Nomad and next-generation application architectures
Armon Dadgar
Founder and CTO
@armon
HashiCorp Suite

**CONNECT**
Infrastructure & applications
- Consul

**RUN**
Applications
- Nomad

**SECURE**
Infrastructure & applications
- Vault

**PROVISION**
Infrastructure
- Terraform
- Packer
- Vagrant
Nomad Cluster Manager
Scheduler
Nomad

Cluster Manager

Scheduler
Schedulers map a set of work to a set of resources.
CPU Scheduler

**Work (Input)**
- Web Server - Thread 1
- Web Server - Thread 2
- Redis - Thread 1
- Kernel - Thread 1

**Resources**
- CPU - Core 1
- CPU - Core 2
## Schedulers in the Wild

<table>
<thead>
<tr>
<th>Type</th>
<th>Work</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Scheduler</td>
<td>Threads</td>
<td>Physical Cores</td>
</tr>
<tr>
<td>AWS EC2 / OpenStack Nova</td>
<td>Virtual Machines</td>
<td>Hypervisors</td>
</tr>
<tr>
<td>Hadoop YARN</td>
<td>MapReduce Jobs</td>
<td>Client Nodes</td>
</tr>
<tr>
<td>Cluster Scheduler</td>
<td>Applications</td>
<td>Servers</td>
</tr>
</tbody>
</table>
Advantages

Higher Resource Utilization

Decouple Work from Resources

Better Quality of Service
Advantages

- Higher Resource Utilization
- Decouple Work from Resources
- Better Quality of Service
- Bin Packing
- Over-Subscription
- Job Queueing
Advantages

Higher Resource Utilization
Decouple Work from Resources
Better Quality of Service

Abstraction
API Contracts
Packaging
Advantages

- Higher Resource Utilization
- Decouple Work from Resources
- Better Quality of Service
- Priorities
- Resource Isolation
- Pre-emption
Nomad
Nomad

Cluster Scheduler
Easily Deploy Applications
Operationally Simple
Built for Scale
API for Next-Gen Patterns
job "redis" {
    datacenters = ["us-east-1"]

    task "redis" {
        driver = "docker"
        config { image = "redis:latest" }

        resources {
            cpu = 500 # Mhz
            memory = 256 # MB

            network {
                mbits = 10
                port "redis" {}
            }
        }
    }
}
Declares what to run
Nomad determines **where** and manages **how** to run
Nomad abstracts work from resources
<table>
<thead>
<tr>
<th>OS</th>
<th>Workloads</th>
<th>Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>Long Running Service</td>
<td>Docker / Rkt / LXC</td>
</tr>
<tr>
<td>Linux</td>
<td>Short Lived Batch</td>
<td>Qemu / KVM</td>
</tr>
<tr>
<td>BSD</td>
<td>Periodic Cron</td>
<td>“exec” cgroups+chroot</td>
</tr>
<tr>
<td>Solaris</td>
<td>System Agents</td>
<td>Static Binaries / Fat JARs</td>
</tr>
</tbody>
</table>
Nomad

Declarative Jobs
Infrastructure as Code
Consul Integration
Vault Integration
Composable vs Platform
Empowers developers by de-coupling operators
Operationally Simple
Built on Experience

GOSSIP

CONSENSUS
Serf

Cluster Management
Gossip Based (P2P)
Membership
Failure Detection
Event System
Large Scale
Production Hardened
Simple Clustering and Federation
Service Discovery
Configuration
Coordination (Locking)
Central Servers + Distributed Clients
Multi-Datacenter
Raft Consensus
Large Scale
Production Hardened
Nomad

- Single Binary
- No Dependencies
- Highly Available
- Multi-DC/Region Support
Built for Scale
Built on Experience

Mature Libraries

Proven Design Patterns

Lacking Scheduling Logic
Built on Research
**Large-scale cluster management at Google with Borg**

**Abstract:** Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines. It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

---

**Omega: flexible, scalable schedulers for large compute clusters**

**Abstract:** Increasing scale and the need for rapid response to changing requirements are hard to meet with current monolithic cluster scheduler architectures. This restricts the rate at which new features can be deployed, decreases efficiency and utilization, and will eventually limit cluster growth. We present a novel approach to address these needs using parallelism, shared state, and lock-free optimistic concurrency control. We compare this approach to existing cluster scheduler designs, evaluate how much interference between schedulers occurs and how much it matters in practice, present some techniques to alleviate it, and finally discuss a use case highlighting the advantages of our approach — all driven by real-life Google production workloads.

---

**Sparrow: Low Latency Scheduling for Interactive Cluster Services**

**Posted on March 28, 2012 by Patrick Wendell**

The Sparrow project introduces a distributed cluster scheduling architecture which supports ultra-high throughput, low latency task scheduling. By supporting very low-latency tasks (and their associated high rate of task turnover), Sparrow enables a new class of cluster applications which analyze data at unprecedented volume and speed. The Sparrow project is under active development and maintained in our public github repository.

---

**Mesos – Dynamic Resource Sharing for Clusters**

**Posted on November 21, 2011 by kilov**

Mesos is a cluster manager that provides efficient resource isolation and sharing across distributed applications, or frameworks. It can run Hadoop, MPI, hyperic, Spark (a new framework for low-latency interactive and iterative jobs), and other applications. Mesos is open source in the Apache Incubator.
Single Region Architecture

CLIENT (DC1) → SERVER (FOLLOWER)

CLIENT (DC2) → SERVER (LEADER)

CLIENT (DC3) → SERVER (FOLLOWER)

RPC

REPICATION

FORWARDING
Multi Region Architecture

REGION A
SERVER Follower
SERVER Leader
SERVER Follower
GOSSIP
REGION B
SERVER Follower
SERVER Leader
SERVER Follower
REGION FORWARDING
REPLICATION
FORWARDING
REPLICATION
FORWARDING
REPLICATION
FORWARDING
100’s of Regions
10,000’s of Clients per Region
1000’s of Jobs per Region
Nomad

Inspired by Google Omega

Optimistic Concurrency

Service & Batch workloads

Pluggable Architecture
Nomad

Million Container Challenge

1,000 Jobs
1,000 Tasks per Job
5,000 Hosts on GCE
1,000,000 Containers
“640 KB ought to be enough for anybody.”

- Bill Gates
2nd Largest Hedge Fund

18K Cores

5 Hours

2,200 Containers/second
Next-Gen Patterns
Micro-Service  SOA Spectrum  Monolith
Utility of Service

Micro-Service

SOA Spectrum

Monolith
Monoliths have high application complexity
Microservices have high operational complexity
Abstractions allow us to scale complexity
Frameworks :: Monoliths
Schedulers :: Services
Schedulers abstract details, focus on service composition
Side Cars

- Sidecar or Co-Process Pattern
- Application that runs alongside “main” process
- Nomad “Task Group”
- Borg “Alloc”
- Kubernetes “Pod”
Side Cars

- Configuration (Consul-Template)
- Logging Agents (Splunk, CloudWatch)
- Telemetry Agents (Datadog)
- Service Mesh (Envoy, Linkerd)
- Load Balancing (HAProxy, Nginx, Fabio)
Nomad

Transparent Scheduling
API awareness
Dynamic Behavior
Queues

Producers

Web Server
API Server

Queue

Consumers

Worker 1
Worker 2
Worker N
Queues

- Workers are **online** service doing **batch** work
- Workers provisioned in advance
- N+1 instances for high availability
- Typically idle or underutilized
Nomad Dispatch

- “Dispatch” a worker for each incoming event
- Consumer launched on-demand and terminates when done
- Publisher shielded from implementation detail
- Nomad job acts like a future, queues when busy
- Avoids underutilization
job "my-dispatch" {
    datacenter = ["dc1"]
    type = "batch"
    parameterized {
        meta_required = ["input"]
    }
}

task "worker" {
    driver = "docker"
    config = {
        image = "myworker:latest"
        args = ["--input", "${NOMAD_META_INPUT}"]
    }
}
Dispatch Job → Register → Nomad Server
Dispatch Job \(\rightarrow\) Register \(\rightarrow\) Nomad Server \(\rightarrow\) Dispatch \(\rightarrow\) Web Server
Dispatch Job -> Register -> Nomad Server -> Schedule -> Worker 1

Dispatch

Web Server
Function-as-a-Service

- AWS Lambda
- Small Granularity
- Low Volume, Latency Insensitive => Nomad Dispatch
- High Volume, Latency Sensitive => Setup Overhead Prohibitive
FaaS / Serverless

- Process multiple events per worker
- Dynamically scale workers
- Queue messages to avoid dropping
Dispatch Job \rightarrow \text{Register} \rightarrow \text{Nomad Server} \rightarrow \text{Controller}
Nomad
Register
Dispatch Job

Nomad Server

Dispatch

Web Server

Controller

Push
Nomad Server Register Dispatch Job

Web Server Push Controller

Nomad Server Schedule Worker 1

Dispatch

Controller Pull
Big Data Processing

- Large scale batch workload
- Graph of processing steps
- Each phase dynamic size
- Programmatically setup/teardown workers
- Native Spark Integration!

**Figure 5:** Segregating prod and non-prod work into different cells would need more machines. Both graphs show how many extra machines would be needed if the prod and non-prod workloads were sent to separate cells, expressed as a percentage of the minimum number of machines required to run the workload in a single cell. In this, and subsequent CDF plots, the value shown for each cell is derived from the 90%tile of the different cell sizes our experiment trials produced; the error bars show the complete range of values from the trials.
Scheduler API blurs line between Application and Infrastructure
Nomad enables dynamic behavior while optimizing utilization
Nomad
Cluster Scheduler
Easily Deploy Applications
Operationally Simple
Built for Scale
API for Next-Gen Patterns
Thanks!