SUCCEEDED	map > sort	/default-rack/soit-hdp-pro-27.ucc.usyd.edu.au:8042	logs	Mon Apr 16 14:51:09 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	24sec	
attempt_1522231243385_0346_m_000000_1	KILLED		/default-rack/soit-hdp-pro-3.ucc.usyd.edu.au:8042	logs	Mon Apr 16 14:51:18 +1000 2018	Mon Apr 16 14:51:33 +1000 2018	14sec	Speculation: attempt_1522231243385_0346_m_000000_0 succeeded first!

The first attempt succeeded while the speculative one is killed

Progress score

- Hadoop monitors task progress using a *progress score* to select speculative tasks
  - Map task's progress score is the fraction of input data read
  - Reduce task's execution is divided into three phases, each of which account for 1/3 of the score. In each phases, the score is the fraction of data process
- When a task's progress score is less than the average for its category minus 0.2 and the task has run for at least one minute, it is marked as a straggler



Task Attempts	Machine	Status	Progress	Start Time	Shuffle Finished	Sort Finished	Finish Time
attempt_201108241404_1214_r_000000_0	/default-rack/gpu1.cs.usyd.edu.au	SUCCEEDED	100.00%	6-Oct-2011 15:21:48	6-Oct-2011 15:23:36 (1mins, 47sec)	6-Oct-2011 15:23:36 (0sec)	6-Oct-2011 15:23:39 (1mins, 51sec)

For a ***reduce*** task, the execution is divided into three phases, each of which accounts for 1/3 of the score progress score is the fraction of input data read

Copy phase sort phase reduce phase

References

- Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung, *The Google File System*. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03),2003
- Tom White, Hadoop The Definitive Guide, 4th edition, O'Reilly, 2015
  - Chapter 4, YARN
  - Chapter 9 , MapReduce Features
- Apache Hadoop Yarn Introduction
  - http://hortonworks.com/blog/introducing-apache-hadoop-yarn/
  - http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/
- Determine YARN and MapReduce Memory configuration Settings
  - http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html
- How to plan and configure YARN and MapReduce 2 in HDP
  - http://hortonworks.com/blog/how-to-plan-and-configure-yarn-in-hdp-2-0/

References

- ***Improving MapReduce Performance in Heterogeneous Environment*.** Matei, Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, Ion Stoica. OSDI'2008
- Job Scheduling
  - FairScheduler: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
  - CapacityScheduler: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html