Paper Summary

YARN: Yet Another Resource Negotiator
Borg, Omega, and Kubernetes
Vandit Agarwal, Chun-Jung Chiu, Ting-Wei Cho

Problem:
The summary of moving from Apache Hadoop to the development of YARN as a generic resource management platform for varied application frameworks by the community can be narrated as a story. Initial implementation of Apache Hadoop focussed aggressively on simply running gigantic MapReduce jobs, possibly neglecting key design features in the process of doing so. This resulted in a monolithic architecture. The community of researchers and engineers pushed the MapReduce paradigm beyond the limits of its underlying resource management layer leading to several problems, all of which can be categorized into two main buckets:

- tight coupling between the resource management architecture and the application framework and
- centralized flow control handling of the jobs

The paper describes and attempts to answer the following issues:

1. Scalability: The existing infrastructure at Yahoo! (called Dreadnaught) had reached its limits as the web was expanding. They had to migrate to a new system which would keep pace with the rate of web's expansion.
2. Multi-tenancy: As the researchers and engineers realized the potential of the MapReduce model (which was initially designed with an intention to crawl and index web), more and more applications found confidence in using this framework. This demanded a multi-tenant support system.
3. Serviceability: Yahoo!'s HoD was flexible enough to keep pace with the aggressive release cycle of Hadoop. Researchers and engineers could run slightly older versions of Hadoop while testing newer version by deploying new cluster every time. This was of critical importance.
4. Locality Awareness: The spatial distribution of the compute and the storage is of key importance to counter latency issues specially when the system scales. HoD did not take care of this key aspect.
5. High Cluster Utilization: In HoD, cluster allocation latency was very high. This stemmed from people not having a good estimate of required resources and amplified by people sharing the clusters with their friends.
6. Reliability/Availability: Retiring from HoD to move to shared clusters exposed a plethora of reliability issues and downtimes.
7. Secure and Auditable Operations: As more and more tenants moved to Hadoop, security and Auditing became a major requirement to be preserved through every implementation.
8. Support for Diverse Programming Models: Users of MapReduce occasionally spawned spun off alternative frameworks. To the resource manager they just looked like isolated map jobs with unfamiliar resource utilizations that led to issues like starvation, deadlocks, under-utilization etc.
9. Flexible Resource Model: Typed slots in emerging frameworks proved to be bottlenecks that led to this requirement.
10. Backward Compatibility: The wide adoption/success rate of MapReduce in place did not allow a radical design change. This forced a need to be backward compatible.

Hypothesis:
The hypothesis can be derived directly from the requirements that warranted the need for YARN in the first place. The authors (and the community in general) saw the development of the trends with the broader adaptability of the Hadoop and MapReduce frameworks. They realised that the the current systems in place are not robust enough to handle the growing demands, both in terms of current functionality and more features. They anticipated and realized that with growing scale and multiple tenants all sharing the same physical resources (logically abstracted as containers or any other form of logically bundled resources), needs for secure, reliable operations which are auditable and always available will also arise. The model under design had to be be flexible, support a plethora of programming models being developed every day, be backward compatible with the already existing frameworks and yet be serviceable. For efficiency's sake, the model also had to take into account the locality awareness of physical resources that helps achieve high utilization of resources. We see that all these do hold true in today's infrastructure and having paid attention to these details during development stages itself have gone a long way in proving this system to be useful and efficient than many already existing solutions as suggested by the results achieved in various deployed use cases.

Solution:
1. Scalability: YARN makes the system more scalable compared to Hadoop 1.0 by splitting the role of JobTracker into smaller pieces: Resource Manager (RM), Application Master (AM), and Node Master (NM). Since it is more light-weighted, it can manage more nodes than Hadoop 1.0.
2. Multi-tenancy: Allocating resources in unit of containers implements multi-tenancy, which enables nodes to run multiple tasks simultaneously in isolation.
3. Serviceability: The fact that the AM can run arbitrary user code and can be written in any language leads to serviceability. Now that using different versions of Hadoop in a cluster is possible, we don't need to bring down the entire service all at once for software upgrade. Upgrade can be done on a small part of the cluster one at a time without the user's' knowledge.
4. Locality Awareness: The AM is allowed to issue locality (node-level, rack-level, global) requests to the RM so that computation can be done close to data's storage location.
5. High Cluster Utilization: In YARN, high cluster utilization is achieved by optimizing and limiting the resources requested by individual jobs. This is done through ResourceRequests tied to each job in AM, which limit the amount of resource of the job so that the resource in a cluster is not locked in applications which do not require that much resource.
6. Reliability/Availability: If the RM fails, it is able to restore to it previously stored state. Some frameworks run on top of YARN can recover user pipelines after the failure. This ensures that the applications are available even when the RM fails.
7. Secure and Auditable Operation: All submitted jobs need to go through an admission control phase, where their security credentials are validated and other operational and administrative checks are performed. Also, a NM does not configure containers for use until the lease issued to it is validated. These validation and checks ensures that all jobs are secure and under control.
8. Support for Programming Model Diversity: The use of AM makes decoupling frameworks and Hadoop possible. The AM is designed such that it can run any user code in any languages.

Furthermore, all communications between AM and its containers that are application-specific are not facilitated by YARN, and are managed by each framework.

9. Flexible Resource Model: In YARN, resources are statically partitioned (e.g. map slots and reduce slots in Hadoop 1.0) so that they are fungible rather than fixed. The use of containers leads to more flexible resource management e.g. the resources used by a map task can later be used by a reduce task.
10. Backward Compatibility: The code base of YARN is establish on the previous version to make sure that applications coupled with older versions can easily upgrade to newer ones.

Drawback and Improvement:

1. YARN only deals with allocating resources for computation jobs, but not deploying software on machines. On contrast, some other systems e.g. Mesos, can do both. YARN can adopt concept of container, dependency isolation for container deployment as in systems such as Borg and Kubernetes.
2. The paper doesn't mention what if the master (RM) node failed. We suggest to create a shadow RM to continuously heartbeat to the master node and be in synch with the master RM.
3. If ApplicationMaster dies, all running tasks will be lost because the platform does not store AMs' state. A possible solution to this would be to keep track of the status of each task so that if the AM fails, all running jobs can be restored from the log history. This can also be implemented by some sort of checkpointing and roll-backing mechanism.
4. Failures of containers are not handled by YARN as of now, and YARN requires the framework developers to handle those by themselves. Since container failure is a fairly likely scenario in a large cluster, it should be handled by YARN by default.

Comparison of Systems:

1. Scheduling:
   a. Omega belongs to distributed and multi-level scheduling, which can focus on scalability. Omega is an internal system in Google, so it grant scheduler full access to entire cluster and there is no master node's bottleneck.
   b. Kubernetes has a monolithic scheduler. All the scheduler's jobs is in a master node. It is able to implement a more complex scheduling algorithm running in one node.
   c. YARN is a hybrid scheduler architecture between a distributed one and a monolithic one. The scheduler job is separated in a master node and slave nodes.
2. Locality:
   a. YARN's ApplicationMaster can select a task with input data close to the container.
   b. Kubernete has an option to specify the locality preference when creating a pod.
3. Resopnsibility:
   a. Hadoop YARN is focused on running different big data framework (MapReduce, REEF, etc.)
   b. Borg, Omega, Kubernetes is an orchestration system for container management. It even optimize the network for container-use only.
   c. Mesos is aimed to support a wide range of different frameworks.Therefore, it can be responsible for both container deployment and assigning compute jobs of big data.