Amazon S3: Architecting for Resiliency in the Face of Failures

Jason McHugh

Can your Service Survive?

CAN YOUR SERVICE SURVIVE?

Can your Service Survive?

- Datacenter loss of connectivity
- Flood
- Tornado
- Complete destruction of a datacenter containing thousands of machines

KEY TAKEAWAYS

- Dealing with large scale failures takes a qualitatively different approach
- Set of design principles here will help
- AWS, like any mature software organization, has learned a lot of lessons about being resilient in the face of failures

OUTLINE

- AWS
- Amazon Simple Storage Service (S3)
- Scoping the failure scenarios
- Why failures happen
- Failure detection and propagation
- Architectural decisions to mitigate the impact of failures
- Examples of failures

ONE SLIDE INTRODUCTION TO AWS

- Amazon Elastic Compute Cloud (EC2)
- Amazon Elastic block storage service (EBS)
- Amazon Virtual Private Cloud (VPC)
- Amazon Simple storage service (S3)
- Amazon Simple queue service (SQS)
- Amazon SimpleDB
- Amazon Cloudfront CDN
- Amazon Elastic Map-Reduce (EMR)
- Amazon Relational Database Service (RDS)

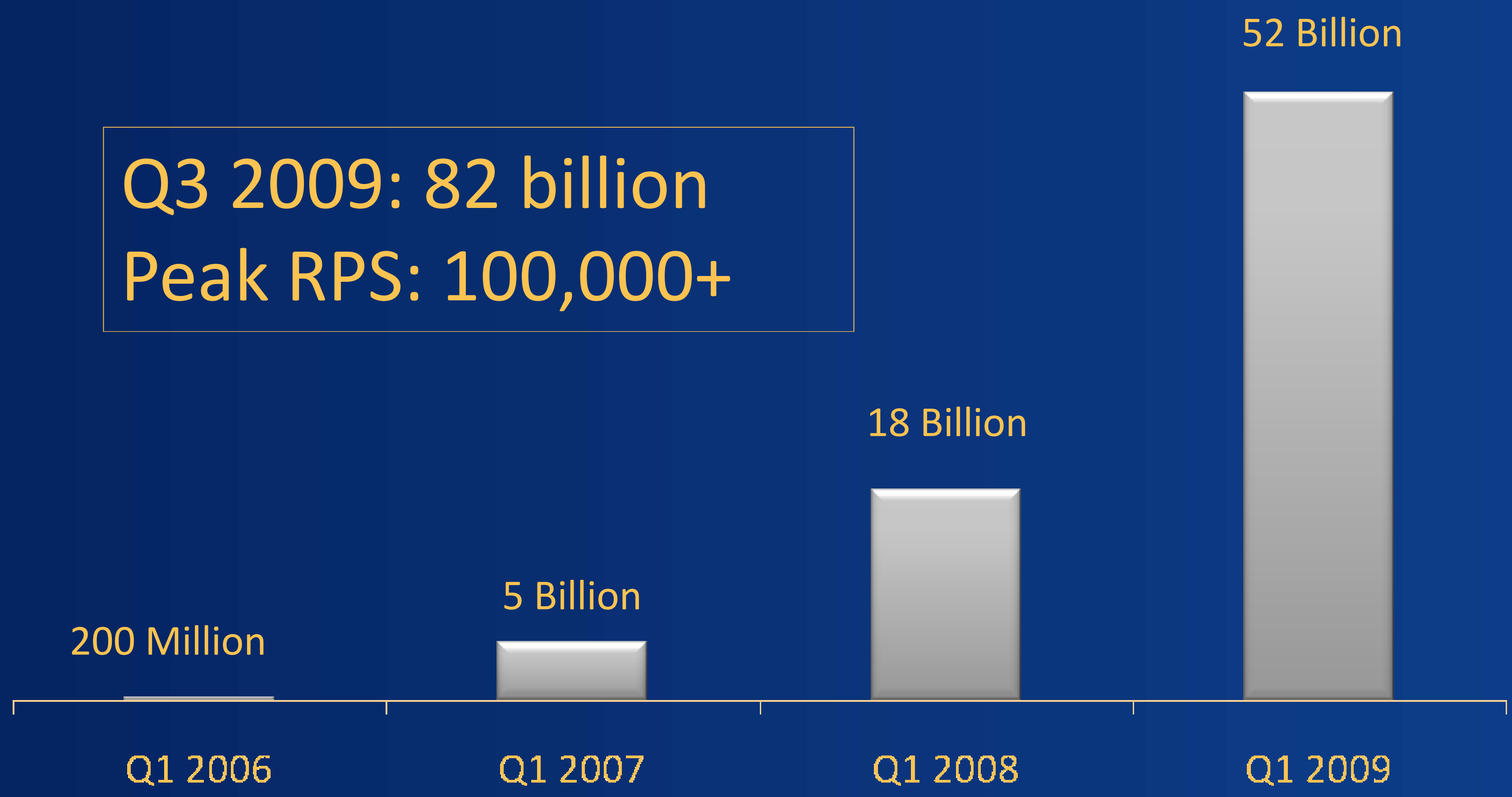
AMAZON S3

- Simple storage service
- Launched: March 14, 2006 at 1:59am
- Simple key/value storage system
- Core tenets: simple, durable, available, easily addressable, eventually consistent
- Large scale import/export available
- Financial guarantee of availability
  - Amazon S3 has to be **above** 99.9% available

Amazon S3 Momentum

Q3 2009: 82 billion

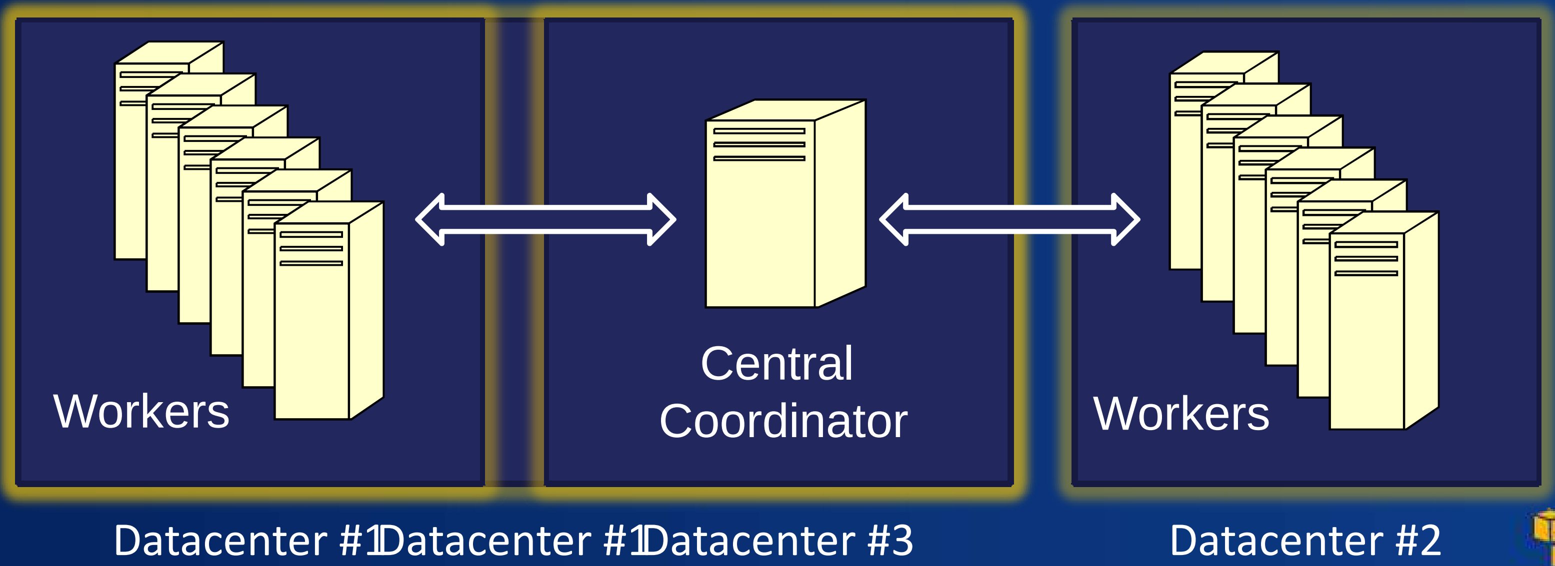
Peak RPS: 100,000+



Total Number of Objects Stored in Amazon S3

Failures

- There are some things that pretty much everyone knows
  - Expect drives to fail
  - Expect network connection to fail (independent of the redundancy in networking)
  - Expect a single machine to go out



FAILURE SCENARIOS

- Corruption of stored and transmitted data
- Losing one machine in fleet
- Losing an entire datacenter
- Losing an entire datacenter and one machine in another datacenter

Why Failures Happen

- Human error
- Acts of nature
- Entropy
- Beyond scale

FAILURE CAUSE: HUMAN ERROR

- Network configuration
  - Pulled cords
  - Forgetting to expose load balancers to external traffic
- DNS black holes
- Software bug
- Failure to use caution while pushing a rack of servers

FAILURE CAUSE: ACTS OF NATURE

- Flooding
  - Standard kind
  - Non-standard kind: Flooding from the roof down
- Heat waves
  - New failure mode: dude that drives the diesel truck
- Lightning
  - It happens
  - Can be disruptive

FAILURE CAUSE: ENTROPY

- Drive failures
  - During an average day many drives will fail in Amazon S3

- Rack switch makes half the hosts in rack unreachable
  - Which half? Depends on the requesting IP.
- Chillers fail forcing the shutdown of some hosts
  - Which hosts? Essentially random from the service owner's perspective.

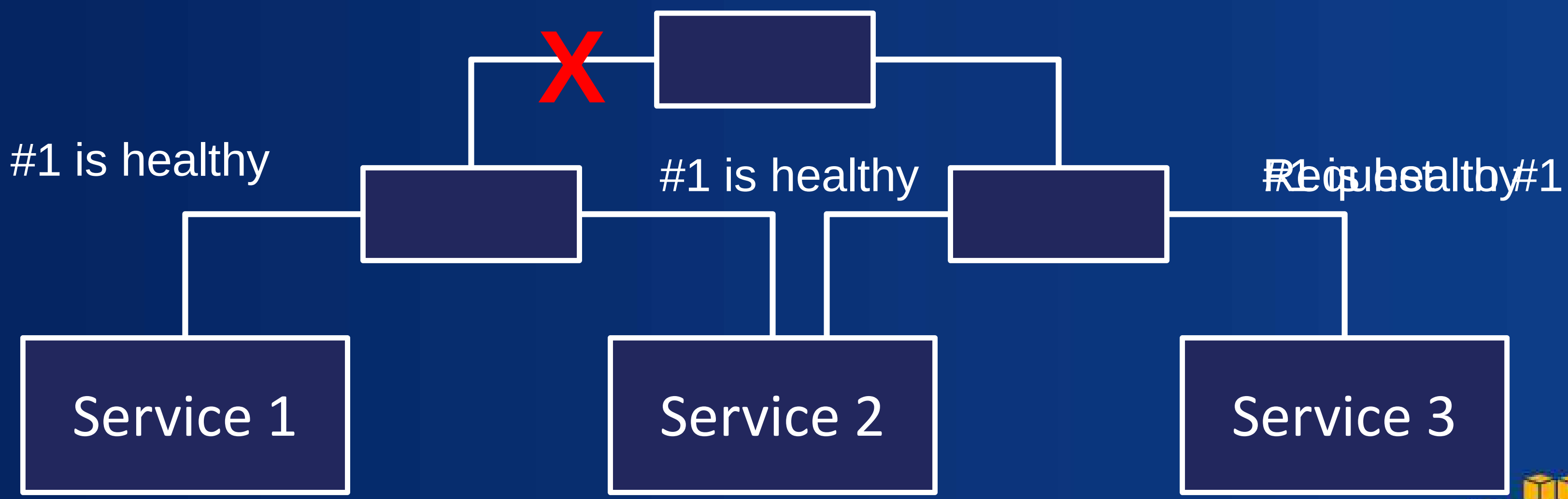
amazon webservices™

FAILURE CAUSE: BEYOND SCALE

- Some dimensions of scale are easy to manage
  - Amount of free space in system
  - "Precise" measurements of when you could run out
  - No ambiguity
  - Acquisition of components by multiple suppliers
- Some dimensions of scale are more difficult
  - Request rate
  - Ultimate manifestation: DDOS attack

Recognizing When Failure Happens

- Timely failure detection
- Propagation of failure must handle or avoid
  - Scaling bottlenecks of their own
  - Centralized failure of failure detection units
  - Asymmetric routes



Gossip Approach for Failure Detection

- Gossip, or epidemic protocols, are useful tools when probabilistic consistency can be used
- Basic idea
  - Applications, components, or *failure units*, heartbeat their existence
  - Machines wake up every time quantum to perform a "round" of gossip
  - Every round machines contact another machine randomly, exchange all "gossip state"
- Robustness of propagation is both a positive and negative

S3's Gossip Approach – The Reality

- No, it really isn't this simple at scale
  - Can't exchange all "gossip state"
    - Different types of data change at different rates
    - Rate of change might require specialized compression techniques
  - Network overlay must be taken into consideration
  - Doesn't handle the bootstrap case
  - Doesn't address the issue of application lifecycle
    - This alone is not simple
    - Not all state transitions in lifecycle should be performed automatically. For some human intervention may be required.

Design Principles

- Prior just sets the stage
- 7 design principles

DESIGN PRINCIPLES – TOLERATE FAILURES

- Service relationships

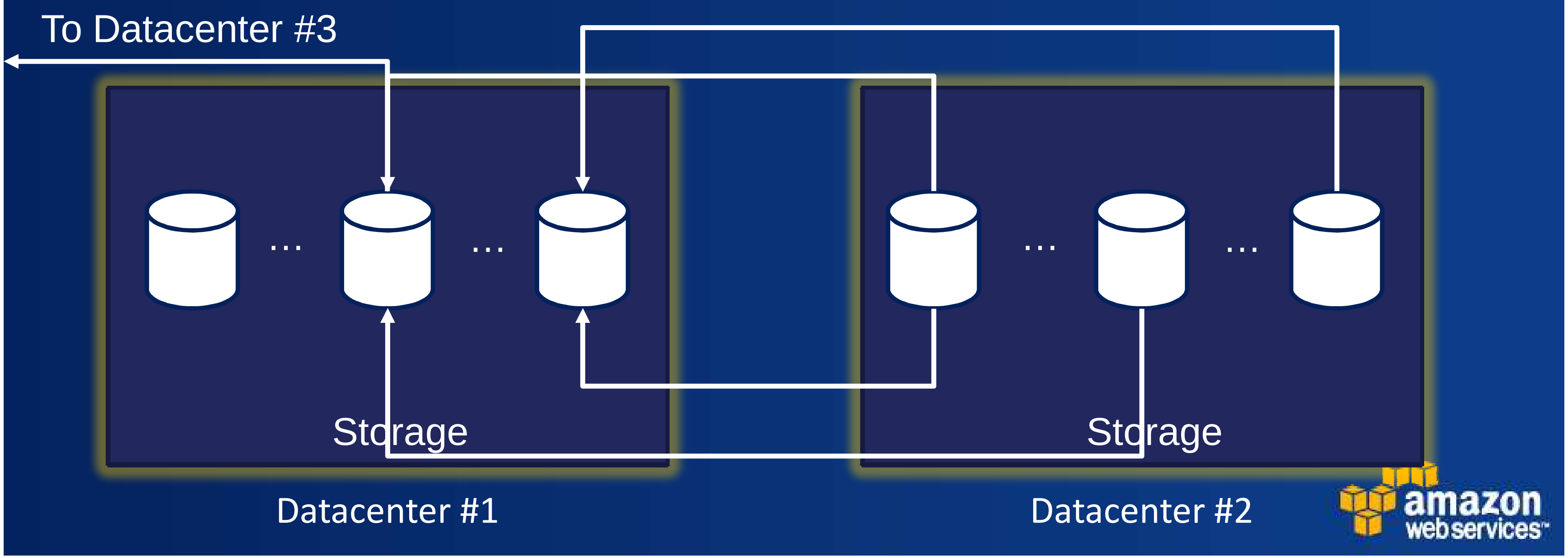
- Decoupling functionality into multiple services has standard set of advantages
  - Scale the two independently
  - Rate of change (verification, deployment, etc)
  - Ownership
  - encapsulation and exposure of proper primitives

Design Principles – Tolerate Failures

- Protect yourself from upstream service dependencies when they haze you
- Protect yourself from downstream service dependencies when they fail

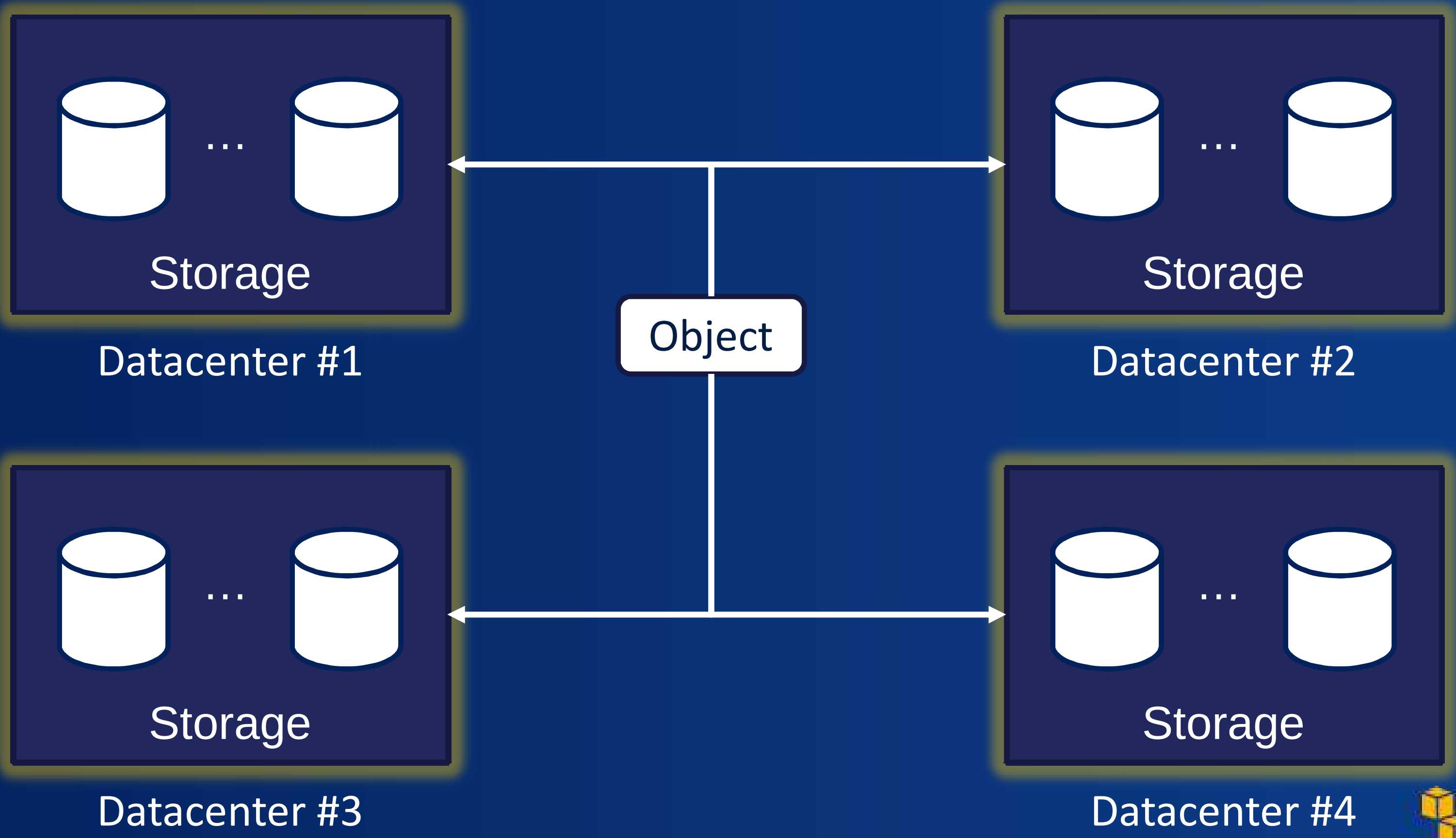
Design Principles – Code for Large Failures

- Some systems you suppress entirely
- Example: replication of entities (data)
  - When a drive fails replication components work quickly
  - When a datacenter fails then replication components do minimal work without operator confirmation



Design Principles – Code for Large Failures

- Some systems must choose different behaviors based on the unit of failure



Design Principle – Data & Message Corruption

- At scale it is a certainty
- Application must do end-to-end checksums
  - Can’t trust TCP checksums
  - Can’t trust drive checksum mechanisms
- End-to-end includes the customer

amazon webservices™

DESIGN PRINCIPLE – CODE FOR ELASTICITY

- The dimensions of elasticity
  - Need infinite elasticity for cloud storage
  - Quick elasticity for recovery from large-scale failures
- Introducing new capacity to a fleet
  - Ideally you can introduce more resources in the system and capabilities increase
  - All load balancing systems (hardware and software)
    - Must become aware of new resources
    - Must not haze
    - How not to do it

amazon webservices™

Design Principle – Monitor, extrapolate, and react

- Modeling
- Alarming
- Reacting
- Feedback loops
- Keeping ahead of failures

Design Principle – Code for Frequent Single Machine Failures

- Most common failure manifestation – a single box
  - Also sometimes exhibited as a larger-scale uncorrelated failure
- For persistent data consider use Quorum
  - Specialization of redundancy
  - If you are maintaining n copies of data
    - Write to w copies and ensure all n are eventually consistent
    - Read from r copies of data and reconcile

amazon webservices™

Design Principle – Code for Frequent Single Machine Failures

- For persistent data use Quorum
  - Advantage: does not require all operations to succeed on all copies
    - Hides underlying failures
    - Hides poor latency from users
  - Disadvantages
    - Increases aggregate load on system for some operations
    - More complex algorithms
    - Anti-entropy is difficult at scale

Design Principle – Code for Frequent Single Machine Failures

- For persistent data use Quorum
  - Optimal quorum set size
    - System strives to maintain the optimal size even in the face of failures
  - All operations have a “set size”
    - If available copies are less than the operation set size then the operation is not available
    - Example operations: read and write
  - Operation set sizes can vary depending on the execution of the operations (driven by user’s access patterns)

DESIGN PRINCIPLE – GAME DAYS

- Network eng and data center technicians turn off a data center
  - Don't tell service owners
  - Accept the risk, it is going to happen anyway
  - Build up to it to start
  - Randomly, once a quarter minimum
  - Standard post-mortems and analysis
- Simple idea – test your failure handling – however it may be difficult to introduce

REAL FAILURE EXAMPLES

- Large outage last year
- Traced down to a single network interface card
- Once found the problem was easily reproduced
- Corruption leaked past TCP checksuming on the single communication channel that did not have application level checksuming

Real Failure Examples

- Network access to Datacenter is lost
- Happens not infrequently
  - Several noteworthy events in the last year
  - Due to transit providers, networking upgrades, etc.
  - None noticed by customers
  - Easily direct customers away from a datacenter
- It helps that we run game-days and irregular maintenance by failing entire datacenters

Real Failure Examples

- Network route asymmetry
  - Learning about machine health via gossip
  - Route taken to learn about health might not be the same taken by communication between two machines
  - Results in split brain
    - I think that machine is unhealthy
    - Everyone else says it is fine, keep trying

Real Failure Examples

- Rack switch makes all or some of hosts unreachable
- Must handle losing hundreds of disks simultaneously
  - Independent of fixing the rack switch and the timeline some action needs to be taken
  - Intersection of a hundreds of sets of objects (say each set is 10 million objects) efficiently taking into account state of the world for other failed components

Design Principles Recap

- Expect and tolerate failures
- Code for large scale failures
- Expect and handle data and message corruption
- Code for elasticity
- Monitor, extrapolate and react
- Code for frequent single machine failures
- Game days

What I Haven't Discussed

- Unit of failures
- Coalescing reporting of failures intelligently
- How to handle a failure
- Recording and trending of failure types
- Tracking and resolving failures
- In general all issues related to maintaining a good ratio of support burden to fleet size

CONCLUSION

- Just scratching the surface
- Set of design principles which can help your system be resilient in the face of failures
- Amazon S3 has maintained aggregate availability far in excess of our stated SLA for the last year
- Amazon AWS is hiring: http://aws.amazon.com/jobs

amazon webservices™

QUESTIONS?