Building a distributed fabric

Neil Soman
EMC^2
Yay! Cloud!

- Systems need to scale out to do useful things
- Self service
- Always available
- Isolate users from bare metal & OS
- Users have dedicated (& root) access to compute
- Wide area, but (generally) infrastructure has a single owner
- Commodity, off-the-shelf systems
  - Storage may not be
- Security
- Provide applications with a base layer of services
  - IaaS, PaaS, etc.
- Simple to program against
Fabric

- Distributed software infrastructure on top of Linux hosts
- Provides API & semantics to deploy & manage applications
- Applications: Object, low latency block storage, big data, user apps
- EMC’s converged infrastructure
  - Storage and compute on same physical hardware
  - Dense appliance SKUs with replaceable high capacity disks (typically 60 x 6TB per node)
  - Customer hardware: DIY
  - Disk monitoring and failure detection
  - Containerized
Building a fabric

• What services do you need to provide?
• What is your topology like?
  – Converged?
• Management layer & user applications
• What is the scale you expect to handle?
  – Build to the scale you expect
• What should the API look like?
  – Do not over-generalize APIs/models
• Fault tolerant
  – Minimum: No single point of failure
  – Better: Handle multiple failures over time.
Provisioning

• API & Contract
• EC2 Model
  – VM instance with root access
  – Instance types
  – We give you the tools, you manage it
  – Not a lot of control over placement
  – Autoscaling
  – Not very performant for disk bound
    • Can pay for more CPU, IOPs, etc.
Provisioning

• Managed
  – “Fabric” manages application lifecycle

• Hard vs soft constraints
  – Node/rack tagging
  – Run the object storage application on “yellow” nodes
  – Need at least xx CPU
  – SAS drives preferred

• More control over hardware
  – Applications are isolated but not necessarily adversarial
  – Direct access to disks

• Contract
  – Fabric will keep application up
  – Provide services to coordinate tasks, perform rolling upgrade, etc.
Tiered architecture
Node management

• Compute
  – Virtual Machines
  – Containers

• Storage
  – Raw disk enclosures
  – Direct attached arrays
  – Filesystems

• Networking
  – Programmable network fabric
  – VLANs
  – Iptables
Cluster management

• Nodes aggregated into “clusters”
  – Nodes may not be homogenous
• Responsible for allocation, failure detection, recovery, notification & migration
• Expansion
• Must itself be fault tolerant
  – Multiple cluster manager instances
• Credentials/certificate authority/distribution
• Application lifecycle services
Lifecycle

- Goal State: What is the “desired state” of an application?
  - e.g. versioned image, CPU, disks, ports, affinity
- Provisioning can take a long time
  - Format disks, create filesystems, open ports, create VLANs, download binaries
  - Drive towards goal state until delta is zero
- Respond to failures, requests for additional capacity (i.e. scale up/down)
- Changes performed by staging an update and then flipping a bit
  - Only the latter needs to be atomic
Geo/Wide Area

• Credential Service
• Licensing
• Federation & membership
  – Dynamic, clusters can join/leave
• Secure communication between clusters
Upgrades

• Upgrade with downtime
  – Easier and okay for management software
  – Not if the application is in the data path

• Rolling upgrades
  – Service must continue to function & accept requests
  – Run multiple versions in cluster and gradually switch over
  – Rollback
  – Versioning

• Transferring binaries or images
  – Layered filesystems: updates are diffs
  – Should not be a single point of failure
Polling vs events

• Polling
  – Periodically ask for state information

• On timeout, take some action
  – Declare component as “unhealthy”
  – Initiate failover

• Events: push out state changes in “real time”
  – More responsive, don’t need to wait for next poll period

• In practice, need a combination
  – Events may be lost

• “Eventing” can be made reliable
  – Seq numbers, persistence, compact encoding, etc.
  – Accessible over REST (give me events starting at seq # X)
Failures & redundancy

• Fault domains
  – Set of components that share a single point of failure
  – Physical and software

• Distribute components based on fault domains
  – Fault isolation: If a failure occurs, system is still available
  – Performance

• System should return to “non degraded” state
Fault domain
Handling faults
Handling faults
Cluster Manager is an Application!
Handling faults
Operator friendliness

• Provide guidelines for infrastructure
  – e.g. sane naming for nodes/racks, redundant switches, etc.
• Notifications: Send events out
• Have a consistent command line experience
• Be able to take over node/cluster & enter maintenance mode
• Configuration management
  – Ability to dynamically update node or cluster-wide defaults from a single terminal
  – Accessible over web services
  – If possible, standardize, but not always possible
    • e.g. OS commands might be different
• Your system will break
  – Yay, but you designed a good upgrade experience!
Software design principles

• Immutability
  – Do not pass around entities with “nullable” fields
• Get your primitives right
  – Threading, profiling, logging, etc.
• Know where your state is
  – State changes should be explicit
  – Avoid side effects
  – Should be able to reason about state changes
• Snapshots, events & replay
• Abstraction
  – Pluggable: can run on laptop or a dense cluster
• Don’t assume you will get notified
  – Component software defects can cause a cascade
  – Push events along with polling
• Audit logs
  – Required for sanity & compliance