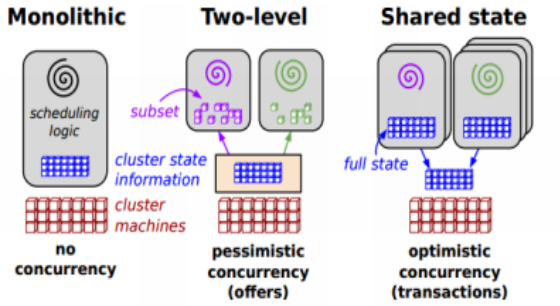
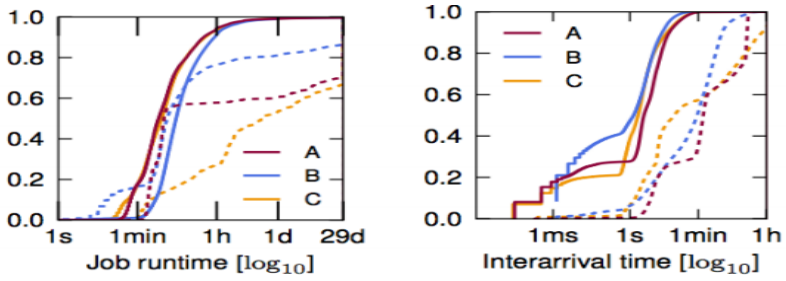
CLUSTER PRINCIPLES

Large scale clusters are expensive also from an energetic point of view, so it important to use them well: improving the efficiency and reducing the utilization both imply a better scheduling which reduces the dimension of the cluster. The scheduling problem indeed become complicate when an heterogeneous mix of application run concurrently, we could have problems like bottlenecks in which cluster and workload sizes grow, and obviously schedulers must be scalable because his complexity is proportional to the cluster size.



These are the different **architecture designes**. I can do a sort of division statically (ex. One machine for storage, one for cloud lab ecc), but the multiplexing is a better idea. The monolitic architecture use a centralized scheduling and resource management for all jobs, so all jobs are send to the same machine; priority are necessary and also policies, but once defined introducing new policies isn't a great idea because they're difficult to add. In the two level, one decides how many resources dedicated to an individual job like a single resource manager: with resources we mean cpu, ram, ecc. So one component take decision, others instead implement the things. This architecture is based of the pessimistic concurrency, the problem in this case is that the resource management and locking (locking = concurrency control) are conservative, which can hurt cluster utilization and performance. Instead, shared state architecture is based on transactions with an optimistic concurrency. Hints about pessimistic and optimistic concurrency from google:
Optimistic concurrency control works on the assumption that resource conflicts between multiple users are unlikely (but not impossible), and allows transactions to execute without locking any resources. Only when attempting to change data are resources checked to determine if any conflicts have occurred. If a conflict occurs, the application must undo and handle the conflict.
Pessimistic concurrency control locks resources as they are required, for the duration of a transaction. Unless deadlocks occur, a transaction is assured of successful completion. We avoid conflicts.

Schedulers should support heterogeneous (lots of different applications) workloads; firstly, we can do a categorization of job types: there are **batch jobs**, fort example MapReduce computations, or **service jobs** like end-user facing web service, which could require more constraints (like MBI apps, which require that alle the machine must be availaible before the application starts.



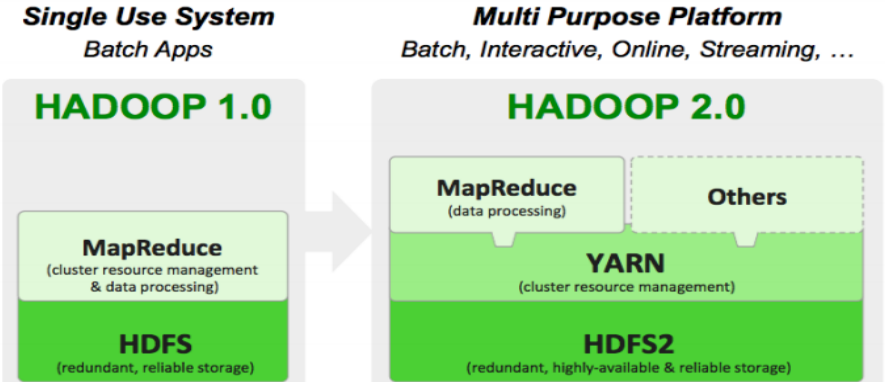
In the picture there is an example of real cluster, in which the solid lines are the batch jobs, the dashed lines the service jobs: the first picture is a CDE: cumulative distribution. A, B, C are three clusters, the data come from google. On the left, we have the distribution of the jobs, at right we've interarrival time, the difference between the time when a job finish and the time in which the next job begin. We noticed that large fraction is small (less than one hour), lots of them require just one task. ; furthermore lots need more or less 2

minutes, so they free resources quickly. Median value: 30s, it means that every 30s a new job is launched.
How to exploit these characteristic while designing a scheduler? What are we supposed to keep in mind? There are multiple frameworks. We can decide to **partition the resource** on the cluster statically or dinamically for example (workload partitioning and specialize cluster), or a load balance workload-oblivious based, or a mix of them. We've to consider the **policies about resources**: which are **available**? All or an subset? We should decide how to handle the **interferences** (pessimistic vs optimistic scheduling). Other things to consider when designing: **cluster-side behaviour** (strict fairness? Priority based? With Pre-emption? Pre-emption = when I don't wait to finish a job that I've already launched if another one has a better priority) and the **allocation granularity** (job task granularity, if I use global policies or per-partition policies for example).

Little resume:
Goal of scheduling: minimize the response time of the system, composed by the queueing time and the service time, working on priorities, per-job constraints, failure tolerance and scalability. The architectures could be: monolithic, statically partioned, two level and shared level. The monolithic design implements the same algorithm to all the jobs, the statically partitioned scheduler is a sort of standard approach but it work well only under assumptions that they must be respected or there will be an elevate fragmentation of resources or a sub-optimal cluster utilization. The two level schedulers obviates these problems, using a dynamical allocation of resources with a logically centralized grant who decide the resources to assign to each job: an example is Mesos, in which the available resources are offered to competing frameworks with exclusive offers in order to avoid interference; so it is a sort of pessimistic concurrency! Another example is YARN, in which the centralized resource allocator is called RM and there is a per-job framework master which is the AM, which provides job management services, not proper scheduling: actually YARN is really close to a monolithic architecture. So let's analyze one by one all the main cluster schedulers.

YARN

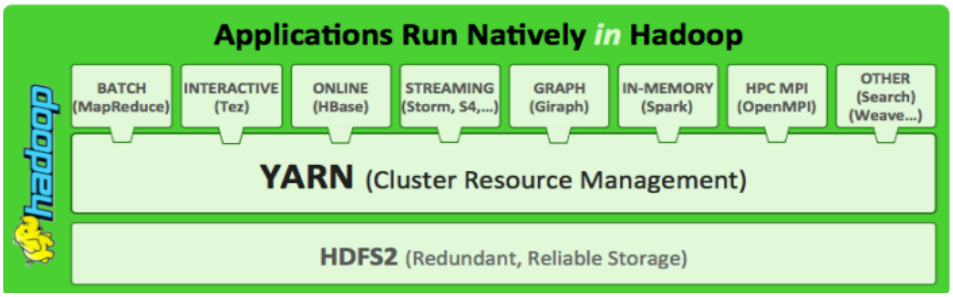
Why YARN?



Let's analyse the first implementation of hadoop, hadoop 1.0, in which neither YARN or another cluster scheduler is not present: this version of hadoop has a serie of limitations. Hadoop 1.0 was built just for batch applications and no other, and it supposes that the clients submits the jobs to an unique job tracker which manages the cluster resources and it performs both job and task scheduling. There is one task tracker per-machine who handle the job execution. So, the limitations are that this framework supports only mapreduce tasks and the other tasks like interactive or online tasks should be casted to batch job, so iterative applications are slow. The maximum cluster dimension is about of 4000 nodes, and the number of tasks that can be executed in parallel is about 40000; another flaw are system failure: they could involve in the destruction of both running and queued jobs. Finally, the resources utilization is not optimal: it is a static partition of the resources in Map or Reduce slots, these slots are fixed.

So, let's see the YARN ecosystem: the main characteristics are that all the data are stored in the same place in order to avoid the cost of the duplication; the interaction with data could happen in several ways and not only in bath mode with the rigid mapreduce model, so the performance will be more predicatable

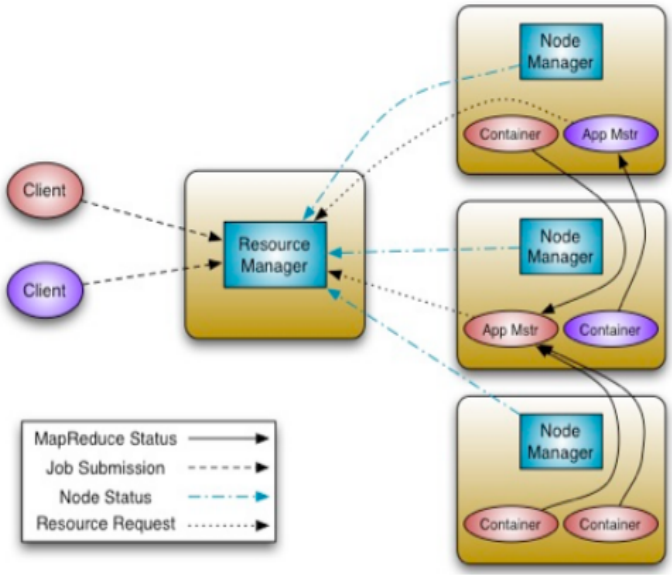
exploiting more modern scheduling mechanisms.



The key improvements with the introduction of YARN are:

- Support for multiple applications: indeed, the assignment of the resources is separated from the application logic and some libraries and protocols exist and they permit the development of custom application: so the same hadoop cluster could be shared by lots of different applications.
- Improved cluster utilization: now there aren't the fixed MR (= mapreduce) slots, they are replaced by generic containers, and their allocation is based on locality and memory, always because more applications share the cluster.
- Improved scalability: removing the application logic from the management resource improve also the scalability: the scheduling protocol is more compact. (? State machine, message passing based loosely coupled design).
- Application Agility: it uses the protocol buffers for RPC (= Remote Procedure Call, the activation of a routine in a machine different by the machine that invoked it). MR bevomes an application in user space. This allows the co-existence of multiple version of an application, and in general the upgrading of framework and apps is easier.
- A data operating system: shared services: in this way, the common services are included in pluggable frameworks, like distributed file sharing services, remote data read services and the log aggregation service.

YARN architecture overview

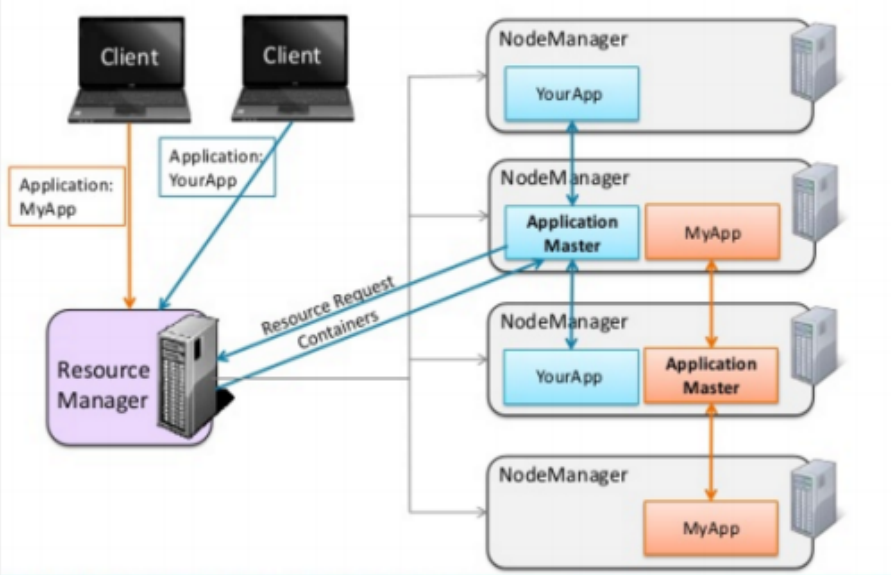


There is no more static resource partition, and no more slots: here nodes have resources which are allocated to applications when requested. The idea is to separate the resource management from

application logic, so the role of the job tracker in hadoop 1.0 now is do by the resource manager and several application masters; furthermore, they support any YARN applications and not only map-reduce apps.
So, the YARN daemons are:

- **Resource Manager:** it runs on the master node, it is a global resource manager and scheduler, which has to arbitrate the concurrent applications.
- **Node Manager:** they run on the slave nodes, the communicate with the Resource Manager and they are useful for reports. It could have only containers, not necessarely one AM.
- **Resource container:** they're created by the Resource Manager upon request, and they allocate a certain amount of resources on slave nodes: applications run in one or more containers.
- **Application Master:** there is one Application Master specific per-application, so every new application requires a new Application Manager that has to be designed and implemented.

Following a complete example with applications: the AM request the resources to RM and it obtains the reference of containers.

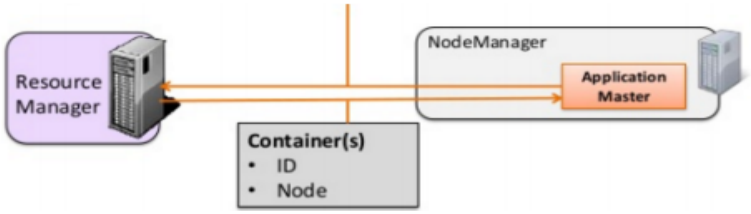


YARN Core Components
Until now we saw the architecture overview, now let's see the YARN Core components. First of all, schedulers (YARN schedulers) are a pluggable component of the Resource Manager, and the supported way of scheduling are: the capacity scheduler (which guarantees a minimal capacity to each task), the fair scheduler (all the tasks receive, on average, an equal share of resources over time) and the scheduler based on dominant resource fairness (a variant of fair scheduler, which is only based on memory: the dominant resource fairness schedulers assign resources both memory and CPU based).
The first abstraction of capacity scheduler is a queue (each queue has a fraction of the total resources), but the fair schedulers have the concept of hierarchical queues, in order to divide the resources assigning a weight to each queue which the purpose to share the cluster in specific proportions. So, the role of sub-queue is sharing resources assigned to queue: for example, a company could assign a weight to engineering taks equal to 2, and 1 to marketing, and have some sub-queues for each products in the first case or for website or data analysis in the second case.

So, the operations performed by a RM are: node management, container management (handling AM request for new containers, de-allocating containers when they expires or when the application end), AM management (create (at least?) a container for each new AM and hadles it) and there are also some security integration (Kerberos protocol).
Instead, the operations performed by a NM are: managing of the communication with the RM periodically (heartbeats) with some node's information including the status of the container, managing of the processes in containers which means launching AMs on request from the RM, launching application processes on

request from the AMs, monitoring the usage of resources and killing processing and containers; finally, it should provides logging services like roll over HDFS.

The resource requests (from AM to RM) are composed by the resource name, a priority internal to application (not across apps), the amount of resources requested in terms of both memory and CPU and the number of containers. The YARN Containers have got a Container ID, some commands to start the application task(s), an enviroment configuration and some local resources like binary or HDFS files.



YARN Fault Tolerance: the failure points are several:

- Container failure (they are handled by node manager. When a container fails or dies, node-manager detects the failure event and launches a new container to replace the failing container and restart the task execution in the new container, after launching an exception or a failure. If an application has too many failures, it is considered failed.
- Then we've AM failure: if it happens, the RM will re-attempt the whole application. There is an option called Job Recovery: if it is false, all containers are re-scheduled, if it is true it uses a saved state in order to find which containers succeeded and which failed, and it re-schedule only failed ones.
- NM failure: if the node master stops sending heartbeats to RM, RM removes it from active node list, and containers on the failed node are re-scheduled wile AM on the failed node are re-submitted completely.
- RM failure: it is the worst one becase the application can't be run if it is down. It can work in active-passive mode like the NN of HDFS.

Finally, the shuffle mechanisms still exists but it is just an auxiliary system, it runs on the JVM as persistent system.
Example WordCount: the RM launch the client application in a node, the AM of that node performs a resource request for the map tasks, and if availaible the desidered number of containers are allocated; when these tasks terminate, the RM de-allocates them and it receive a new request, this time for the reduce tasks: the process is equal to the previous one and at the end the NM of the application communicate to the RM the ending of the job.

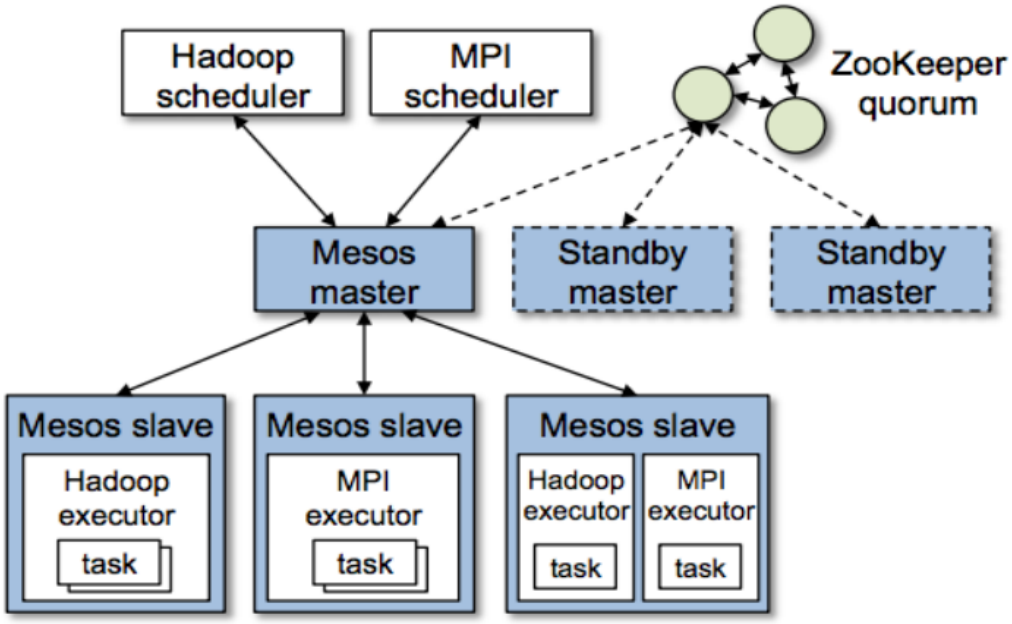
By notes:
Yarn is a 2 level scheduler but is is similar to monolithc one; the resources are not statically allocated but there are dinamically shared between heterogeneous apps, the binding with apps and protocols written with different languages is possible. AM decides how many containers the app needs and communicates it to RM.
Domande:
Why do you need an external shuffle service?
If I lost data, whats happen?
Why am I not interested on where the reducers are? (No data locality, once executing the reducers data have to been shuffled on each free reducer).

MESOS

Mesos is built using the same principles as the Linux kernel, only at a different level of abstraction. The Mesos kernel runs on every machine and provides applications (applications to "program the cluster" e.g., Hadoop, Spark, Kafka, Elastic Search) with API's for resource management and scheduling across entire

datacenter and cloud environments. Every application needs the right framework, so it is possible having several frameworks in parallel: the multiplexing cluster resources among frameworks improve the cluster utilization and allows sharing of data without the need to replicate it. The main solutions to achieve the cluster sharing are the static partitioning ad the traditional virtualization; the problems of the existing approaches are the mismatching between allocation granularities and the absence of mechanisms to allocate resources to short-lived tasks, so the motivation of MESOS are the following: to achieve the operation of cluster frameworks with short-lived tasks and to free up quickly the cluster resources: this allows a data locality. So, the definition of MESOS is: *a thin resource sharing layer enabling fine-grained sharing across diverse frameworks.* His purpose is, as definition says, assist each supported framework which has different scheduling needs, it must be scalable (10000+ nodes) and with an high availability and fault tolerance.,
So it appears natural that a centralized approach cannot work, due to a scalability and complexity problem: indeed, the input are the framework requirements (which remind that they are framework-specific), the instantaneous resource availability and the organization policies, and the output is a global schedule for all tasks of all jobs of all frameworks, so at the limitations we've to add the voice "cost", because moving framework-specific scheduling to a centralized scheduler requires an expensive refactoring. Therefore MESOS is based on a decentralized approache, based on the abstraction of resource offer, so the system decides how many resources to offer to a framework, then the framework decides which resources to accept and which tasks to run on them. The typical enviroment sees a workload in which the 90% of the jobs ends in less than 1000 second, and they could be MR jobs, ad-hoc queries SQL like, large scale machine learning ecc.

MESOS Architecture: the design is datacenter-like, with a scalable and resilient core exposing low level interfaces and high-level libraries for common functionalities; there is a minimal interface to support resource sharing, but MESOS itself manages the cluster resources and the framework**s** control task scheduling and his execution: it is a two-level approach and we've the advantages of this architecture that we discussed at the beginning: indeed, frameworks are independent and they can support different scheduling requirements, MESOS is kept simple in order to minimaze the rate of change to the system.

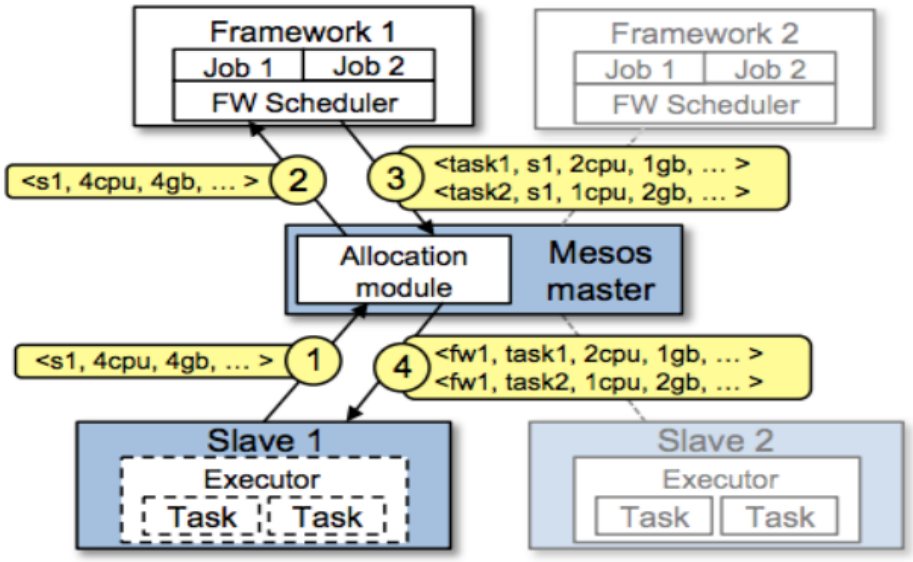


The most important guy are:

- **Mesos master:** it uses the mechanism of resource offers (a list of free resources on multiple slaves) to implement a fine-graned sharing, it collects the resource utilization from slaves. In the policy First-level scheduling the master decides how many resources to offer a framework, implementing a cluster-wide allocation approach priority based and fair sharing.
- **Mesos framework:** it is composed by the **framework scheduler** and the **framework executor**. The

first one talks with the master and it decides which offer to accept and it describes the tasks to launch on accepted resources, the second one is launched on MESOS slaves and it uses the accepted resources. Here there's a second-level scheduling: indeed there is one framework scheduler per application which decides how to execute a job and its tasks.

Following, an example of resource offer: each framework make a request for resources to the master, the master receives the characteristics of each slave. It sees the request and it makes an offer to the framework. This one decides which resources accept and it communicate to the master (after a second-level scheduling) which tasks of which jobs he is going to run with those resources. After that, the framework executor, on the slaves, runs the tasks. The master could seems a bottleneck, but it isn't so.



The architecture see so far imply that MESOS make **no assumption on framework requirements,** differently from other approaches in which the scheduler has to understand the application constraints (ex YARN); note that the user has to specify the same there constraints! So when the framework receives an offer, it could decide to reject it if it doesn't satisfy the application constraints: they can wait for an adeguate offer. Also for this reason MESOS is simple, because the logic and the control is delegated to individual frameworks, even if MESOS could implements filters in order to optimize resource offers.

The resource allocation module is pluggable and it could be based on a (max-min) fairness or on a strict priority, but it is bosed on the fondamental assuption that tasks are short and that MESOS only reallocates the resources when a task finishes: for example, assume that a framework's share is % of the cluster: it needs to wait the % of the mean task length to receive its share. Indeed, the strictly fairness approach is not effective, because if an application requires the 20% an another the 60% and the cluster is completely free, the offer for both is 50% of the resources.
Resource revocation: in general jobs are short, but some jobs like a streaming may have long tasks: in this case, MESOS could kill running tasks. Therefore some **preemption primitives** exist and they require knowledge about potential resource usage by a framework: this because in some cases killing an application could be just wasteful but not critical (like MR applications), while some applications (like message-passing interface application MPI) could be harmed so it could be better not killing them. The guaranteed allocation is a minimum set of resources granted to a framework: if the resources allocated to a framework are below the guaranteed allocation, it is not necessary that MESOS kill its tasks, if above some killing are possible (if it hasn't got preemption!).
Performance isolation: Through low-level OS primitives (part of pluggable modules) it is possible to achieve an isolation between executors on the same slave machine: the supported mechanisms are about the CPU limitations, memory, network and I/O bandwith of a process tree, it is an isolation better that the current approach process based, but a fine-grained isolation isn't yet fully functional.
MESOS Scalability: to achieve scalability, MESOS has some **filter mechanisms** in order to avoid unnecessary communications in the rejection process (ex if I offer you 10 and you reject, next time I don't ask you if you

want 8). There are two types of filter: the first one restrict which slave machine to use, the second one check resource availability on slaves. Another way is improving the **speed-up of the resource offer mechanism** because MESOS keep trace of the offers to a framework toward its allocation. Finally, a framework could wait the resources but MESOS can decide to **invalidate an offer** to avoid blocking and misbehaviors: all these things improve the scalability.
Mesos fault tolerance: the tolerance to fault is helped by the design of the master, which has a **Soft-State** that contains, at every time, a list of active slaves, a list of registered frameworks and a list of running tasks: this should imply a faster recovery if one of these 3 things fail. The master failure is contrasted with the introduction of **multiple masters:** if so, the first thing to do is electing a leader through **Zookeeper**: the leader is active, others in stand-by: if a failure happens a new master is elected and slaves and executors gives him the information necessary to popolate the new master's (soft) state. MESOS helps the frameworks to tolerate failures too, because it sends *health reports* to frameworks schedulers periodically and allows multiple schedulers for a single framework. The important thing is that MESOS proposes always resources in order to reach the 50% max of the total cluster resources or until the framework stops it, so if a failure happens a new master is elected and there is however 50% of resources available in these cases, for example for rescheduling.

Yarn: Container = linux process
Mesos: Container = linux container

System behaviour: the ideal workloads for MESOS is composed by elastic frameworks, supporting scaling up, with an homogeneus and short task durations without strict preferences over cluster nodes. If happens that nodes prefer different nodes, MESOS can emulate a centralized scheduler in this case, instead if the tasks have got an heterogeneous duration MESOS is able to handle short and long lived tasks and the performance degradation is accetable. Let's see some defenition that we could meet:

- **Workload characterization -> Elasticity:** elastic workloads can use resources as soon as they are acquired, and release them as soon as tasks finish; in contrast, rigid frameworks (like MPI) can only start a job when **all** resources have been acquired, and they don't work well with scaling.
- **Workload characterization -> Task runtime distribution:** we can have an homogeneus distribution or not, it could be uniform or exponential.
- **Resource characterization -> Mandatory:** it is a resource that a framework **must** acquire to work. The natural assumption is that mandatory resources < guaranteed share.
- **Resource characterization -> Preferred:** resources that a framework should acquire to achieve better performance but are not necessary to achieve the job purpose.
- **Performance metrics:** we can meet the framework ramp-up time which is the time that a new framework take to achieve its fair share, the job completion time which is the time it takes a job to complete (since it started), assuming one job per framework (so without second-level scheduling), and the system utilization which is the total cluster resource utilization under the CPU and memory point of view.

Some examples:
Let's do some considerations on Homogeneus and Heterogeneous tasks. Suppose homogeneus tasks with a mean task duration equal to T, a cluster with n slots and a framework f with k slots: the job duration is B*k*T, so if f has k slots the job duration is B*T. With elastic frameworks, the utilization is 1, ramp-up time is T or T ln k (with exp. Distribution) and the completion time is (1/2+B)*T.
Suppose that placement preferences exists, two cases could happen: if there exists a configuration that stisfy all frameworks constraints the optimal allocation is achieved in at most T interval. Otherwise, it means that the demand is larger than supply: a weighted fair allocation could be achieved with the **lottery scheduling**: suppose that si is the intended allocation to framework i, m is the total number of framework registered to MESOS, the probability to offer a slot to framework i is si/sum (i=1...m) si, so frameworks which are supposed to receive more resources have more probability to obtain them firstly.

Having heterogeneous tasks means that the tasks could be either long or short, and the worst scenario is

that all nodes required by a short job are filled with long tasks, which means a big wait time. If X is the fractions of long tasks (< 1) and a cluster has S slots per node, the probability for a node to be filled with only long tasks is X^S. If S=8 and X=0,5, it is equal to 0,4%.

Finally, let's talk about the **limitations of distributed scheduling:** one of these is the fragmentation which implies an under utilization of system resources, besides a distributed collection of frameworks could not achieve the same quality of a centralized schedulers but this is mitigated by having clusters of big nodes (many CPUs, cores) running small tasks. Another problem is the starvation, in which large jobs wait indefinitely for slots to become free and small tasks jobs monopolize the cluster: a solution is a *minimum offer size mechanism:* so it becomes possible to give an enough number of resources also to large job.
Performance of MESOS: both CPU and memory utilization are higher in MEMOS than a static partitioning (an average of 80-60 for the CPU and 30-10 for memory), the data locality can be hight in mesos that a static, the job running time is higher in static than in mesos.

Domande:
Which is the most important parameter among T, K, B ?
Difference between executing an elastic and a rigid framework ?
Why tasks with an exponential time distribution are more difficult to be handled?

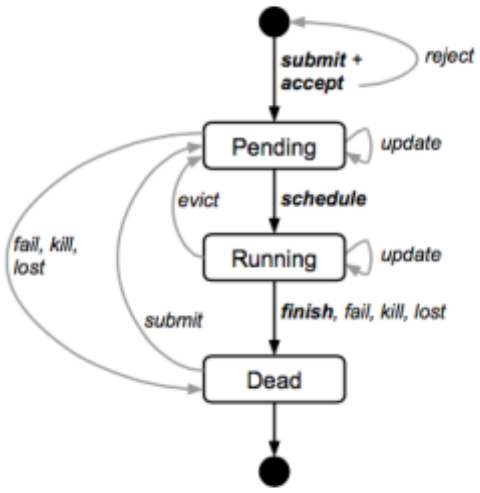
BORG

Google Borg is another cluster manager, it want to hide the details of the cluster management in order to allow the user on focusing on application development, it is characterized by an high reliability and availability because it tolerates failures within a datacenter and across datacenters which can involve a degradation of the service; like MESOS, it can run heterogeneous workloads and scale across thousand of machines.

The user perspective: the applications developed by users are called jobs which are composed by one or more tasks, all tasks run the same binary, each job runs in a set of machines managed as a unit called **borg cell.**

Workload: there are 2 main categories supported: the **long running services** which are jobs thatshould never go down and characterized by short-lived and latency sensitive requests; they could include the **storage services** like the previous but used to store data. Then we've tje **batch jobs**, delay tolerant jobs that should take from few seconds to few days to complete. Note that the workload composition in a cell is dynamic, and it depends from the tenants using the cell and from time: for example, some end-user-facing jobs are diurnal, the batch jobs are better distributed.
Cluster and cells: a **borg cluster** is a set of machines connected by an high-performance datacenter-scale network: note that all the machines in a borg cell all belong to a single cluster which lives inside a datacenter which composes a site. The **borg machines** are physical servers dedicated to execute Borg applications, heterogeneous in terms of resources and with a public IP address. Median cell size: 10k machines, used to schedule application tasks but also install their binaries and dependencies, monitor their helth and restart them in case of failure.
Jobs and Tasks: a job is defined by a name, the owner and the number of tasks, but also the constaints must be specified: these force tasks to be run on machines with particular attributes, and they could be hard or soft (preferences). Each task maps to a set of UNIX processes running in a container on a Borg machine in a Borg cell. A task is composed ve the index within his parent job and the resource requirements; they tipically have the same definition and they can run on any resources dimension because there aren't fixed-size slots or buckets.In order to specify jobs and tasks Borg use his own declarative configuration language which allows lambda functions to calculations. With RPC, user can interact with live jobs, updating the specification of tasks while their parent job is running; depending on the entity of the changing, there could be some **side effects:** indeed the tasks could be restarted (es if I push a new binary), or could require a

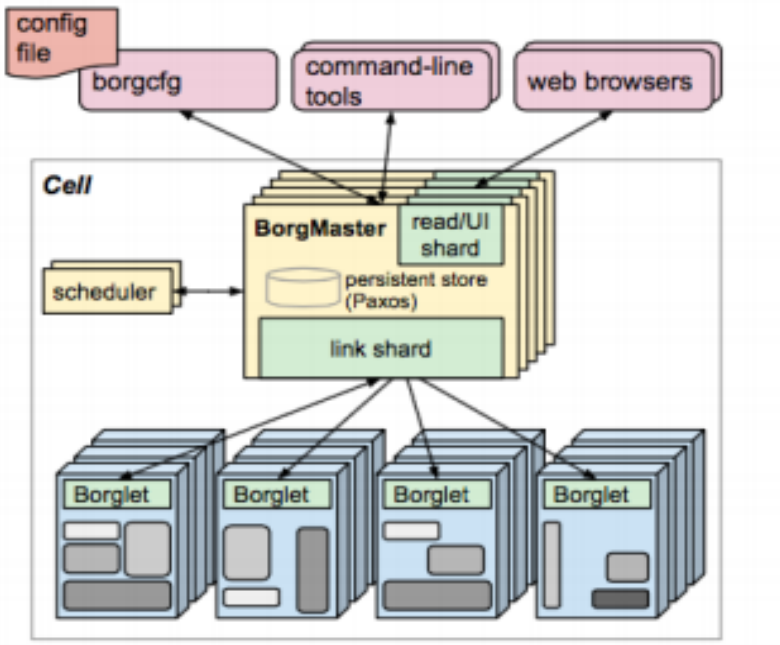
migration (for a change on specification) or nothing (like a change on priority).



In the picture we can see the **jobs state diagram,** self explained.
Resource allocations: a **Borg alloc** is a reserved set of resources on an individual machine, they can be used to execute one or more tasks that equally share resources; note that **resources remain assigned wheter or not they are used**. The typical use of the Borg alloc is set resources aside for future tasks or between stopping and starting them. An **alloc set** is a group of allocs on different machines: once an alloc has been created, one or more jobs can be submitted. More or less, task : alloc = job : set.

Priority, quotas, admission control: what's happen if more works shows up that they can be accomodated? The solution is the job priority and quota. Every job has a priority, a small positive integer. A highpriority task can obtain resources at the expense of a lowerpriority one, even if that involves preempting (killing) the latter. Borg defines non-overlapping priority bands for different uses, including monitoring, production, batch, and best effort. This essentially means that users must "manually" cluster their application according to such bands; the job/user quotas are used to decide which job to admit for scheduling, they're expressed as a vector of resource quantities.Note that this isn't scheduling but more likely an admission control.
Borg name service is a mechanism to assign a name to a taske, a task name is composed by the cell name, job name and a task number.
Monitoring and billing services: each task in Borg has a built in HTTP server that provides health information and performance metrics. Borg SIGMA monitores UI Services, it has the state of the jobs and of the cells: it is a debugging service and it helps users finding job specifications that can be easy scheduled. Billing services use monitoring information to help users to debug their jobs and other things. Thanks to this, it is easier answering to the question "why my task is pending?".

BORG ARCHITECTURE:



The architecture components are:

- A set of physical machines (borg machines)
- A logically centralized controller, the **Borgmaster**
- An agent process running on all machines, the **Borglet**

There is a Borgmaster is each cell and it orchestrates cell resources: it is composed by the Borgmaster process and the scheduler process. The first one handles client RPCs that mutate state or lookup for state, manages the state machine for all Borg object like machines, tasks, allocs ecc, communicate with all Borglets in the cell and provides a web-based UI.
Borgmaster reliability: it is achieved through replication, for example it is a single logical process but it is replicated 5 times in a cell. The election of the master happen using Paxos when a cell starts or upon a failure of the current master. The Borgmaster replicas mantain an in-memory fresh copy of the cell state, and persist their state to a distributed Paxos persistent store; they help building the most up-to date when a new master is elected. NB Replicas are not passive ! We'll see it.
The Borgmaster state: Borgmaster checkpoint its state with both a time based and an event based mechanism: checkpoints have many uses, including restoring a Borgmaster's state to an arbitrary point in the past (e.g., just before accepting a request that triggered a software defect in Borg so it can be debugged); fixing it by hand in extremis; building a persistent log of events for future queries; and offline simulations.

A high-fidelity Borgmaster simulator called Fauxmaster can be used to read checkpoint files, and contains a complete copy of the production Borgmaster code, with stubbed-out interfaces to the Borglets. It accepts RPCs to make state machine changes and perform operations, such as "schedule all pending tasks", and we use it to debug failures, by interacting with it as if it were a live Borgmaster, it connects with simulated Borglets replaying real interactions from the checkpoint file. A user can observe the changes to the system state that actually occurred in the past. Fauxmaster is also useful for capacity planning ("how many new jobs of this type would fit?"), as well as sanity checks before making a change to a cell's configuration ("will this change evict any important jobs?").

Scheduling: New submitted jobs (and their tasks) are stored in the **Paxos store** (for reliability) and put in the **pending queue:** then, the scheduler process **operate at the task level, not the job level,** it scans **asynchronously** the pending queue and assigns tasks to machines that satisfy constraints and that have enough resources. The scan proceeds from high to low priority, modulated by a round-robin scheme within a priority to ensure fairness across users and avoid head-of-line blocking behind a large job. The scheduling

algorithm has two parts: feasibility checking, to find machines on which the task could run, and scoring, which picks one of the feasible machines. In feasibility checking, the scheduler finds a set of machines that meet the task's constraints and also have enough "available" resources – which includes resources assigned to lower-priority tasks that can be evicted. In scoring, the scheduler determines the "goodness" of each feasible machine. The score takes into account user-specified preferences, but is mostly driven by built-in criteria such as minimizing the number and priority of preempted tasks, picking machines that already have a copy of the task's packages, spreading tasks across power and failure domains, and packing quality including putting a mix of high and low priority tasks onto a single machine to allow the high-priority ones to expand. There are 3 types of scoring mechanism:

- **Worst fit: spreading tasks,** there is a single cost value across heterogeneous resources, this approach wanna minimize the change in cost when placing a new task: this lead fragmentation.
- **Best fit: waterfilling algorithm:** it tries to fill machines as tightly as possible. This leaves some machines empty of user jobs (they still run storage servers) that can be used to place large tasks.
- **Hybrid:** it tries to reduce the amount of stranded resources – ones that cannot be used because another resource on the machine is fully allocated.

Resuming, the selection of pending happen with a round-robin priority mechanisms: when a new task arrives, the system doea a feasibility checking and it discovers wheter ther are enough free machines; it doesn't do the preemption against high priority tasks, and it prefers a machine in which there are already a copy of my process with the dependencies already installed. But, it is possible the preemption of low priority tasks which are suspended for one high priority task, this is negative.

Domanda: who decides the priorities?

Time startup latency: this metric measures the time from jubmission to a task running, and it could be really variable; the main part of this time is caused by binary and package installations, so the idea iin order to reduce latency is placing tasks on machines that already have dependencies installed.

The Borglet: it is a Borg agent present on every machine in a cell, it starts and stops tasks and restarts the failed ones. It manages the machine resources interacting with the OS, it maintains and rolls over debug logs; therefore it interacts also with the Borgmaster, reporting the state of the machine with a pull-based mechanisms, with periodical messages every few seconds: the master perform the flow control to avoid message storm. A Borglet continue operations even if the communication to Borgmaster is interrupted, but a failed Borglet is blacklisted and all tasks are rescheduled. But if many borglets communicate concurrently, it is possible a message overhead, and the solution is that many Borgmaster replicas receive state updates. This is called link shard mechanism: each borgmaster replica communicates with a subset of the cell Borglets, and the partitioning is computed at each leader election. A Borglet report a full state, but the link shard mechanism aggregate state information in order to reduce the load at the master.

Scalability: the typical requirements are 10k tasks per minute, 10+ cores and 50 GB of RAM for 1000 machine in a cell. The scalability is enforced by the decentralized design: indeed the scheduler process is separate from Borgmaster process, indeed there is one scheduler per Bergmaster replica so the scheduling is somehow decentralized: each changing is communicated from replicas to Borgmaster that finalizes the state update. Another improvement is the caching.

Borg Behaviour: experimental perspective

Availability: in large scale systems, failure are the norm, of both Borg and its running applications. The base techniques in order to achieve high availability are: replication, storing persistent state in a distributed file system, checkpointing, but also the automatic rescheduling of failed task, the rate limitation (against message storm), avoiding duplicate computation, admission control, minimizing external dependencies for task binaries.

System utilization: however, the primary goal of a cluster scheduler is to achieve high utilization, because the machines, network, power, cooling ecc cost a lot.

A sophisticated metric is called **Cell Compaction**, it replaces the typical average utilization metric and provides a fair, consistent way to compare scheduling policies, translating directly into cost/benefit result. The computation happen in this way: given a workload in a point in time, enter a loop of workload packing; at each iteration, remove physical machines from the cell, and exit the loop when the workload can no longer fit the cell size. Use Fauxmaster to produce experimental results.
Let's talk about the **cell sharing:** is it better to share or not? Many current system apply static partitioning, for example one cluster is dedicated only to production jobs, one cluster for non-prod jobs. The benefits from sharing are that Borg can reclaim resources reserved by "anxious" production jobs.
Cell sizing: is better having them large or small: in large cells large jobs could accomodated, and large cells also avoid fragmentation.
Fine-grained resource request: Borg users specify job **requirements** in terms of resources, in milli-core for CPU and bytes for RAM and disk. Fixed size containers/slots are not a good idea because it would require more machines in a cell. Borg users specify also resource **limits** for their jobs, used to perform admission control and feasibility checking; some jobs need to use all their resources, others never use all resources.
Resource reclamation: Borg estimates of resource usage (resource reservations); the Borgmaster receives usage updates from Borglets. The production jobs do not rely on reclaimed resources, Borgmaster only uses resource limits. So there is an issue of good estimation. Note that users tend to ask resources than they actually need, so there exist monitoring mechanisms that check in real time the usage of resources and if it is less than expected they free them.
Isolation: sharing is positive, but tasks may interfere one with each other, we must prevent it for security and performance. It is achieved thinking that all tasks run in Linux cgroup-based containers, Borglets operate on OS to control container resources; a control loop assigns resources based on predicted future usage. Other techniques: application classes (latency sensitive vs batch), resources classed, ecc.

Disadvantages: the jobs abstraction is too simplistic, multi-job services cannot be easily managed nor addressed. The addressing services is critical, one IP address implies managin ports as a resources, which complicated tasks. An high instruction is required to use Borg.
Advantages: Alloc are useful under the point of view of sharing resources, cluster management is more than task management, introspection is useful for debugging and monitoring, the master is the kernel of a distributed system, necessary because monolithic system are not affective. The open source version of Borg is called Kubernetes.