

Cloud Computing

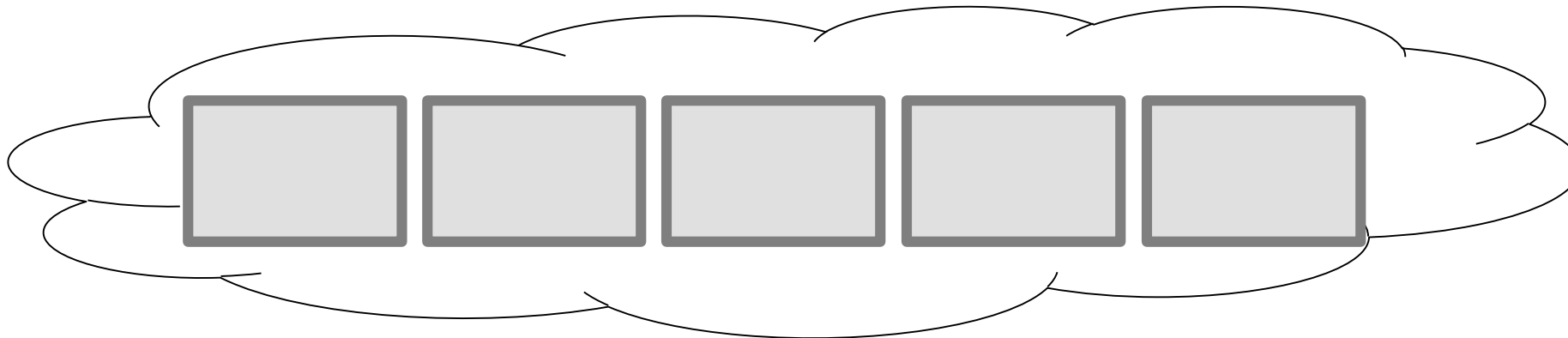
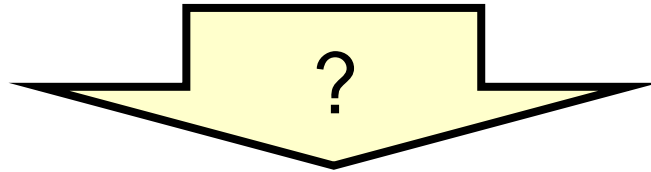
Kapitel 6: Cluster Scheduling

Dr. Josef Adersberger

Das Problem

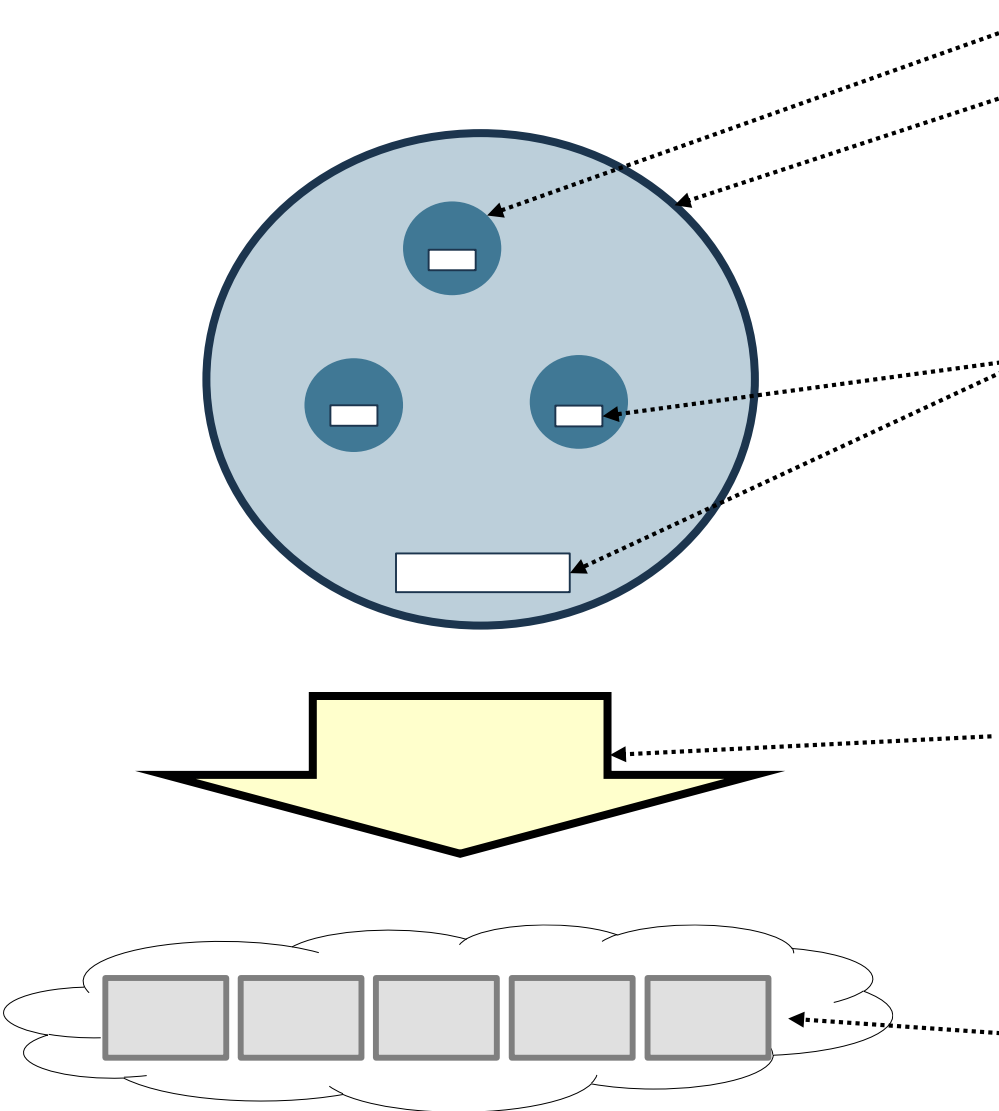


Rechenaufgaben



Rechen-
Ressourcen
(z.B. per IaaS)

Terminologie



Task: Atomare Rechenaufgabe inklusive Ausführungsvorschrift.

Job: Menge an Tasks mit gemeinsamen Ausführungsziel. Die Menge an Tasks ist i.d.R. als DAG mit Tasks als Knoten und Ausführungsabhängigkeiten als Kanten repräsentiert.

Properties: Ausführungsrelevante Eigenschaften der Tasks und Jobs, wie z.B.:

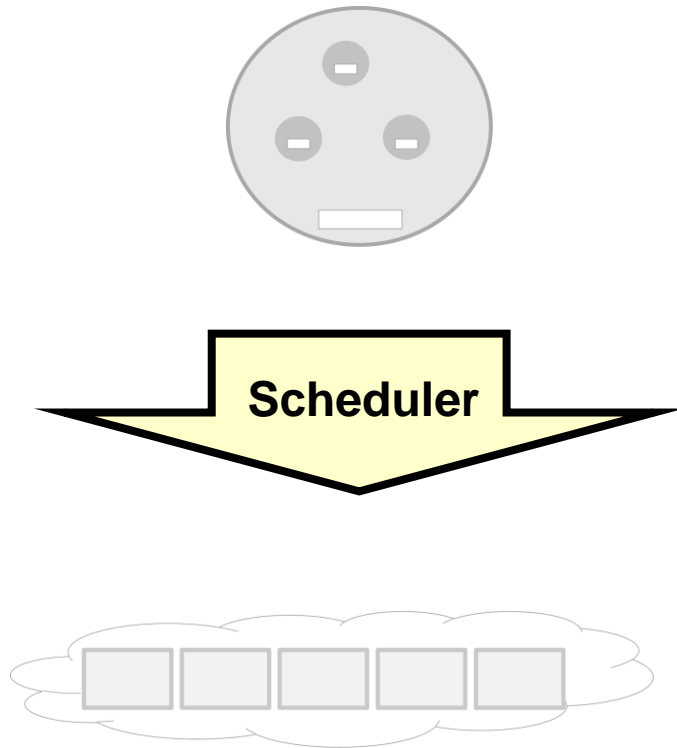
- Task: Ausführungsdauer, Priorität, Ressourcenverbrauch
- Job: Abhängigkeiten der Tasks, Ausführungszeitpunkt

Scheduler: Ausführung von Tasks auf den verfügbaren Resources unter Berücksichtigung der Properties und gegebener

Scheduling-Ziele (z.B. Fairness, Durchsatz, Ressourcenauslastung). Ein Scheduler kann **präemptiv** sein, also die Ausführung von Tasks unterbrechen und neu aufsetzen können.

Resources: Cluster an Rechnern mit CPU-, RAM-, HDD-, Netzwerk-Ressourcen. Ein Rechner stellt seine Ressourcen temporär zur Ausführung eines oder mehrerer Tasks zur Verfügung (**Slot**). Die parallele Ausführung von Tasks ist isoliert zueinander.

Aufgaben eines Cluster-Schedulers:



- **Cluster Awareness:** Die aktuell verfügbaren Ressourcen im Cluster kennen (Knoten inkl. verfügbare CPUs, verfügbarer RAM und Festplattenspeicher sowie Netzwerkbandbreite). Dabei auch auf Elastizität reagieren.
- **Job Allocation:** Zur Ausführung eines Services die passende Menge an Ressourcen für einen bestimmten Zeitraum bestimmen und allokkieren.
- **Job Execution:** Einen Service zuverlässig ausführen und dabei isolieren und überwachen.

Die einfachste Form des Scheduling: Statische Partitionierung



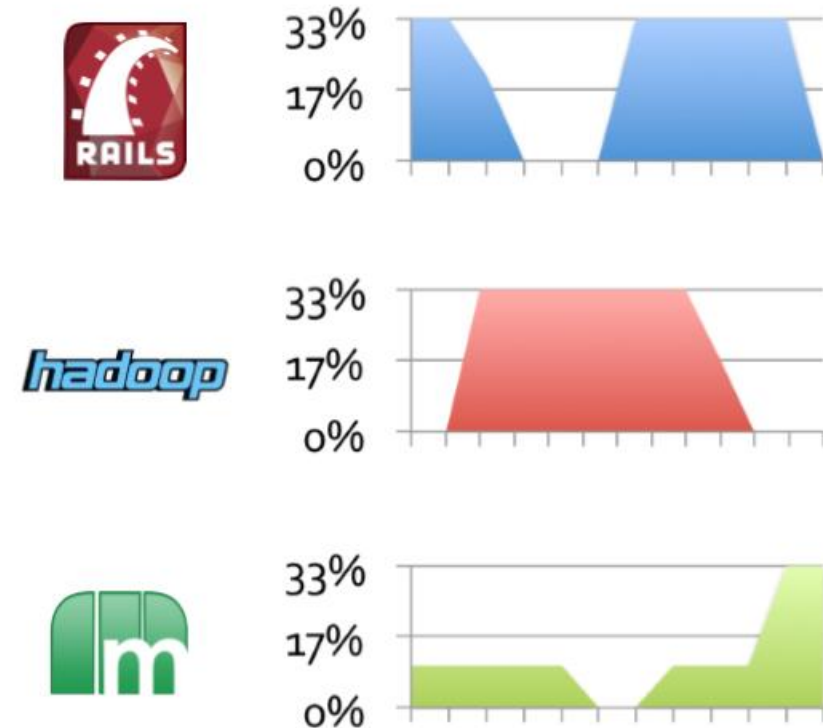
Vorteil:

- Einfach zu realisieren

Nachteile:

- Nicht flexibel bei geänderten Bedürfnissen
- Geringere Auslastung → hohe Opportunitätskosten

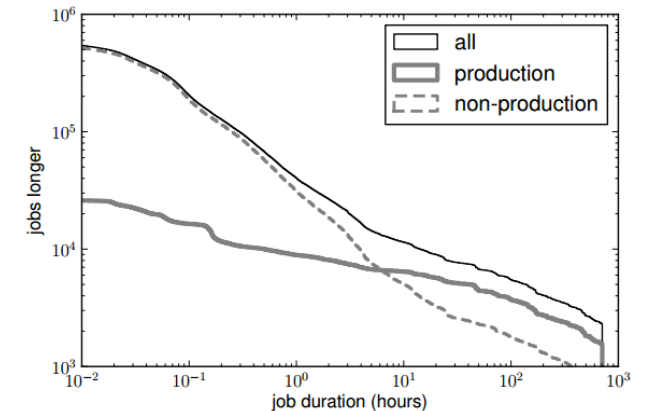
Auslastung pro Knoten



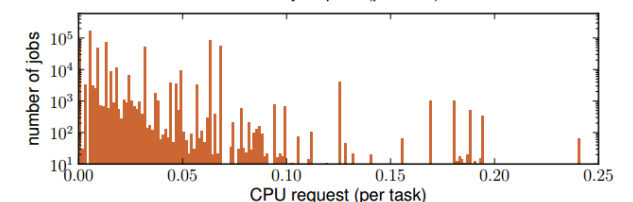
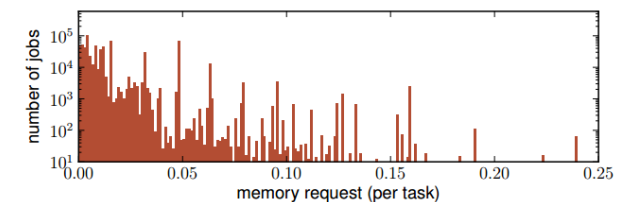
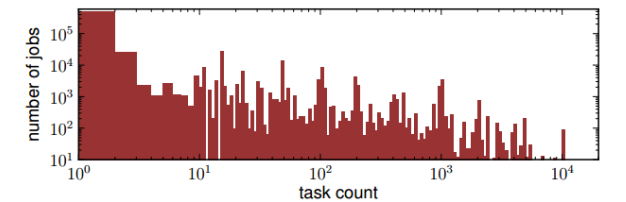
Bildquelle: Practical Considerations for Multi-Level Schedulers, Benjamin Hindman, 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015

Heterogenität im Scheduling

- In typischen Clustern ist die Workload an Jobs sehr heterogen.
- Charakteristische Unterschiede sind:
 - Ausführungsdauer: min, h, d, INF.
 - Ausführungszeit: sofort, später, zu einem Zeitpunkt.
 - Ausführungszweck: Datenverarbeitung, Request-Handling.
 - Ressourcenverbrauch: CPU-, RAM-, HDD-, NW-dominant.
 - Zustand: zustandsbehaftet, zustandslos.
- Zu unterscheiden sind mindestens:
 - **Batch-Jobs:** Ausführungszeit im Minuten- bis Stundenbereich. Eher niedrige Priorität und gut unterbrechbar. Müssen i.d.R. bis zu einem bestimmten Zeitpunkt abgeschlossen sein. Zustandsbehaftet.
 - **Service-Jobs:** Sollen auf unbestimmte Zeit unterbrechungsfrei laufen. Haben hohe Priorität und sollten nicht unterbrochen werden. Sind teilweise auch zustandslos.

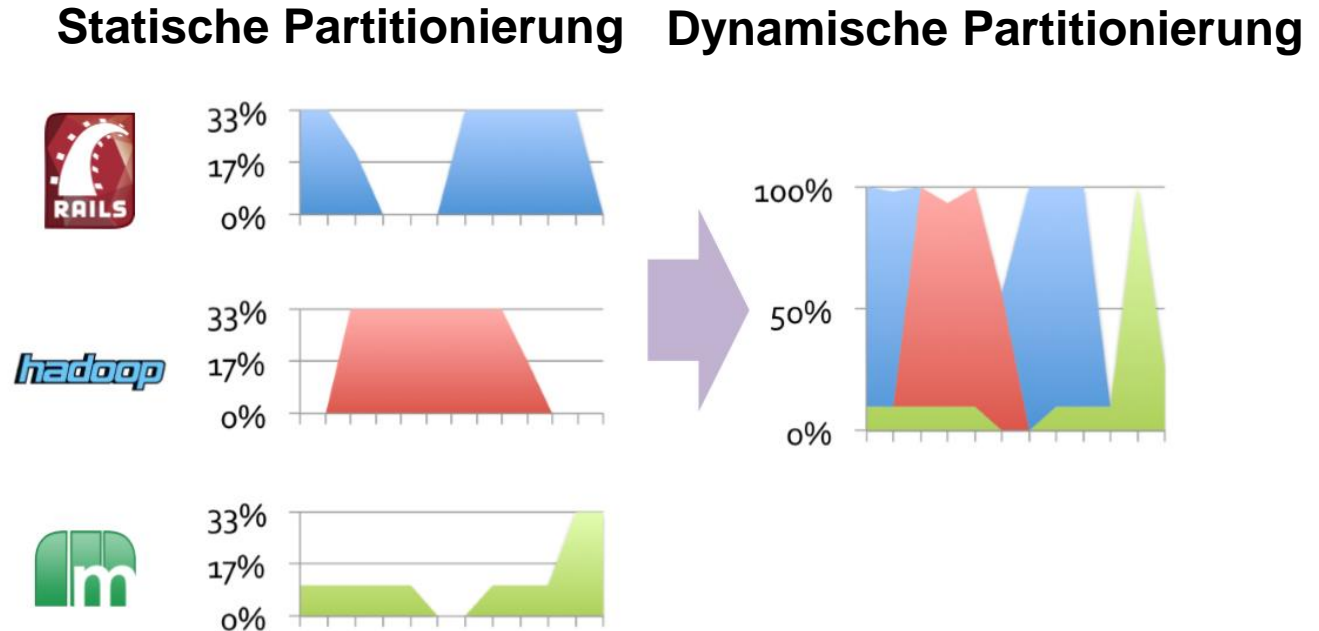
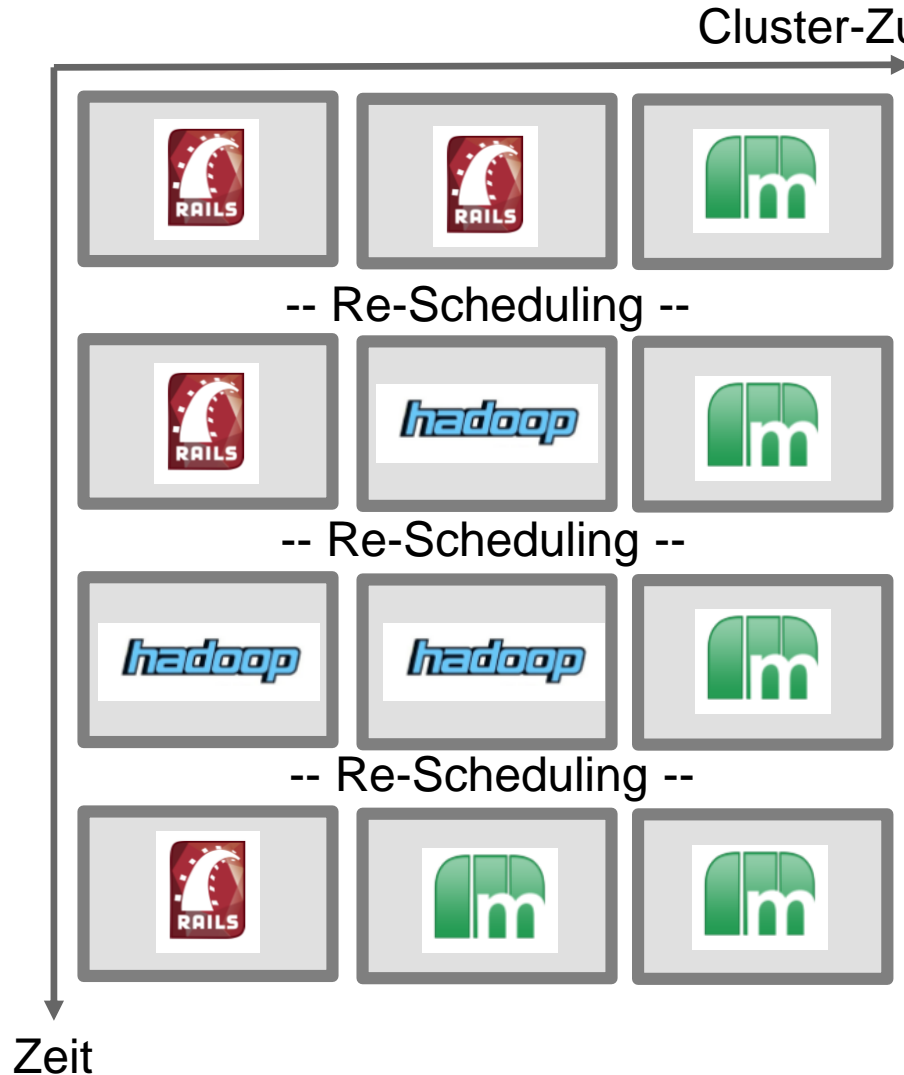


Ausführungsdauer



Ressourcenverbrauch

Bestehende Ressourcen einer Cloud können durch dynamische Partitionierung wesentlich effizienter genutzt werden.



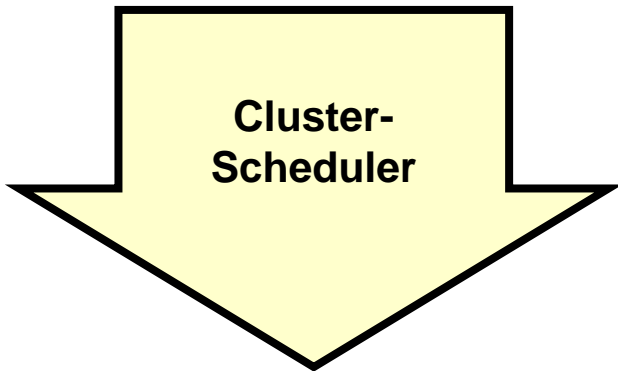
Vorteile der dynamischen Partitionierung:

- Höhere Auslastung der Ressourcen → weniger Ressourcen notwendig → geringere Betriebskosten
- Potenziell schnellere Ausführung einzelner Tasks, da Ressource opportun genutzt werden können.

Ein Cluster-Scheduler: Eingabe, Verarbeitung, Ausgabe.

Eingabe eines Cluster-Schedulers ist Wissen über die Jobs und Tasks (Properties) und über die Ressourcen:

- **Resource Awareness:** Welche Ressourcen stehen zur Verfügung und wie ist der entsprechende Bedarf des Tasks?
- **Data Awareness:** Wo sind die Daten, die ein Task benötigt?
- **QoS Awareness:** Welche Ausführungszeiten müssen garantiert werden?
- **Economy Awareness:** Welche Betriebskosten dürfen nicht überschritten werden?
- **Priority Awareness:** Wie ist die Priorität der Task zueinander?
- **Failure Awareness:** Wie hoch ist die Wahrscheinlichkeit eines Ausfalls? (z.B. da ein Rack oder eine Stromvers.)
- **Experience Awareness:** Wie hat sich ein Tasks in der Vergangenheit verhalten?



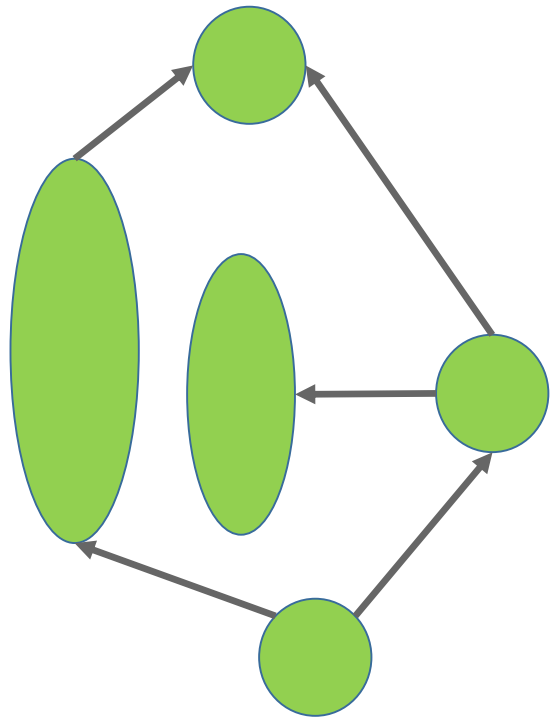
Verarbeitung im Cluster-Scheduler: **Scheduling-Algorithmen** entsprechend der jeweiligen **Scheduling-Ziele**, wie z.B.:

- **Fairness:** Kein Task sollte unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird.
- **Maximaler Durchsatz:** So viele Tasks pro Zeiteinheit wie möglich.
- **Minimale Wartezeit:** Möglichst geringe Zeit von der Übermittlung bis zur Ausführung eines Tasks.
- **Ressourcen-Auslastung:** Möglichst hohe Auslastung der verfügbaren Ressourcen.
- **Zuverlässigkeit:** Ein Task wird garantiert ausgeführt.
- **Geringe End-to-End Ausführungszeit** (z.B. durch Daten-Lokalität und geringe Kommunikationskosten, syn. Makespan)

Ausgabe eines Cluster-Schedulers:
Placement Decision als

- **Slot-Reservierungen**
- **Slot-Stornierungen** (im Fehlerfall, Optimierungsfall, Constraint-Verletzung)

Cluster-Scheduling ist eine Optimierungsaufgabe.



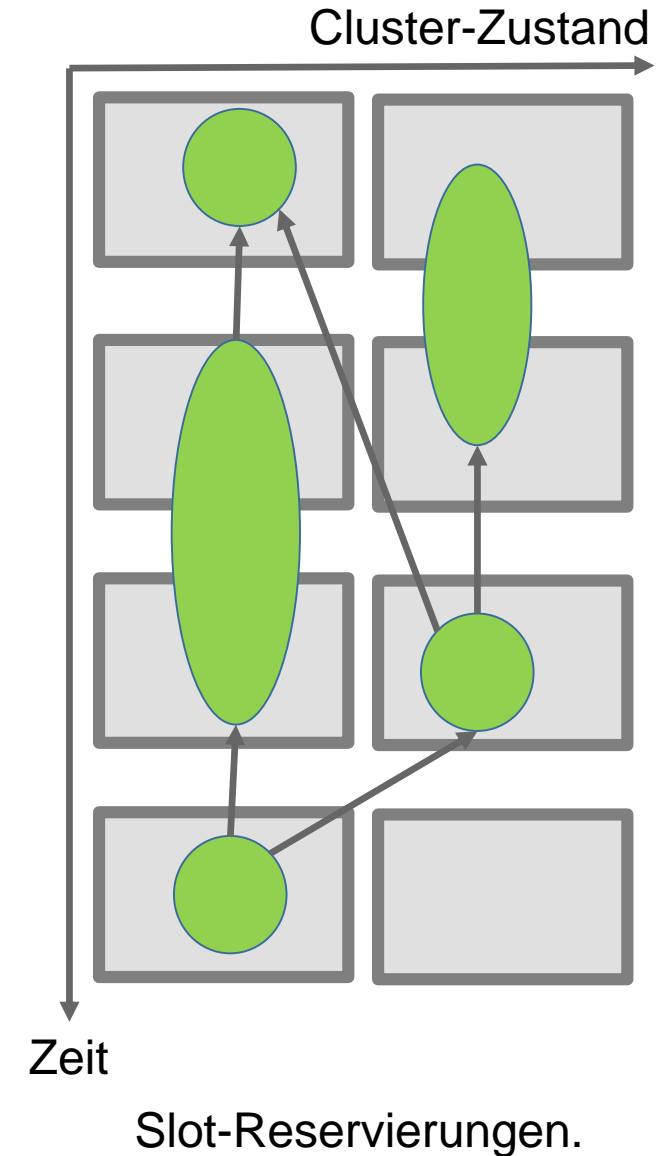
Tasks mit Zeitbedarf (Höhe)
und Ausführungsabhängigkeiten
(Pfeile).

+



Verfügbare Ressourcen.

=



Scheduling ist eine Optimierungsaufgabe, die NP-vollständig ist.

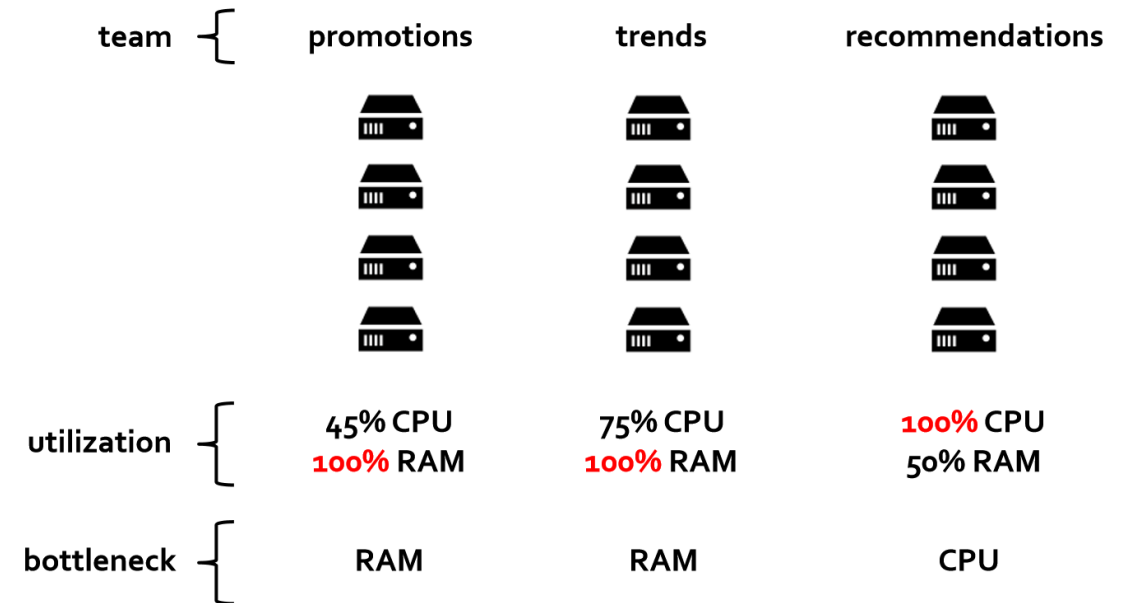
- ... und ist NP-vollständig. Die Optimierungsaufgabe lässt sich auf das Travelling Salesman Problem zurückführen.
- Das bedeutet:
 - Es ist kein Algorithmus bekannt, der eine optimale Lösung garantiert in Polynomialzeit erzeugt.
 - Algorithmus muss für tausende Jobs und tausende Ressourcen skalieren. Optimale Algorithmen, die den Lösungsraum komplett durchsuchen sind nicht praktikabel, da deren Entscheidungszeit zu lange ist für große Eingabemengen ($|Jobs| \times |Ressourcen|$).
 - Praktikable Scheduling-Algorithmen sind somit Algorithmen zur näherungsweisen Lösung des Optimierungsproblems (Heuristiken, Meta-Heuristiken).
- Darüber hinaus kommen Job-Anfragen kontinuierlich an, so dass selbst bei optimalem Algorithmus der Re-Organisationsaufwand pro Job unverhältnismäßig hoch werden kann.

Ein naiver Scheduling-Ansatz

Round Robin

Scheduling-Algorithmus mit Fokus auf Fairness: Dominant Resource Fairness (DRF).

- Aufteilung der Ressourcen an verschiedene „Teams“ (Applikationen, Jobs).
- Ausgangslage: Würden die Ressourcen gleichteilig statisch an die Teams verteilt, so hat jedes Team eine dominante Ressource, die besonders intensiv genutzt wird. Diese dominante Ressource kann durch Beobachtung ermittelt werden und balanciert sich über alle Teams hinweg aus.
- Fairness-Auffassung: Jedes Team bekommt mindestens $1/N$ aller Ressourcen der dominanten Ressourcen. Der Scheduling-Algorithmus ist darauf ausgelegt, die minimal verfügbaren dominanten Ressourcen pro Team zu maximieren.
- Die Fairness kann justiert werden. Jedem Team kann ein gewichteter Anteil der Ressourcen in der statischen Ausgangslage zugesprochen werden. Die Fairness-Auffassung ist dann entsprechend gewichtet.



- Bildquelle: Practical Considerations for Multi-Level Schedulers, Benjamin Hindman, 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015
- Dominant Resource Fairness: Fair Allocation of Multiple Resource Types, Ghodsi et al., 2011
- DRF ist eine Generalisierung des Min-Max Algorithmus für mehrere Ressourcen:
<http://www.ece.rutgers.edu/~marsic/Teaching/CCN/minmax-fairsh.html>

Scheduling-Algorithmus mit Fokus auf Fairness: Capacity Scheduler (CS).

- Es werden Job Queues definiert und zu jeder Queue eine Kapazitätssusage in Ressourcenanteil vom Cluster definiert.
- Der Algorithmus stellt sicher, dass diese Kapazitätssusage eingehalten wird.
- Damit das Cluster dafür nicht statisch partitioniert werden muss, ist ein sog. Over-Commitment von Ressourcen erlaubt.
- Wird durch ein Over-Commitment aber eine Kapazitätssusage gefährdet, werden die over-committeten Ressourcen entzogen. Hierfür ist also ein präemptiver Scheduler notwendig.

Scheduling-Algorithmus mit Balance aus Fairness und Ausführungszeit: Tetris.

■ Schritt 1: Effizientes multidimensionales Bin Packing → hohe Auslastung

■ Gegeben:

- Vektor des Ressourcenbedarfs entlang der Dimensionen des Ressourcenangebots wie z.B. CPU, RAM, Disk IO, NW IO.

- Raum des Ressourcenangebots entsprechend der Dimensionen plus einer diskretisierten Zeit-Dimension.

- Gesucht: Wie packt man die Ressourcenbedarfe so in den Raum des Ressourcenangebots, dass möglichst viel freier Raum übrig bleibt (hohe Packungsdichte, geringe Fragmentierung).

- Unscharfe Lösung über eine Heuristik.

■ Schritt 2: SRT (Smallest Remaining Time) → geringe Makespan

- Tasks werden entsprechend ihrer aufsteigend sortierten Rest-Laufzeit höher priorisiert bei der Ausführung.

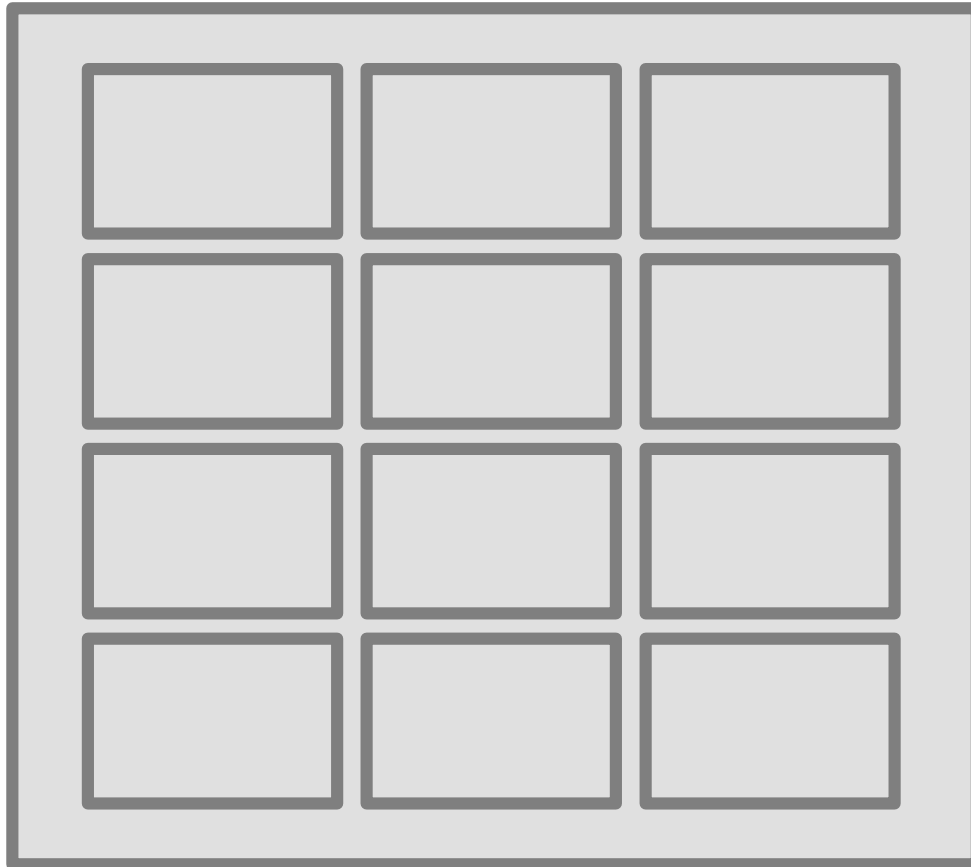
■ Fairness Knob → Fairness vs. Performance

- Es kann ein Wert zwischen 0 und 1 angegeben werden, der Fairness und Performance (bzgl. Makespan) balanciert. Je mehr Fairness gefordert wird desto höher werden Jobs priorisiert, die weiter Weg von einem fairen Zustand (im Sinne von DRF) sind.

- Deutlicher Performance-Vorteil bzgl. Makespan gegenüber DRF, gerade bei hoher Last im Cluster.

Architektur eines Cluster-Schedulers

The Datacenter as a Computer



Idee: Ein Cluster sieht von Außen aus wie ein großer Computer.

Konsequenz: Es müssen als Fundament viele Konzepte klassischer Betriebssysteme übertragen werden (ein Cluster-Betriebssystem). Das gilt insbesondere auch für das Scheduling.

Buch: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 2009, Luiz André Barroso und Urs Hölzle

Eine konzeptionelle Architektur für Cluster-Scheduler.

Job Queue:

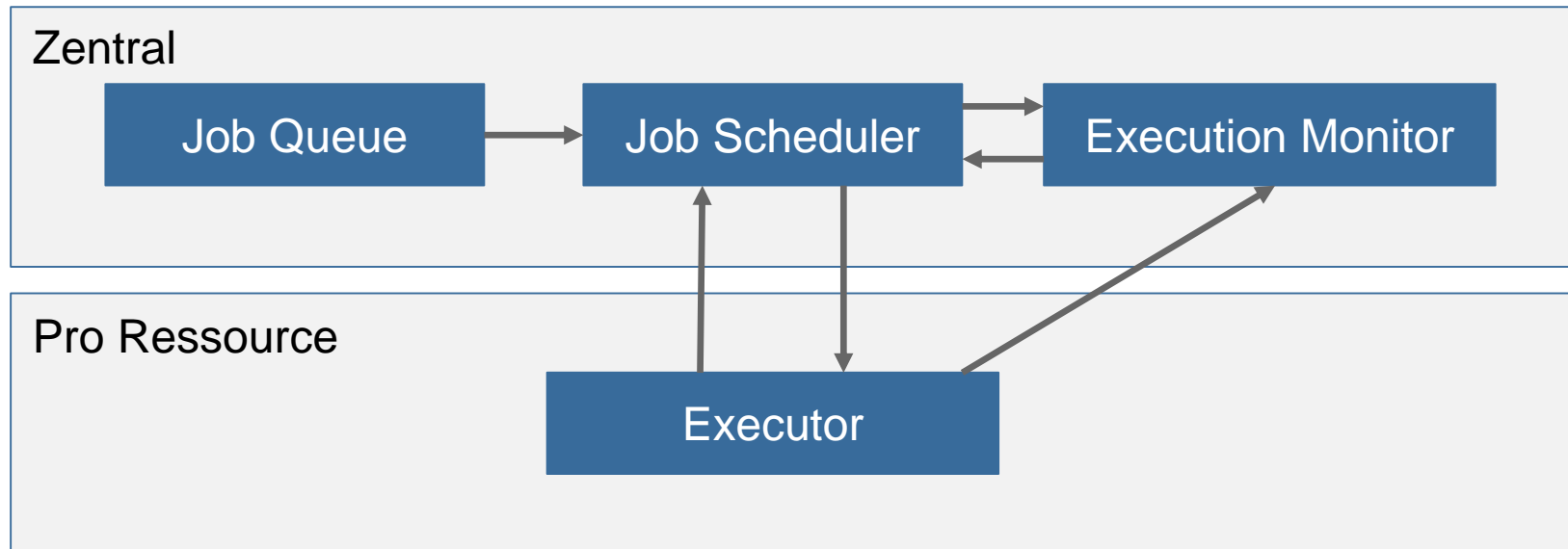
- Eingehende Jobs zur Ausführung
- Events zu eingegangenen Jobs

Job Scheduler:

- Jobs einplanen
- Taskausführung steuern

Execution Monitor:

- Task-Ausführung überwachen
- Ressourcen überwachen



Anforderungen:

- Performance
 - Geringe Queing-Time
 - Geringe Decision-Time
 - Geringe Ausführungslatenz
- Hoch-Verfügbarkeit und Fehlertoleranz
- Skalierbarkeit bzgl. Anzahl an Jobs und verfügbaren Ressourcen.

Executor:

- Task ausführen
- Informationen zur Ressource bereitstellen

Scheduler-Architektur.

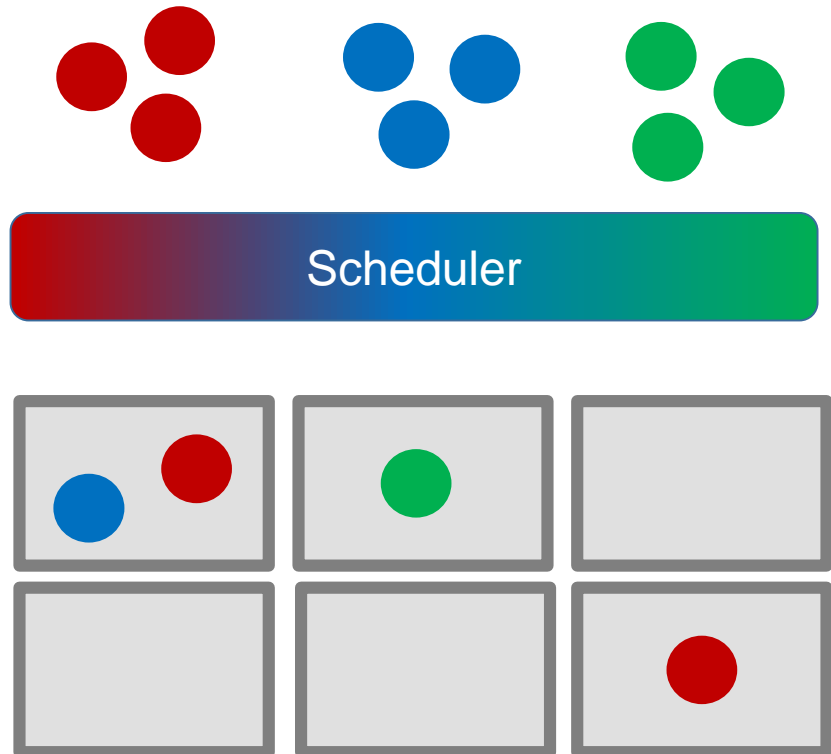
Variante 1: kein Scheduler.



Statische Partitionierung.

Scheduler-Architektur.

Variante 2: Monolithischer Scheduler.



▪ **Google Borg**
Large-scale cluster management at Google with Borg, Verma et al., 2015
▪ **Hadoop YARN**
▪ **Kubernetes**
▪ **Docker Swarm**

Vorteile:

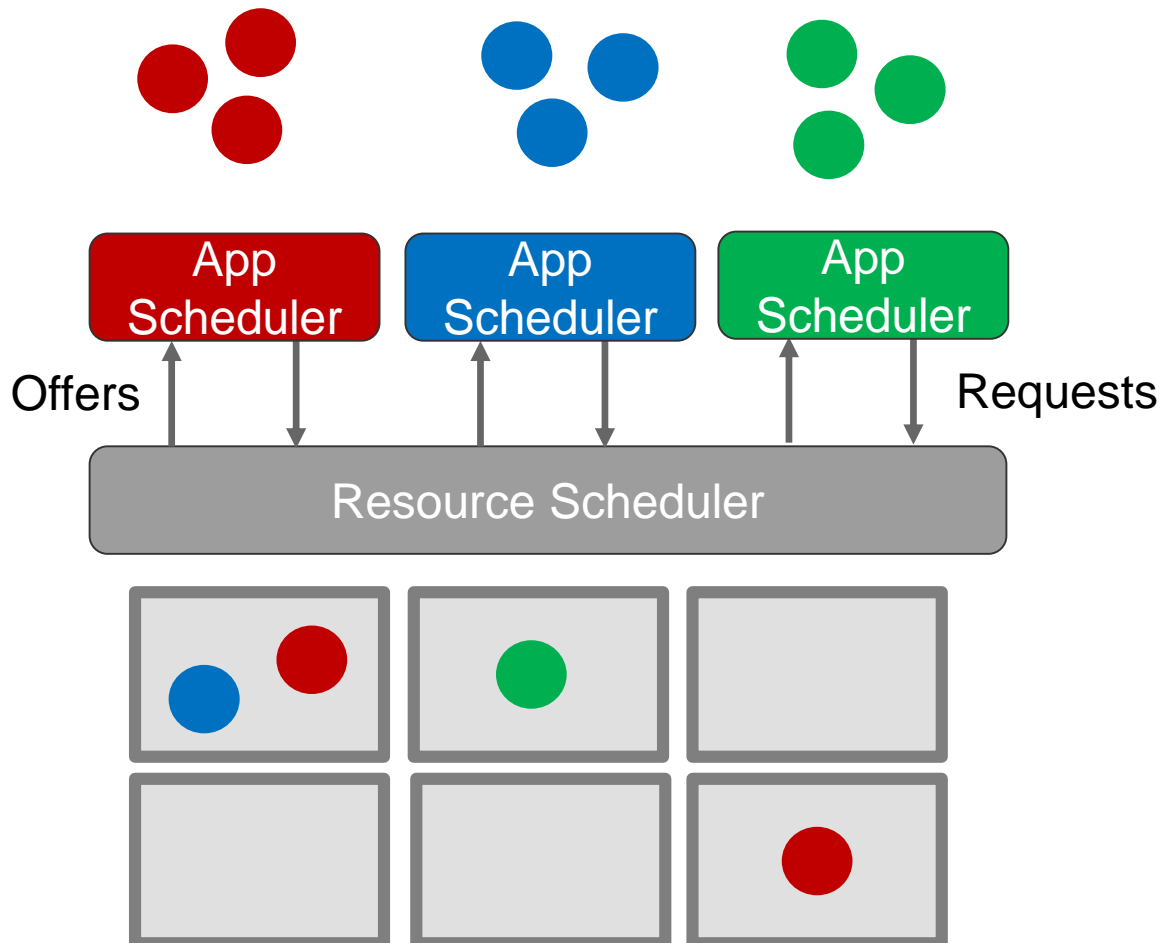
- Globale Optimierungsstrategien einfach möglich.

Nachteile:

- Heterogenes Scheduling für heterogene Jobs schwierig:
 - Komplexe und umfangreiche Implementierung notwendig
 - ... oder homogenes Scheduling von geringerer Effizienz.
- Potenzielles Skalierbarkeits-Bottleneck.

Scheduler-Architektur.

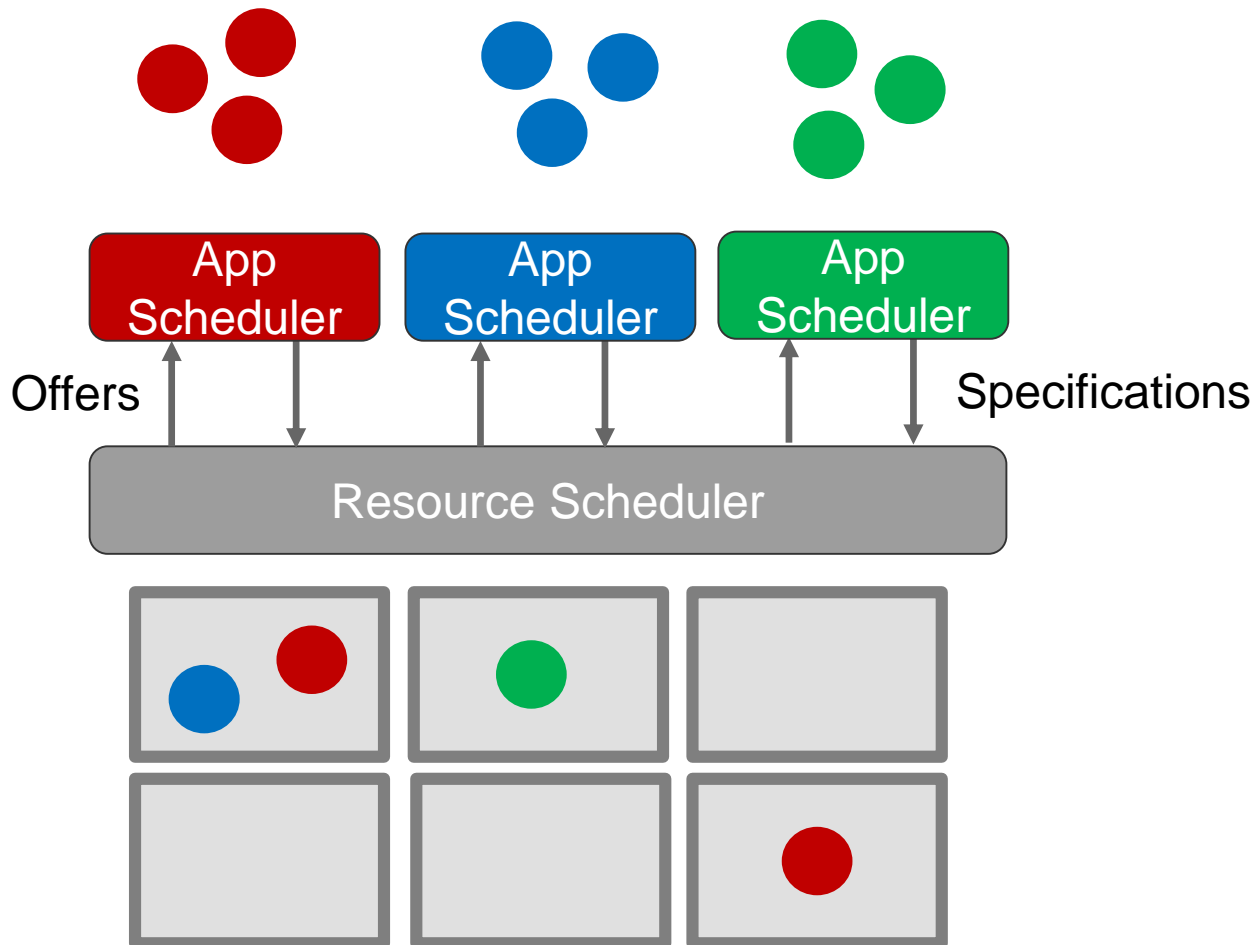
Variante 3: 2-Level-Scheduler.



- Auftrennung der Scheduling-Logik in einen Resource Scheduler und einen App Scheduler.
- Der **Resource Scheduler** kennt alle verfügbaren Ressourcen und darf diese allokalieren. Er nimmt Ressourcen-Anfragen (Requests) entgegen und unterbreitet entsprechend einer Scheduling-Policy (definierte Scheduling-Ziele) Ressourcen-Angebote (Offers).
- Der **App Scheduler** nimmt Jobs entgegen und „übersetzt“ diese in Ressourcen-Anfragen und wählt applikationsspezifisch die passenden Ressourcen-Angebote aus.
- Offers sind eine zeitlich beschränkte Allokation von Ressourcen, die explizit angenommen werden muss.
- Grundsätzlich **pessimistische Strategie**: Disjunkte Offers. I.d.R. sind aber auch optimistische Offers verfügbar, bei denen eine gewisse Überschneidung erlaubt ist.
- Im Sinne der Fairness kann ein prozentualer Anteil der Ressourcen für einen App Scheduler garantiert werden.

Scheduler-Architektur.

Variante 3: 2-Level-Scheduler.



Apache Mesos
Mesos: A Platform for Fine-Grained Resource Sharing in the
Data Center, Hindman et al., 2010

■ Vorteile:

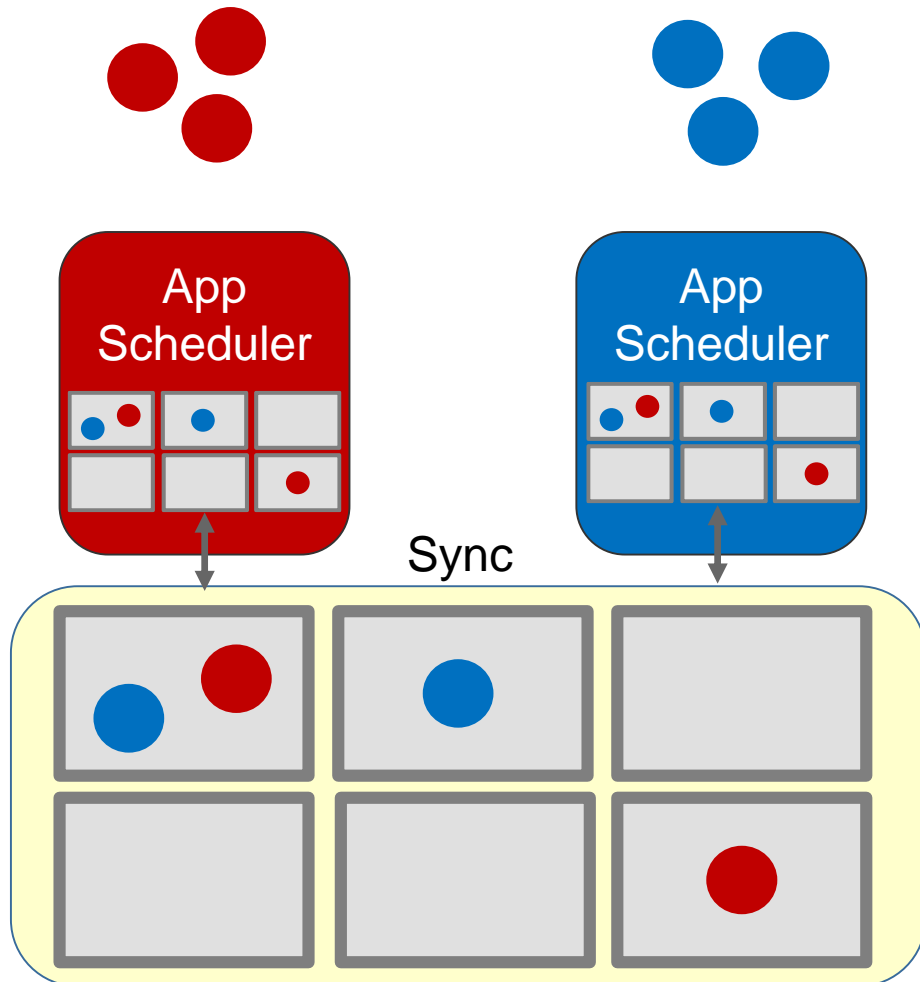
- Nachgewiesene Skalierbarkeit auf tausende von Knoten (z.B. Twitter, Airbnb, Apple Siri).
- Flexible Architektur für heterogene Scheduling-Logiken.

■ Nachteile:

- App-Scheduler übergreifende Logiken nur schwer zu realisieren (z.B. globaler Ausführungsverzicht oder Gang-Scheduling)

Scheduler-Architektur.

Variante 4: Shared-State-Scheduler.

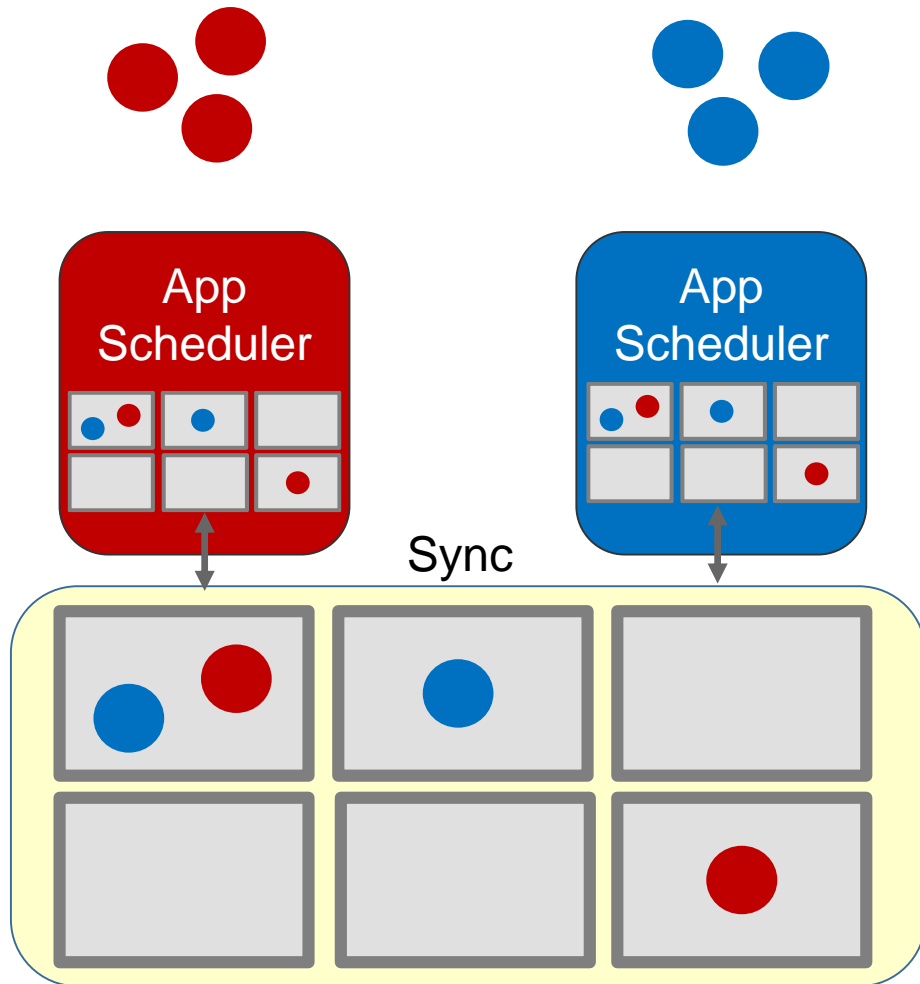


- Es gibt ausschließlich applikationsspezifische Scheduler.
- Die **App Scheduler** synchronisieren kontinuierlich den aktuellen Zustand des Clusters (**Cluster-Zustand**: Job-Allokationen und verfügbare Ressourcen).
- Jeder App Scheduler entscheidet die Platzierung von Tasks auf Basis seines aktuellen Cluster-Zustands.
- **Optimistische Strategie**: Ein zentraler Koordinationsdienst erkennt Konflikte im Scheduling und löst diese auf, indem er Zustands-Änderungen nur für einen der beteiligten App Scheduler erlaubt und für die anderen App Scheduler einen Fehler meldet.

Scheduler-Architektur.

Variante 4: Shared-State-Scheduler.

Google Omega
Omega: flexible, scalable schedulers for large
compute clusters, Schwarzkopf et al., 2013



■ Vorteile:

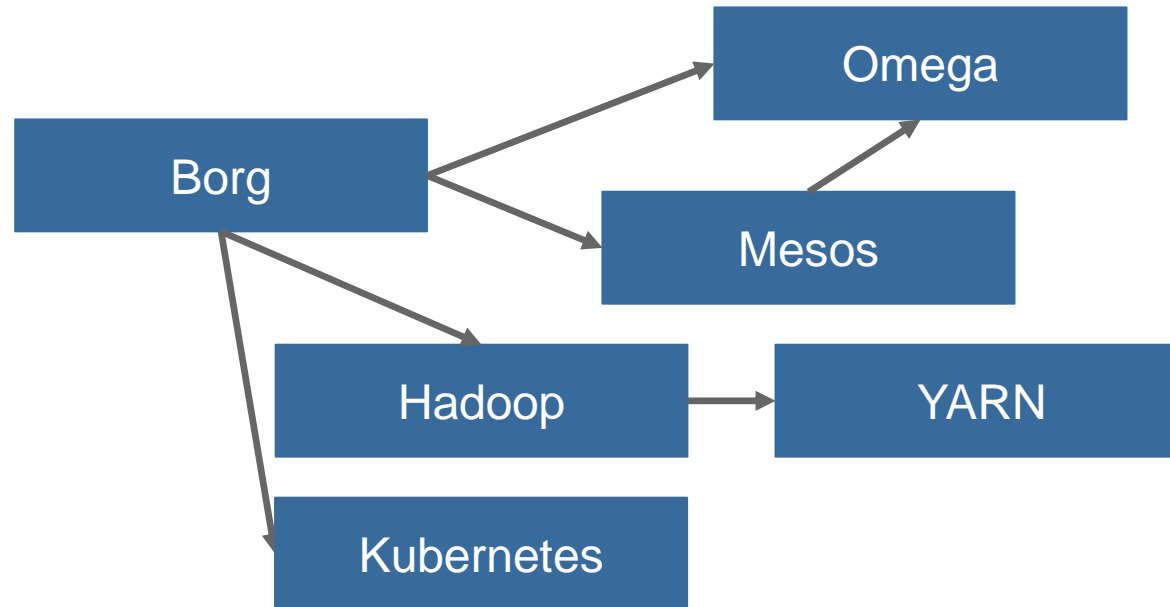
- Tendenziell geringerer Kommunikations-Overhead.

■ Nachteile:

- Komplettes Scheduling muss pro App Scheduler entwickelt werden.
- Keine globalen Scheduling-Ziele (z.B. Fairness) möglich.
- Skalierbarkeit in großen Clustern unklar, da noch nicht in der Praxis erprobt und insbesondere Auswirkung bei hoher Anzahl an Konflikten ungeklärt.

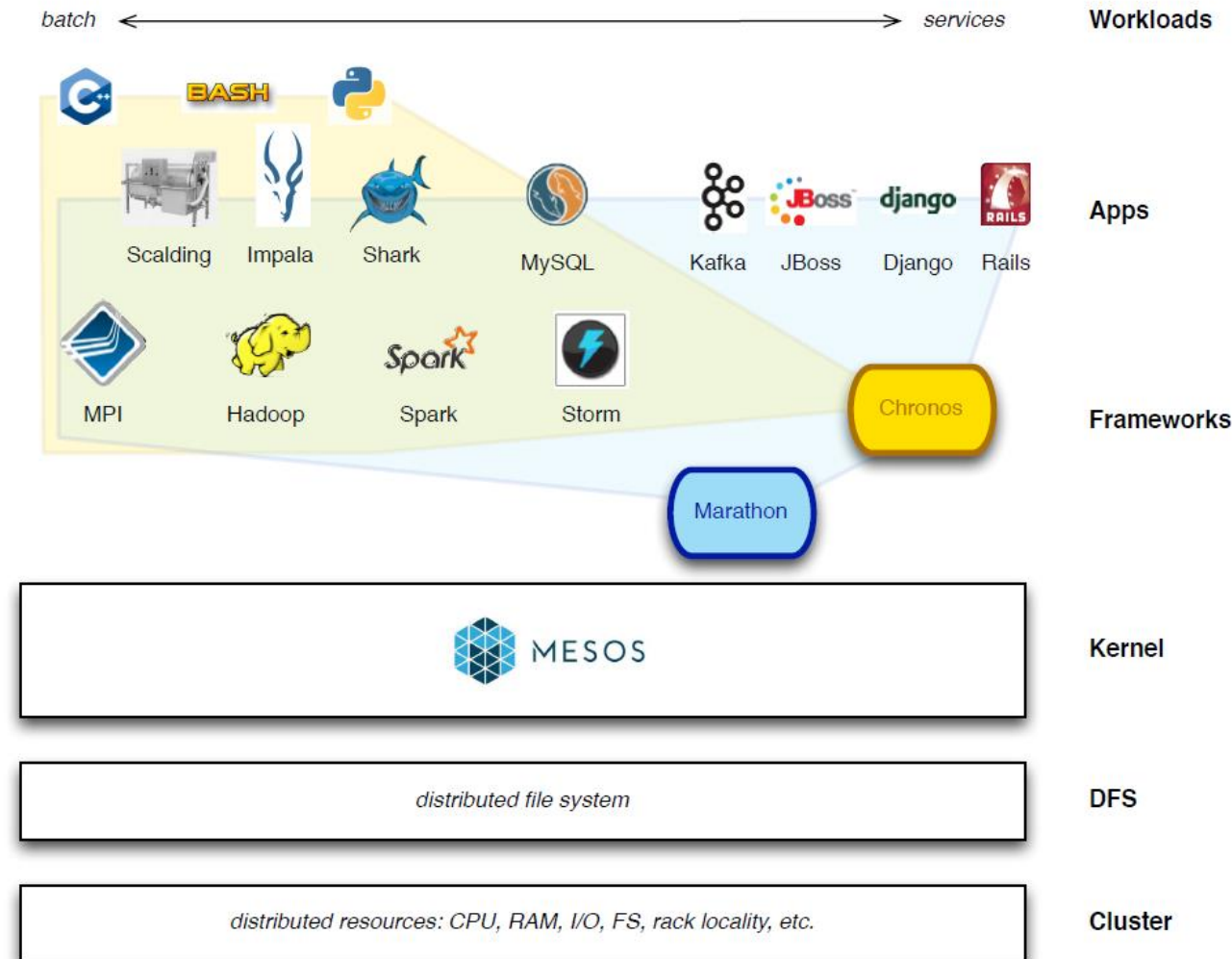
Cluster-Scheduler: Mesos

Die Genealogie der Cluster-Scheduler.



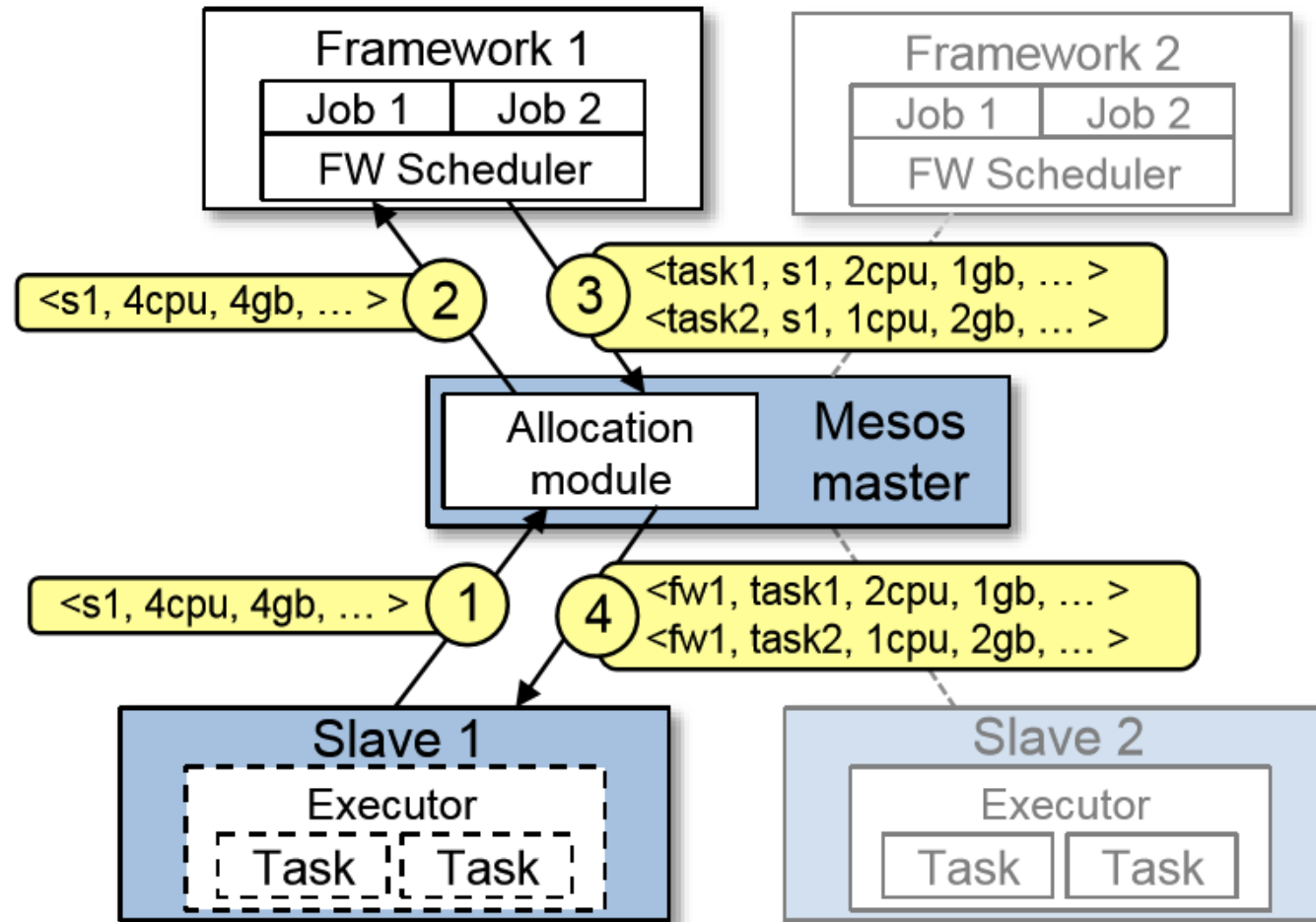
Apache Mesos

- Entstanden an der UC Berkeley im Rahmen der Arbeiten von Ben Hindeman. 1. Release 2009.
- Open Source Projekt unter der Apache Lizenz 2.0.
- Ist im Kern ein Cluster-Scheduler und fasst sich selbst als Cluster-Betriebssystem auf.
- Eine kommerzielle Variante wird unter dem Namen DCOS von der Firma Mesosphere, Inc. entwickelt und vertrieben.
- 2-Level-Scheduler mit DRF als Default-Scheduling-Algorithmus.
- Alle Bestandteile von Mesos können ausfallsicher ausgelegt werden.
- Wird im großen Stil eingesetzt bei Twitter, Apple, Microsoft, Verizon, CERN, Airbnb, ...
- Alle Teile sind per REST-API zugänglich. Task-Isolation per Docker oder eigenem Mechanismus.



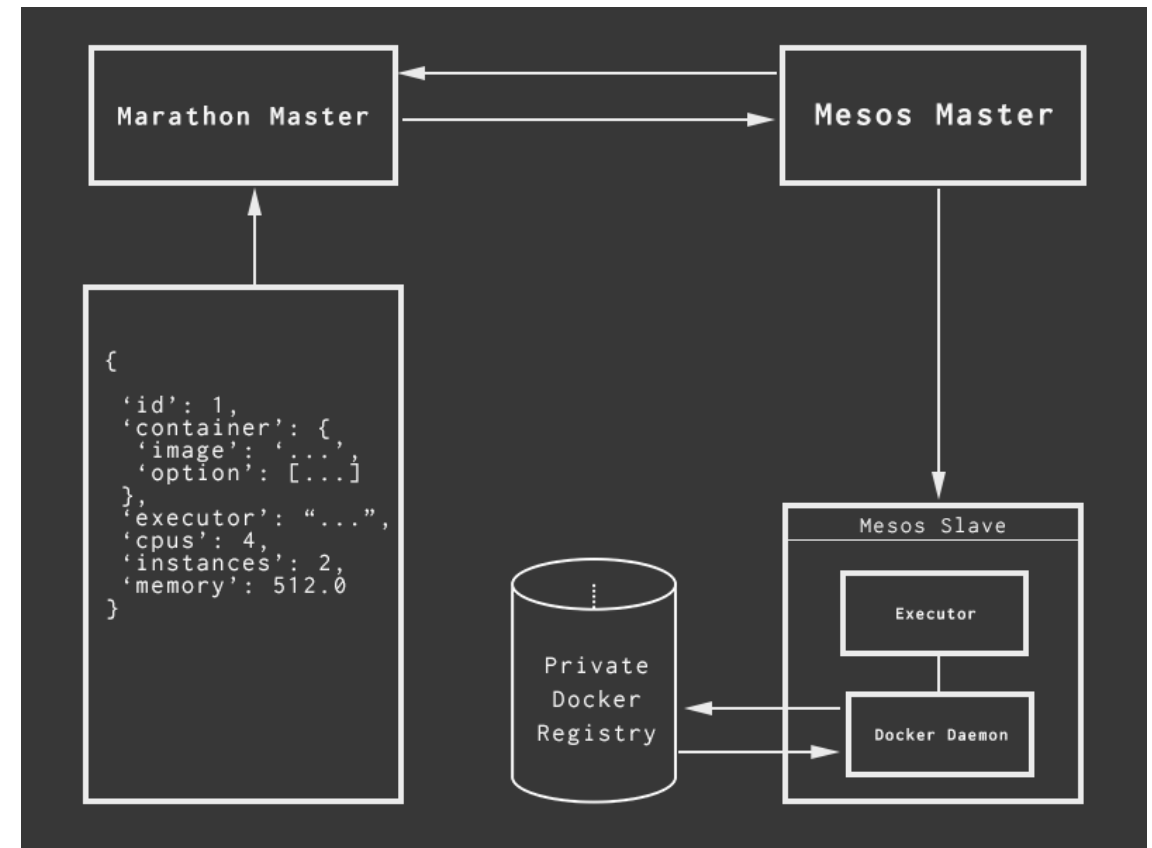
Quelle: <https://www.typesafe.com/blog/play-framework-grid-deployment-with-mesos>

Apache Mesos: Zusammenspiel der Bausteine.



Marathon ist der 2nd-Level-Scheduler in Mesos, der auf die Ausführung von zustandslosen Services ausgelegt ist.

- Wurde initial von Tobi Knaup entwickelt als Mesos-Aufbau um langlaufende, zustandslose Services zuverlässig und komfortabel ausführen zu können.
- Besitzt eigenständige Web-UI und REST-API.
- Prozess werden kontinuierlich am Leben gehalten. Terminiert ein Prozess, so wird er automatisch wieder gestartet.
- Mechanismus für Health-Checking von Services.
- Eingebauter Mechanismus für Service Discovery und Load Balancing.



Quellen

Videos zum Thema Cluster Scheduling.

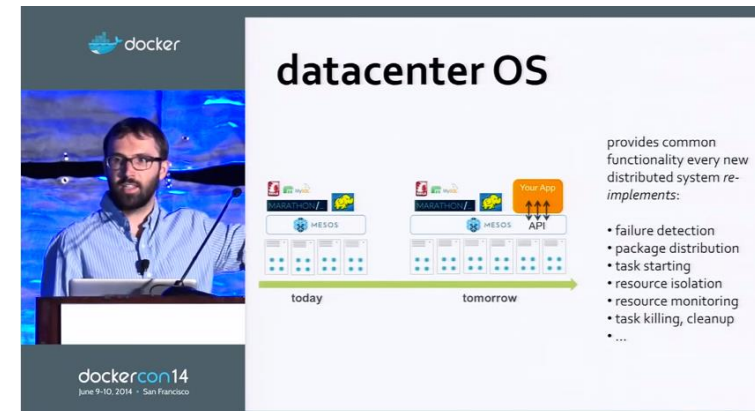
- Scheduling in Cloud by Ms. Mala Kalra, <https://youtu.be/hhwCc5yPylg>



- Omega: flexible, scalable schedulers for large compute clusters, <https://youtu.be/XsXlm4wmB6o>



- Cluster Management and Containerization, Ben Hindman, <https://youtu.be/F1-UEIG7u5g>



Literatur

- Omega: flexible, scalable schedulers for large compute clusters, Schwarzkopf et al., 2013
- Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, Hindman et al., 2010
- Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis, Charles Reiss et al., 2012
- Scheduling Algorithms for Grid Computing: State of the Art and Open Problems, Dong et al., 2006
- Scheduling in Grid Computing Environment, Prajapati, 2014
- A Survey on Task Scheduling For Parallel Workloads in the Cloud Computing System, Abhijeet Tikar et al., 2014
- Multi-Resource Packing for Cluster Schedulers Paper, Grandl, 2014
- Dominant Resource Fairness: Fair Allocation of Multiple Resource Types, Ghodsi et al., 2011
- Mesos, Omega, Borg – A Survey: http://www.umbrant.com/blog/2015/mesos_omega_borg_survey.html
- Übersicht zum Scheduling in Mesos: <http://cloudarchitectmusings.com/2015/04/08/playing-traffic-cop-resource-allocation-in-apache-mesos>