YARN

Resource Management

Now, we have a bunch of MapReduce tasks and HBase tasks running at the same time from different clients.

- Each application has its own master dedicated for its own applications.

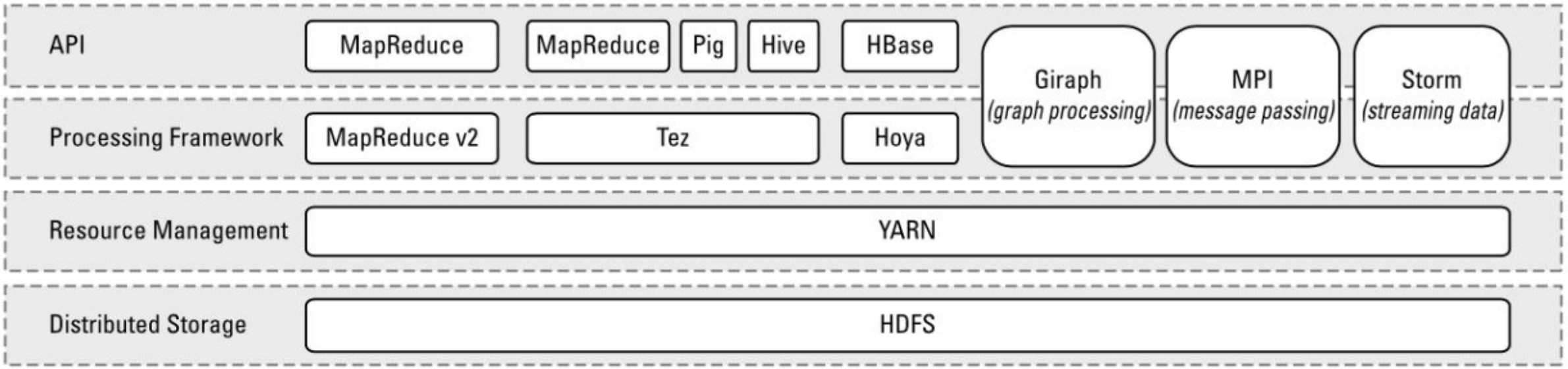
How can we efficiently manage all the resources we have in Hadoop?

YARN

YARN stands for Yet Another Resource Negotiator, a tool that enables other data processing frameworks to run on Hadoop.

YARN is meant to provide a more efficient and flexible workload scheduling as well as a resource management facility.

Both of which will ultimately enable Hadoop to run more than just MapReduce jobs.



YARN

Distributed storage: HDFS is still the storage layer for Hadoop.

Resource management: Decoupling resource management from data processing.

- This enables YARN to provide resources to any processing framework written for Hadoop, including MapReduce and HBase.

Processing framework: Because YARN is a general-purpose resource management facility, it can allocate cluster resources to any data processing framework written for Hadoop.

- It acts as the master for all applications in Hadoop

Working of YARN in Big Data

- YARN provides core services via two types of long-running daemons:
  - **Resource Manager** (one per cluster):
    - Manages the use of resources across the cluster.
  - **Node Managers** (running on all nodes):
    - Launch and monitor containers on each node.
- **Containers**:
  - Execute an application-specific process.
  - Constrained by resources like memory, CPU, and more.
  - May be a Unix process or a Linux cgroup (depending on YARN configuration).

Resource Manager

- Resource Manager is the core component of YARN, which governs all the data processing resources in the Hadoop cluster.
- Its only tasks are to maintain a global view of all resources in the cluster.
  - Handling resource requests, scheduling the request, and then assigning resources to the requesting application.
- The Resource Manager, a critical component in a Hadoop cluster, should run on a dedicated master node.

Resource Manager

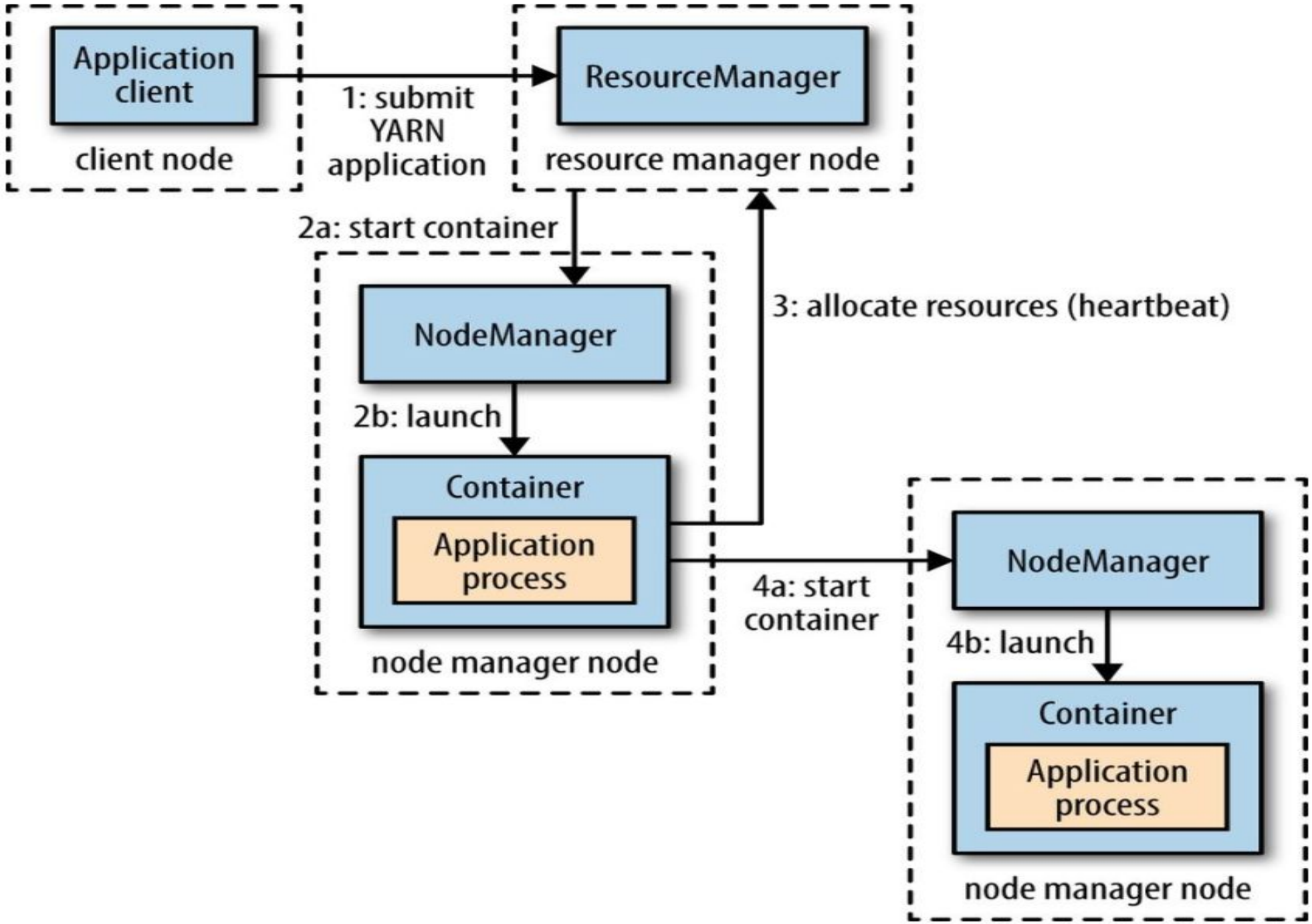
- The Resource Manager is completely agnostic with regard to both applications and frameworks.
  - “You know nothing, resource manager” – Hadoop.
- It has no concept of map or reduce tasks, it doesn’t track the progress of jobs or their individual tasks, and it doesn’t handle failovers.
- Resource Manager only does one thing: schedule workloads, and it does that job well.
  - Concentrating on one aspect while ignoring everything else.
- This high degree of separating duties is exactly what makes YARN much more scalable, able to provide a generic platform for applications.

Node Manager

- Doing scheduling only in a master node is not enough.
- Each slave node has a Node Manager daemon, which acts as a slave for the Resource Manager.
- Each Node Manager tracks the available data processing resources on its slave node and sends regular reports to the Resource Manager.
  - Colonels (Node Managers) in the battlefield (slave nodes) report to the supreme commander (Resource Manager) in the headquarter (master node).

Container

- The processing resources in a Hadoop cluster are consumed in bite-size pieces called containers.
- A container is a collection of all the resources necessary to run an application: CPU cores, memory, network bandwidth, and disk space.
- A deployed container runs as an individual process on a slave node in a Hadoop cluster.
- All container processes running on a slave node are initially provisioned, monitored, and tracked by that slave node’s Node Manager daemon.


Application client
client node
1: submit YARN application
ResourceManager
resource manager node
2a: start container
3: allocate resources (heartbeat)
NodeManager
2b: launch
Container
Application process
node manager node
4a: start container
NodeManager
4b: launch
Container
Application process
node manager node

Application Master

- Each application running on the Hadoop cluster has its own, dedicated Application Master instance.
  - E.g. Master in MapReduce and HMaster in HBase.
- The Application Master actually runs in a container process on a slave node.
- The Application Master sends heartbeat messages to the Resource Manager with its status and the state of the application's resource needs.
- Based on the results of the Resource Manager's scheduling, it assigns *container resource leases* —basically reservations for the resources containers need — to the Application Master on specific slave nodes.
- Each application framework that's written for Hadoop must have its own Application Master implementation.

Running an Application on YARN

- Client contacts the **Resource Manager** to run an **Application Master**.
- Resource Manager finds a **Node Manager** to launch the Application Master in a container.
- **Application Master**:
  - Can perform a computation within the container and return the result to the client.
  - Alternatively, can request more containers to perform distributed computation (e.g., MapReduce).

Communication in YARN

- YARN does not provide a built-in way for different parts of an application (client, master, process) to communicate.
- Most YARN applications use remote communication (e.g., Hadoop RPC) to:
  - Pass status updates
  - Return results to the client.

Resource Requests in YARN

- **Flexible Resource Request Model**:
  - Applications specify required resources (memory, CPU) for containers.
  - Requests can include **locality constraints**.
- **Locality Constraints**:
  - Critical for efficient data processing and bandwidth use.
  - Request containers based on:
    - **Specific node**
    - **Specific rack**
    - **Any node in the cluster** (off-rack)
- If constraints can’t be met:
  - No allocation, or
  - Loosen constraints (e.g., same rack, any node).

Locality Example and Handling

- **Example: HDFS Block Processing**:
  - Request a container on one of the nodes hosting a block's replicas.
  - If not possible:
    - Request on a node in the **same rack** as the replicas.
    - If still not possible, request on **any node** in the cluster.
- **Handling Failed Constraints**:
  - If specific node or rack isn't available:
    - YARN will allocate a container on a less preferred node or rack.
    - Ensures no unnecessary delays.

Dynamic Resource Requests

- **YARN Allows Resource Requests During Execution**:
  - Requests can be made upfront or dynamically as needs change.
- **Two Approaches**:
  - **Static (Spark)**:
    - Fixed number of executors started at the beginning.
  - **Dynamic (MapReduce)**:
    - Map task containers requested upfront.
    - Reduce task containers requested later.
    - If tasks fail, additional containers requested for retries.

Application Lifespan in YAR

- **Lifespan of YARN Applications**:
  - Can range from **short-lived** (seconds) to **long-running** (days or months).
  - Lifespan varies based on the type of application and job requirements.
- **Categorizing Applications**:
  - Instead of focusing on the duration, applications are categorized by how they map to **user jobs**.
- **Model 1: One Application per User Job**:
  - Each user job runs as a **separate application**.
  - Example: **MapReduce** follows this model.
  - Simple but less efficient due to lack of resource reuse between jobs.

Models for Application Lifespan

- **Model 2: One Application per Workflow/User Session**:
  - **More efficient** than Model 1.
  - Containers can be reused between jobs.
  - Intermediate data can be **cached** between jobs, saving processing time.
  - Example: **Spark** uses this model.
- **Advantages**:
  - **Improved efficiency**: reusing containers avoids repeated setup.
  - **Better performance**: caching intermediate data reduces time for multi-step processes.
- **Model 3: Long-Running Shared Applications**:
  - The application runs **continuously**, shared by multiple users.
  - Often used for applications requiring **coordination** of resources.
  - Example: **Apache Slider** for launching applications and **Impala** for SQL queries.

Benefits of Long-Running Applications

- **Low Latency for User Queries**:
  - "Always on" **Application Master** reduces the time needed to launch new applications.
  - Users benefit from **low-latency responses**, as resources are already managed.
- **Efficient Resource Utilization**:
  - Shared applications minimize the overhead of constantly starting new application masters.
  - Useful for **interactive or real-time** queries.
- **Example: Impala**:
  - Uses a **proxy application** that coordinates cluster resource requests.
  - Ensures fast query execution with minimal overhead.
- **Conclusion**:
  - YARN supports a variety of application models to meet different job types and performance needs.
  - Long-running and session-based applications can dramatically improve **resource efficiency** and **response times**.

MapReduce 1 (Hadoop 1) vs YARN (Hadoop 2)

- **MapReduce 1 (Hadoop 1)**:
  - Two types of daemons:
    - **JobTracker**:
      - Coordinates job execution, schedules tasks on **TaskTrackers**.
      - Monitors task progress and reschedules failed tasks.
    - **TaskTrackers**:
      - Execute tasks and report progress to JobTracker.
      - Handle task failures by reassigning tasks.
  - **JobTracker's Dual Role**:
    - **Job scheduling**: Assigns tasks to TaskTrackers.
    - **Task monitoring**: Tracks progress, restarts failed tasks, maintains counters.

YARN (Hadoop 2) Enhancements

- **YARN (MapReduce 2)**:
  - **Responsibilities are split**:
    - **Resource Manager**: Manages cluster-wide resources.
    - **Application Master** (one per job): Manages job lifecycle, task scheduling, failure handling.
  - **MapReduce 1 Limitations**:
    - **Single JobTracker** handles both resource allocation and task management, leading to potential bottlenecks.
- **Improvements in YARN**:
  - **Scalability**: Resource Manager and Application Masters divide the workload, improving cluster performance.
  - **History and Monitoring**:
    - **MapReduce 1**: JobTracker stores job history, with the option for a **Job History Server**.
    - **YARN**: Introduces a **Timeline Server** for application history.
  - **Efficiency**: More robust failure handling and better resource utilization.

Benefits of YARN Over MapReduce 1

- **Scalability**:
  - **MapReduce 1**:
    - Scalability bottlenecks around **4,000 nodes** and **40,000 tasks**.
    - **JobTracker** handles both job scheduling and task monitoring.
  - **YARN**:
    - Scales up to **10,000 nodes** and **100,000 tasks**.
    - **Resource Manager** and **Application Master** architecture separates resource management and job coordination.
    - Each application (e.g., MapReduce job) has a dedicated **Application Master**.
    - Aligns with the **Google MapReduce model**, where each job has its own master process.
- **Availability**:
  - **MapReduce 1**:
    - High availability (HA) achieved through replication of **JobTracker state**.
    - Complex to implement HA due to rapidly changing job state.
  - **YARN**:
    - **Resource Manager** and **Application Masters** support HA.
    - Divide-and-conquer approach for HA: manage Resource Manager and Application Master separately.

Utilization and Multitenancy

- **Utilization**:
  - **MapReduce 1**:
    - Uses static allocation of fixed-size **slots** (map and reduce slots).
    - Can encounter inefficiencies if slots are mismatched with task requirements.
  - **YARN**:
    - Manages a **pool of resources** rather than fixed slots.
    - **Fine-grained resource allocation** allows applications to request exactly what they need.
    - Avoids situations where tasks are delayed due to slot mismatches.
- **Multitenancy**:
  - **YARN**:
    - Supports various types of distributed applications beyond MapReduce.
    - Allows running different versions of MapReduce on the same YARN cluster.
    - Facilitates easier upgrades and integration of different applications.