Who guards the guardians?

Engineering reliability in cluster orchestrators
Distributed systems
Efficiency
Cluster Orchestrators
Orchestrators map work to resources

Jobs and Nodes

Placements
What is Nomad?

**Workflows not Technologies**

- Containerized and Legacy Applications
- Pluggable Drivers

**Easy to Use**

- No external state store
- **Single Binary**

**Scalable and Performant**

- 10,000+ Node Deployments
- Schedule 5,500 Placements a Second (1M/18.1s)
job "my_job" {
  datacenters = ["us-west-1", "us-east-1"]
  type = "service"

  group "web" {
    count = 5

    task "frontend" {

      driver = "docker"
      config { image = "hashicorp/web-frontend" }

      resources {
        cpu = 500 # MHz
        memory = 128 # MB
        network {
          mbits = 100
        }
      }
    }
  }
}
job "my_job" {
  datacenters = ["us-west-1", "us-east-1"]
  type = "service"
}

group "web" {
  count = 5
}

  task "frontend" {
    driver = "docker"
    config { image = "hashicorp/web-frontend" }
    resources {
      cpu = 500 # MHz
      memory = 128 # MB
      network { mbits = 100 }
    }
  }
Single Binary - Client/Server Deployment Topology

CLIENT DC1 -> RPC -> SERVER FOLLOWER

CLIENT DC2 -> RPC -> SERVER LEADER

CLIENT DC3 -> RPC -> SERVER FOLLOWER

REPLICATION <-> FORWARDING

HashiCorp
Multi-region Deployment Topology
## Multiple Scheduler Code Paths, Extensible Drivers

<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>
Public Users

CITADEL

SeatGeek

circleci

SAP Ariba

deluxe

PANDORA®
In Orchestrators we trust
We trust orchestrators to:

- Reliably run our applications
- Recover from unexpected failures
- Observability of the apps we run on them
- Resource Usage information
- And so much more
#1 responsibility of orchestrators

Reliability
Reliability Continuum

Application

Meaningful healthchecks
Performance
Well tested code
..

Orchestrator

Handle Node Failures
Handle Control Plane Failures
Handle Application Failures
Gracefully drain traffic
Gracefully migrate workloads
..
Design challenges in building orchestrators

Large blast radius when things go wrong

Applications are opaque to the orchestrator

Design tradeoffs are not obvious
Agenda

Discuss failure modes

What tradeoffs did we make?

Share lessons learned
Failure Modes

Control Plane

Node

Application

Human
Failure Modes

Control Plane

Node

Application

Human
Control Plane Responsibilities

- Reconcile user intent with cluster state
- Schedule new jobs
- Detect node changes
- Application Failures
- Scale up and Scale Down
- Much more
CAP Theorem

Partition Tolerance

Consistency

Availability
CAP Theorem - CP or AP

Partition Tolerance

Consistency

Availability
Control Plane Failures

CP System

Built on top of Raft

Handle (N-1) / 2 failures
Control Plane Design Choices

Why CP?

- Need a single source of truth if you need to reconcile state

Where is state stored?

- **On disk** - replicated logs
- **In memory** - radix tree
Control Plane Design Choices

What happens if the entire control plane is down?

- No updates, No new jobs
- Existing workloads continue to run
Control Plane Performance

Optimistic concurrency (inspired from Omega paper)

Omega: flexible, scalable schedulers for large compute clusters

Malte Schwarzkopf† *  Andy Konwinski‡ *  Michael Abd-El-Malek§  John Wilkes§
†University of Cambridge Computer Laboratory  ‡University of California, Berkeley  §Google, Inc.
†ms705@cl.cam.ac.uk  ‡andyk@berkeley.edu  §mabdelmalek,johnwilkes}@google.com
Optimistic Concurrency
Optimistic Concurrency - Conflict

Plan: Need 500 MB on Node N1 for Job A

Plan: Need 500 MB on Node N1 for Job B
Server

Server

Server

Work Queue

Leader

Plan: Need 500 mbps on Node N1 for Job A

Accept Plan

Plan: Need 500 mbps on Node N1 for Job B

Reject Plan

nack

Allocation

Allocation

Allocation

Optimistic Concurrency - Conflict
Optimistic Concurrency Performance

N servers with M workers

Best case behavior - M*N parallelism

Worst case behavior - Some percentage of plans get rejected, goes back into the work queue
Lessons

Theoretical foundation is critical. You should know whether the system you are building is CP or AP

Consider optimistic concurrency as a pattern
Failure Modes

- Control Plane
- Node
- Application
- Human
Node Responsibilities

- Run user workload
- Reconcile Services with Consul
- Handle retrieving/renewing Vault tokens
- Heartbeat with Server
- Much more
Node Heartbeats

Each server receives heartbeats from a subset of clients.
Heartbeat response contains list of healthy servers.
Node Failure

Server detects missed heartbeats and migrates the workload
Node Failures

Migration takes the type of workload into account

- Completed batch tasks are not migrated.
- Failed tasks that were part of a deployment are not migrated
Node Heartbeat tradeoffs

Configurable parameters to account for network delays

• Min Heartbeat TTL
• Max Heartbeats/Second
• Heartbeat Grace Period
Node Heartbeats - Server unreachable

Some clients can’t reach their heartbeat server
Node Heartbeats - Server unreachable

Send heartbeat to all healthy servers in region

Failover to another server
Failure mode - All Servers unreachable

User workloads continues to run

Initiate discovery loop
Lessons

Consider bidirectional health checks

Graceful degradation

Configurable liveness check behavior
Failure Modes

- Control Plane
- Node
- Application
- Human
How developers see their application
How Nomad sees it

Resources:
1024 MB RAM
10GB disk
100mbps
100 iops

Port:
11360
What causes application failure?

- Missing upstream dependency
- Noisy neighbour
- Code bugs
- Can’t handle incoming traffic

...
Handling Application Failures With Low Information

Likelihood of unrecoverable error

# Restart
Handling failures without enough information

Local restarts are cheap, try them first

If local restarts fail, then reschedule
Local Restarts

```yaml
task "my-api" {
  driver = "docker"
  config {
    image = "my-api:0.2"
  }

  restart {
    attempts = 2
    delay    = "15s"
    interval = "5m"
    mode     = "fail"
  }
}
```
task "my-api" {
    driver = "docker"
    config {
        image = "my-api:0.2"
    }
}

reschedule {
    interval = "30m"
    delay = "30s"
    delay_function = "exponential"
    max_delay = "1h"
    unlimited = true
}
## Reschedule Delay Functions

<table>
<thead>
<tr>
<th>Constant</th>
<th>Exponential</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>30s</td>
<td>30s</td>
<td>30s</td>
</tr>
<tr>
<td>30s</td>
<td>1m</td>
<td>30s</td>
</tr>
<tr>
<td>30s</td>
<td>2m</td>
<td>1m</td>
</tr>
<tr>
<td>30s</td>
<td>4m</td>
<td>1m30s</td>
</tr>
<tr>
<td>30s</td>
<td>8m</td>
<td>2m30s</td>
</tr>
<tr>
<td>30s</td>
<td>16m</td>
<td>4m</td>
</tr>
<tr>
<td>30s</td>
<td>32m</td>
<td>6m30s</td>
</tr>
<tr>
<td>30s</td>
<td>1h4m</td>
<td>10m30s</td>
</tr>
</tbody>
</table>
Reschedule

Automatic Anti-Affinity to the same node

Max delay so that retries are not unbounded

If application stays up successfully, delay is reset to original for next failure
Lesson

When information is insufficient, try cheaper methods first (local restart) vs expensive (reschedule)
Failure Modes

Control Plane

Node

Application

Human
Human Errors

Orchestrator still expected to deal with:

Misconfiguration

Code bugs
$ nomad plan example.nomad

- Job: "example1"

- Task Group: "cache1" (2 create, 1 create/destroy update)
  - Count: "1" => "3" (forces create)

- Task: "redis1" (forces create/destroy update)
  - Config {
    - image: "redis:4.0" => "redis:3.2"
    port_map[0][db]: "6379"
  }
Update Stanza - Rolling Deployments

```yaml
update {
    max_parallel = 2
    min_healthy_time = "30s"
    healthy_deadline = "2m"
}
```
Update Stanza - Canary

```yaml
update {
  max_parallel = 1
  canary = 1
  min_healthy_time = "30s"
  healthy_deadline = "2m"
}
```
count = 3

update {
    max_parallel = 3
    canary = 3
    min_healthy_time = "30s"
    healthy_deadline = "2m"
}

Update Stanza - Blue Green
Update Stanza - Auto Revert

```yaml
update {
  max_parallel = 3
  auto_revert = true
  min_healthy_time = "30s"
  healthy_deadline = "2m"
}
```
Nomad Deployments

- Give enough time to drain traffic
- Gradually roll out new code
- Canaries
- Auto revert on failures
Failure Modes

- Control Plane
- Node
- Application
- Human
Reliability Continuum

Meaningful healthchecks
Performance
Well tested code

Handle Node Failures
Handle Control Plane Failures
Gracefully drain traffic
Gracefully migrate workloads
**Performance monitoring**
Autoscale
Auto revert

Application → Orchestrator
Parting Thoughts..

Embrace failures
Parting Thoughts..

Theoretical foundations are important - decide if you want to be AP or CP. If you don’t know the answer to this, it's a red flag.
Parting Thoughts..

Prefer cheaper methods when there’s insufficient information
Parting Thoughts..

Too much magic in the orchestrator can mask real problems in the application
Thank you

https://github.com/hashicorp/nomad
nomadproject.io
@preethaa