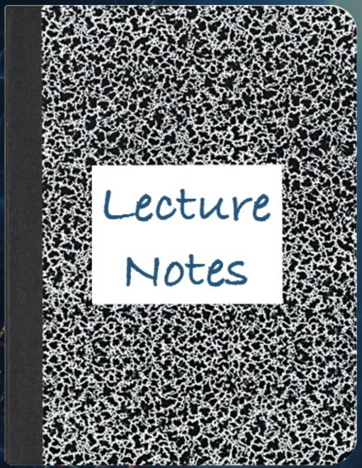
CS 417 – DISTRIBUTED SYSTEMS



Week 13: Infrastructure

Part 2: High Availability (HA) Clusters

Paul Krzyzanowski

Computer System Design Options

- **Highly Available Systems**
  - Incorporate elements of fault-tolerant design
  - Component replication, high-quality components
  - A fully fault tolerant system will offer non-stop availability … *but you can't have this!*
  - Problem: ↑ in availability ⇒ ↑ $$
- **High Performance Systems**
  - SMP architecture
  - Shared memory, shared clock, multiple processors
  - Problems:
    - Performance gain as f(# processors) is sublinear
    - Contention for resources (bus, memory, devices)
    - The solution is also expensive!
- **Commodity off-the-shelf Systems (COTS)**
  - Inexpensive
  - Problem: Not reliable and not high performance

Clustering

Achieve reliability and scalability by interconnecting multiple independent systems

Cluster:
A group of standard, autonomous servers configured so they appear on the network as a single machine

Single system image

Ideally…

- Bunch of off-the shelf machines
- Interconnected on a high-speed LAN
- Appear as one system to users
- Processes are load-balanced across the cluster
  - May migrate
  - May run on different systems
  - All IPC mechanisms and file access available
- Fault tolerant
  - Components may fail
  - Machines may be taken down

But…

We don't get all this in off-the-shelf platforms

- Systems design has engineering trade-offs
- Do you need fault-tolerant hardware?
  - Not if your software can work around it
    - Checkpointing, restarting processes, replicated servers, …
- Do you need high performance?
  - How frequently do processes need to communicate with each other?
    - Scientific computation (e.g., huge matrices) is different from MapReduce or Spark Streaming

Clustering types

- Datacenter job scheduling
- High availability (HA): failover cluster
- Supercomputing (HPC): includes batch processing
- Load balancing: simple workload distribution
- Storage clusters: shared storage

Datacenter Job Scheduling: YARN & Mesos

What was the problem?

- Lots of frameworks, each with its own task management

Hadoop MapReduce	Pregel	Apache Hama	Cassandra	Spark

What was the problem?

- Lots of frameworks, each with its own task management

Hadoop MapReduce
Pregel
Apache Hama
Cassandra
Spark
Microservices
Web sites
Apps

- Goal: manage resource usage among multiple frameworks
  - Long running vs. terminating jobs; interactive vs. batch jobs
  - Production vs. test vs. development

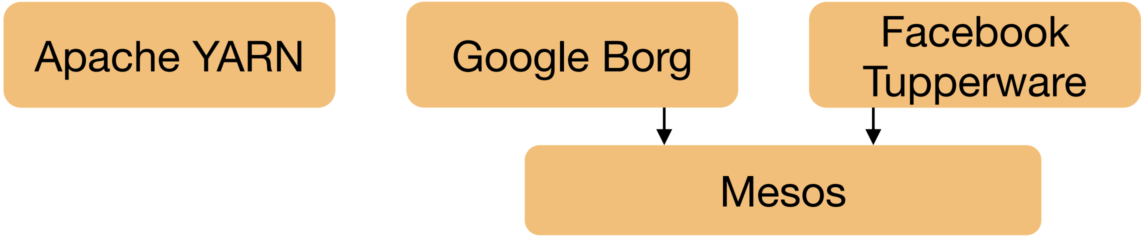
What was the problem?

- Lots of frameworks, each with its own task management

Hadoop MapReduce | Pregel | Apache Hama | Cassandra | Spark

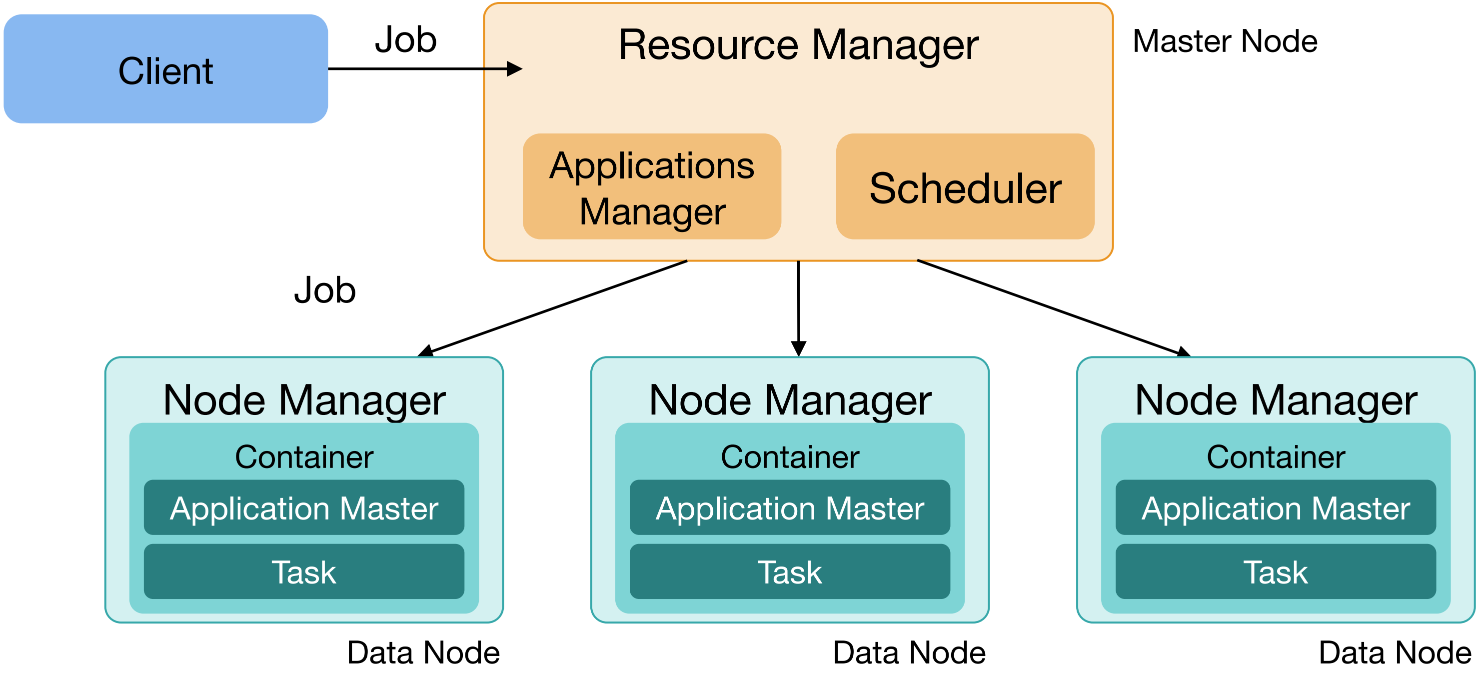
Microservices | Web sites | Apps

- Goal: manage resource usage among multiple frameworks
  – Long running vs. terminating jobs; interactive vs. batch jobs
  – Production vs. test vs. development



Apache Hadoop YARN (Yet Another Resource Negotiator)

Take resource management out of Hadoop MapReduce and create a generic resource manager/scheduler



Apache YARN Components & Functions

One Master Node, multiple Data Nodes

Resource Manager: runs on Master Node

- Tracks resources for the cluster
- Clients submit jobs (e.g., MapReduce or any program) to the Resource Manager
- Contains an *Applications Manager* and *Scheduler*
- **Applications Manager:** Accepts or rejects jobs from clients
  - Validates the request: enough resources available, it's not a submission with a duplicate ID
  - Maintains a queue of submitted jobs & completed jobs (so users can request results)
- **Scheduler:** decides if the job can be run
  - If yes, the Applications Manager allocates a container (reserves CPU & memory) for the job
- Application Manager finds a node that can handle the job and contacts the **Node Manager** at that node

Node Manager: runs on Data Nodes

- Allocates the resources to execute the job
  - Creates a **container**: an environment where a YARN application runs – a pool of memory
  - Optionally, can be an isolated environment with limits via Linux *cgroups*
  - Can also be configured to use Docker containers for isolation
- Launches **Application Master** within the container
  - Framework-specific (e.g., dedicated Application Master for MapReduce)
  - Responsible for executing and monitoring all the tasks for the job
  - Checks if additional resources are needed (e.g., multiple *map* and *reduce* workers)
    - If so, contact the Resource Manager with a *Resource Request* with resource needs and location constraints (e.g., proximity to data)
    - If this results in tasks on multiple nodes, they will each send a heartbeat to the Application Manager
- Each Application Master sends the status of the job execution to the Resource Manager

Apache Mesos

- Developed at UC Berkeley – 2007
- Distributed resource management and job scheduling system
- Used by Airbnb, Apple, Netflix, X, Uber, Yelp, …

Mesos Goals

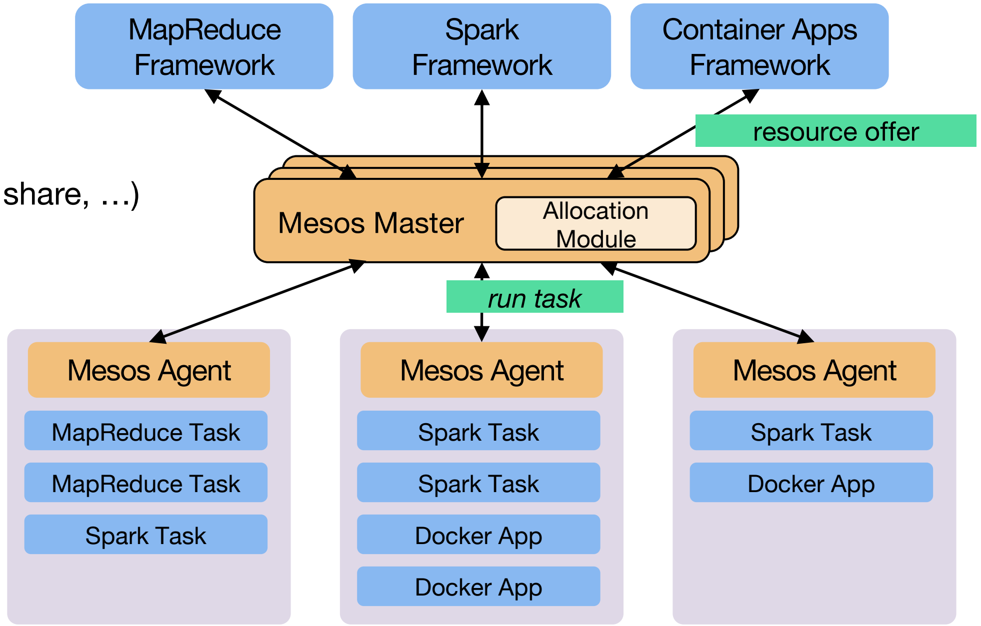
- High utilization of computing resources
- Run multiple frameworks – including future ones
- Run multiple instances of the same framework
- Provide isolation between frameworks
- Scale to tens of thousands of nodes in a data center
- Operate reliably

Microkernel-like approach

- Track available computing resources
- Allow frameworks to run tasks on specific nodes
- **Resource sharing**
  - Frameworks divide work into tasks
  - Mesos allocates tasks
- **Resource offers**
  - Mesos tells frameworks about available computing resources {machine, memory, CPUs, …}
  - A framework can accept or reject a resource offer

Mesos Design

- Mesos Master
  - Tracks available resources on each node
- Allocation module
  - Schedules jobs among frameworks (priority, fair share, …)
  - Controls which framework to offer resources
- Master creates *resource offer*
  - List of nodes & resources:
    node: {#CPUs, #GPUs, memory, …}
- Offer sent to framework
  - Framework can do its own scheduling (e.g., consider locality of data)
  - Framework accepts or rejects offer
- If accepted, framework tells Mesos to launch tasks on a specific node
- Mesos agent launches and task in an isolated *executor*



High Availability (HA) Clustering

Cluster Components

- Cluster membership
- Heartbeat & heartbeat network
- Quorum
- Configuration & service management
- Storage

Cluster Membership

Software to manage cluster membership

- What are the nodes in the cluster?
- Which nodes in the cluster are currently *alive* (active)?

We saw this:

- Group Membership Service in virtual synchrony
- GFS master, HDFS NameNode
- Bigtable master
- Pregel master
- MapReduce Master & Spark Cluster Manager

Quorum

Some members may be dead or disconnected

Quorum: number of elements that must be online for the cluster to function

- Voting algorithm to determine whether the set of nodes has quorum (a majority of nodes to keep running)
- We saw this with Raft consensus (& Paxos): forcing a majority avoids **split-brain**

Quorum disk

- Shared storage: whichever node can reserve the disk owns it
- Enables systems to resolve who runs a service in small clusters even if the network becomes partitioned

Types of Quorum

- **Node Majority**
  - Each available node can vote
  - Need majority (over 50%) of votes for the cluster to continue running
  - Best for odd number of nodes, larger clusters
- **Node & Disk Majority** (Microsoft *Disk Witness*)
  - Designated shared disk = *disk witness*: counts as a vote
  - Need majority of votes to continue running
  - Best for an even # of nodes in one site
- **Node & File Share Majority** (Microsoft *File Share Witness*)
  - Shared file system = *file share witness* : counts as a vote
  - Need majority of votes to continue running
  - Windows Server 2019: File Share Witness on USB stick
    - Shared USB storage on router
  - Best for an even # of nodes in a multi-site cluster
- **No majority**
  - Cluster has quorum if even one node is available and can communicate with a specific disk in the cluster

Cluster configuration & service management

- **Cluster configuration system & manager**
  - UI to manage the configuration of systems and software in a cluster
  - Administrator has a single point of control
- **Cluster management agent**
  - Runs in each cluster node: changes propagate to all nodes
  - Tracks cluster membership – removes failed nodes
  - Keeps track of quorum – stops cluster when ≤ nodes not active
- **Service management & Scheduler**
  - Identify which applications run on which systems
  - Specify how failover occurs
    - Active: system runs a service
    - Standby: Which system(s) can run the service if the active dies
  - E.g., MapReduce, Pregel, Spark all use coordinators for their service
  - General purpose schedulers: *Apache Mesos*, *Hadoop YARN, Google Borg*, *Linux Slurm*

Disks

Shared storage access

- If an application can run on any machine, how does it access file data?
- If an application fails over from one machine to another, how does it access its file data?
- Can applications on different machines share files?

Network (Distributed) File Systems

One option:

- Network file systems: NFS, SMB, AFS, etc.
- Works great for many applications

Concerns

Availability

- Address with replication (most file systems offer little)

Performance

- Remote systems on a LAN vs. local bus access
- Overhead of remote operating system & network stack
- Point of congestion
- Look at GFS/HDFS to distribute file data across lots of servers
  … or other parallel file systems, such as Lustre, GlusterFS, or Ceph

Shared disks & Cluster file systems

Shared disk

- Allows multiple systems to share access to disk drives
- Works well if there isn't much contention

  … but you can't have multiple systems
  reading/writing/caching the same disk blocks

Cluster File System

- Client runs a file system accessing a shared disk at the **block level**
  - *vs. a distributed file system, which access at a file-system level*
- No client/server roles, no disconnected modes
- All nodes are peers and access a shared disk(s)
- **Distributed Lock Manager (DLM)**
  - Process to ensure mutual exclusion for disk access
  - Provides inode-based locking and caching control
  - Not needed for local file systems on a shared disk

Cluster File Systems

Examples:

- IBM General Parallel File System (GPFS)
- Microsoft Cluster Shared Volumes (CSV)
- Oracle Cluster File System (OCFS)
- Red Hat Global File System (GFS2)

Linux GFS2 (no relation to Google GFS)

- Cluster file system accessing storage at a **block level**
- Cluster Logical Volume Manager (CLVM):
  Volume management of cluster storage
- Global Network Block Device (GNBD):
  Block level storage access over ethernet: cheap way to access block-level storage

The alternative: shared nothing

Shared nothing

- No shared devices
- Each system has its own storage resources
- No need to deal with DLMs
- If a machine A needs resources on B, A sends a message to B
  - If B fails, storage requests have to be switched over to a live node

Requires exclusive access to shared storage

Rely on active replication of changes or …

- Multiple nodes may have access to shared storage
- Only one node is granted exclusive access at a time – *one owner*
- Exclusive access changed on failover

SAN: Computer-Disk interconnect

SAN = Storage Area Network

- Separate network between nodes and storage arrays
  - Fibre channel
  - iSCSI
- Any node can be configured to access any storage through a fibre channel switch

Acronyms

- **DAS**: Direct Attached Storage
- **SAN**: block-level access to a disk via a network
- **NAS**: file-level access to a remote file system (NFS, SMB, …)

Failover

HA issues

- How do you detect failover?
- How long does it take to detect?
- How does a dead application move/restart?
- Where does it move to?

Heartbeat network

- Machines need to detect faulty systems
  - **Heartbeat**: Periodic “ping” mechanism
  - An “are you alive” message

- Need to distinguish ***system faults*** from ***network faults***
  - Useful to maintain redundant networks
  - Avoid split-brain issues in systems without quorum
    (e.g., a 2-node cluster)

- Once you know who is dead or alive, then determine a course of action

Failover Configuration Models

- Active/Passive
  - Requests go to active system
  - Passive nodes do nothing until they're needed
  - Passive nodes maintain replicated state (e.g., SMR/Virtual Synchrony)
  - Example: Chubby
- Active/Active
  - Any node can handle a request
  - Failed workload goes to remaining nodes
  - Replication must be N-way for N active nodes
  - Example: GFS chunks
- Active/Passive: $N+M$
  - M dedicated failover node(s) for N active nodes

Design options for failover

- **Cold failover**
  - Application restart
  - *Example: map and reduce workers in MapReduce*
- **Warm failover**
  - Restart last checkpointed image
  - Relies on application checkpointing itself periodically
  - *Example: Pregel*
- **Hot failover**
  - Application state is synchronized across systems
    - E.g., replicated state machines or lockstep synchronization at the CPU level
  - Spare is ready to run immediately
  - May be difficult at a fine granularity, prone to software faults (e.g., what if a specific set of inputs caused the software to die?)
  - *Example: Chubby*

Design options for failover

With either type of failover …

Multi-directional failover

- Failed applications migrate to or restart on available systems

And possibly

Cascading failover

- If the backup system fails, application can be restarted on another surviving system

IP Address Takeover (IPAT)

Depending on the deployment:

- **Ignore**
  - IP addresses of services don't matter. A load balancer, name server, or coordinator will identify the correct machine
- **Take over IP address**
  - A node in an active/passive configuration may need to take over the IP address of a failed node
- **Take over MAC address**
  - MAC address takeover may be needed if we cannot guarantee that other nodes will flush their ARP cache
- **Listen on multiple addresses**
  - A node in an active/active configuration may need to listen on multiple IP addresses

Hardware support for High Availability

- Hot-pluggable components
  - Minimize downtime for component swapping
  - E.g., disks, power supplies, CPU/memory boards
- Redundant devices
  - Redundant power supplies
  - Parity on memory
  - Mirroring on disks (or RAID for HA)
  - Switchover of failed components
- Diagnostics
  - On-line identification & service

Fencing

- **Fencing**: method of isolating a node from a cluster
  - Apply to failed node
  - Disconnect I/O to ensure data integrity
  - Avoid problems with Byzantine failures
  - Avoids problems with *fail-restart*
    - Restarted node has not kept up to date with state changes
- **Types of fencing**
  - **Power fencing**: shut power off a node
  - **SAN fencing**: disable a Fibre Channel port to a node
  - **System service fencing**: disable access to a global network block device (GNBD) server
  - **Software fencing**: remove server processes from the group
    - E.g., virtual synchrony

Cluster software hierarchy

Example: Windows Server cluster abstractions

Top tier: Cluster abstractions

- Failover manager (what needs to be started/restarted?)
- Resource monitor (what's going on?)
- Cluster registry (who belongs in the cluster?)

Middle tier: Distributed operations

- Global status update
- Membership
- Quorum (and leader election)

Bottom tier: OS and drivers

- Cluster disk driver, cluster network drivers
- IP address takeover

The End