Scalable & Available

Patterns for Success

Derek Collison

@derekcollison

dcollison@vmware.com

derek.collison@gmail.com

Special Thanks

Jonas Bonér

twitter: @jboner
http://jonasboner.com/

http://www.slideshare.net/jboner/scalability-availability-stability-patterns

Background

- **Scalable Apps** maintain performance under load
  - More requests, More users, More data
- **Available Apps** maintain the experience during failures
  - Hardware failures, Network splits/partitioning
- **Simple Designs** tend to scale better

Background

Good Performance is good

Background

Predictable
Performance
is king!

Background

Understand
your ***data!***

Background

Understand
the user
experience!

Background

Measure everything

(can’t fix what you don’t know)

Background

Don’t be a ***failure*** of your own ***success***

Master the **Tradeoffs**

(For your app and your data!)

Master the **Tradeoffs**

Performance
vs **Scalability**

Master the **Tradeoffs**

Latency vs **Throughput**

Master the **Tradeoffs**

Availability vs **Consistency**

Lots of ways to skin a **cat!**

Scalability Patterns

Performance

vs

Scalability

How do I know if I have a performance problem?

If your system is slow for a single request/user

How do I know if I have a scalability problem?

If your system is fast for a single request/user but slow for many users

Latency

VS

Throughput

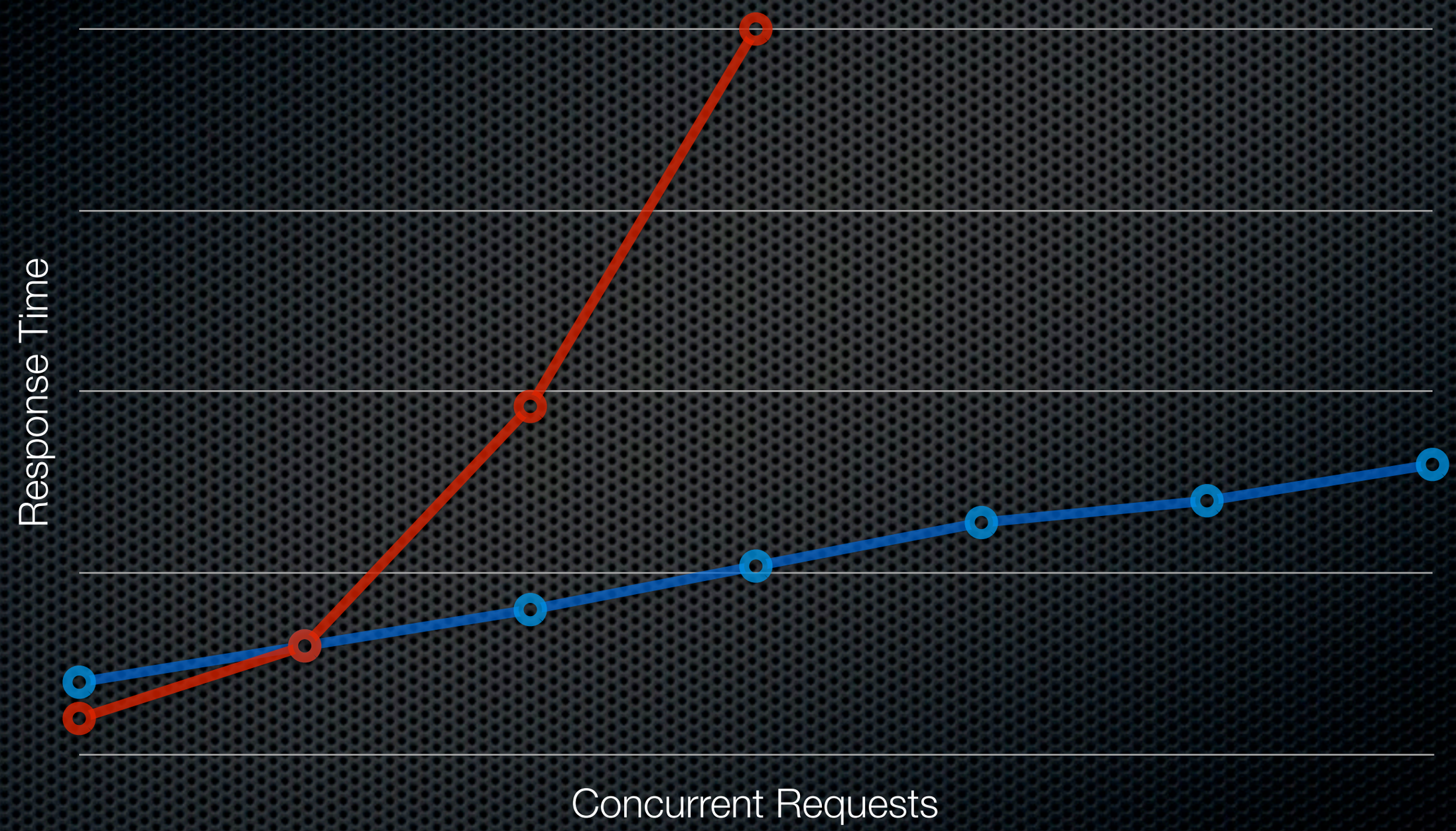
You should strive for

maximal throughput

with

acceptable latency

Performance vs Scalability



Know what to scale!

- CPU or IO Bound?
- Scale up or Scale out?
- Waiting on IO? What? Disk/Net/Other System?
- How many components are used per request?
- Know who and what the slowest will be!

Scalability Patterns

Behavior

Scalability Patterns: Behavior

- ✓ Event-Driven Architectures
- ✓ Load-Balancing
- ✓ Parallel Computing

Event-Driven Architecture

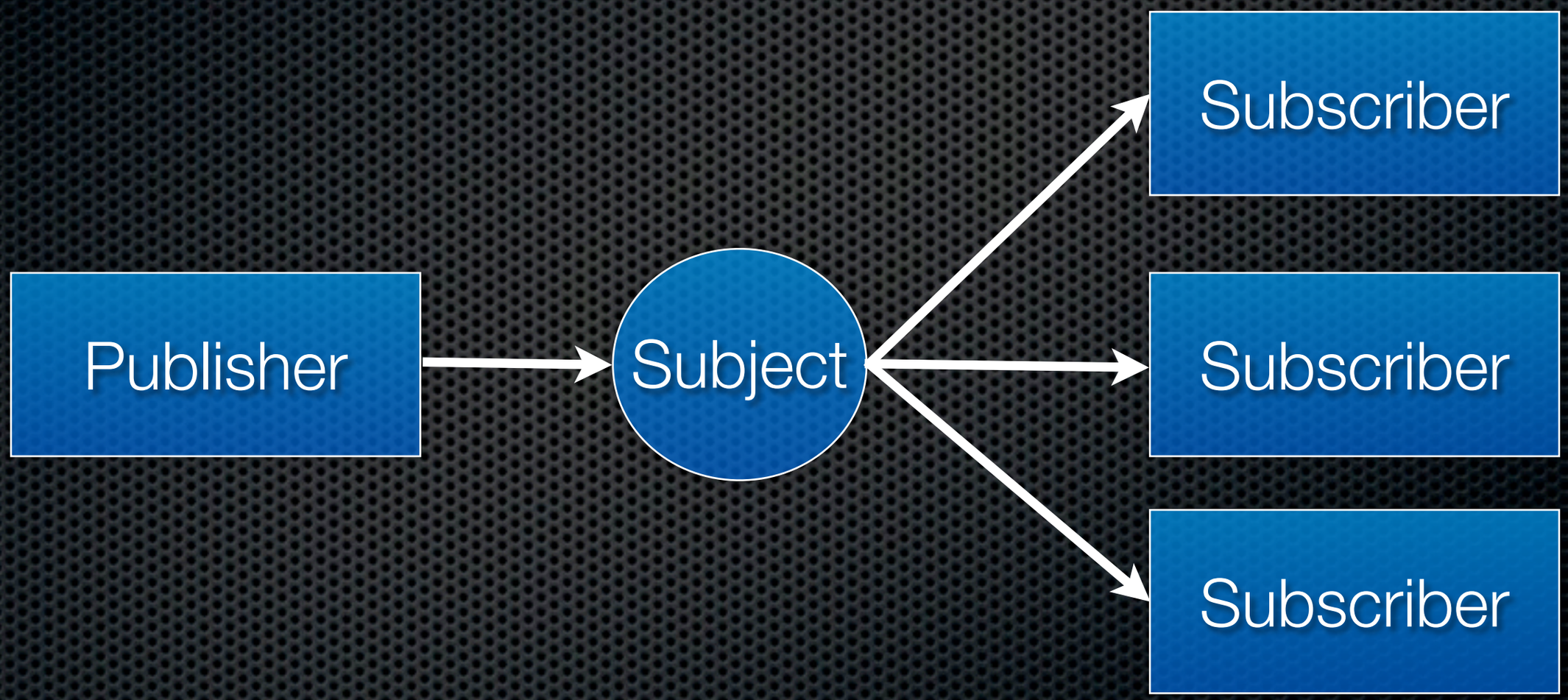
- ✓ Events
- ✓ Messaging
- ✓ Asynchronous
- ✓ Non-blocking

Messaging

- ✓ Publish-Subscribe
- ✓ Queuing
- ✓ Request-Reply
- ✓ Store and Forward

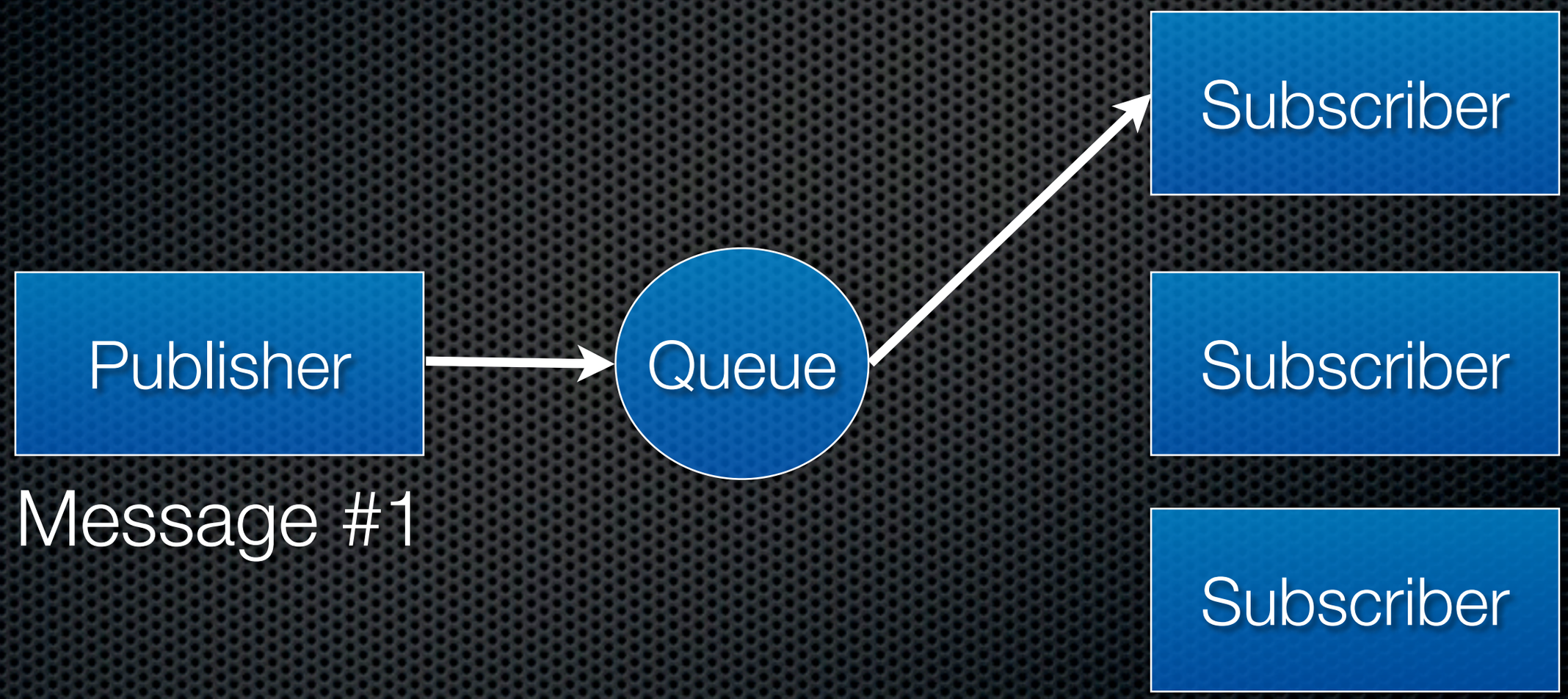
Messaging - Publish Subscribe

1 : N



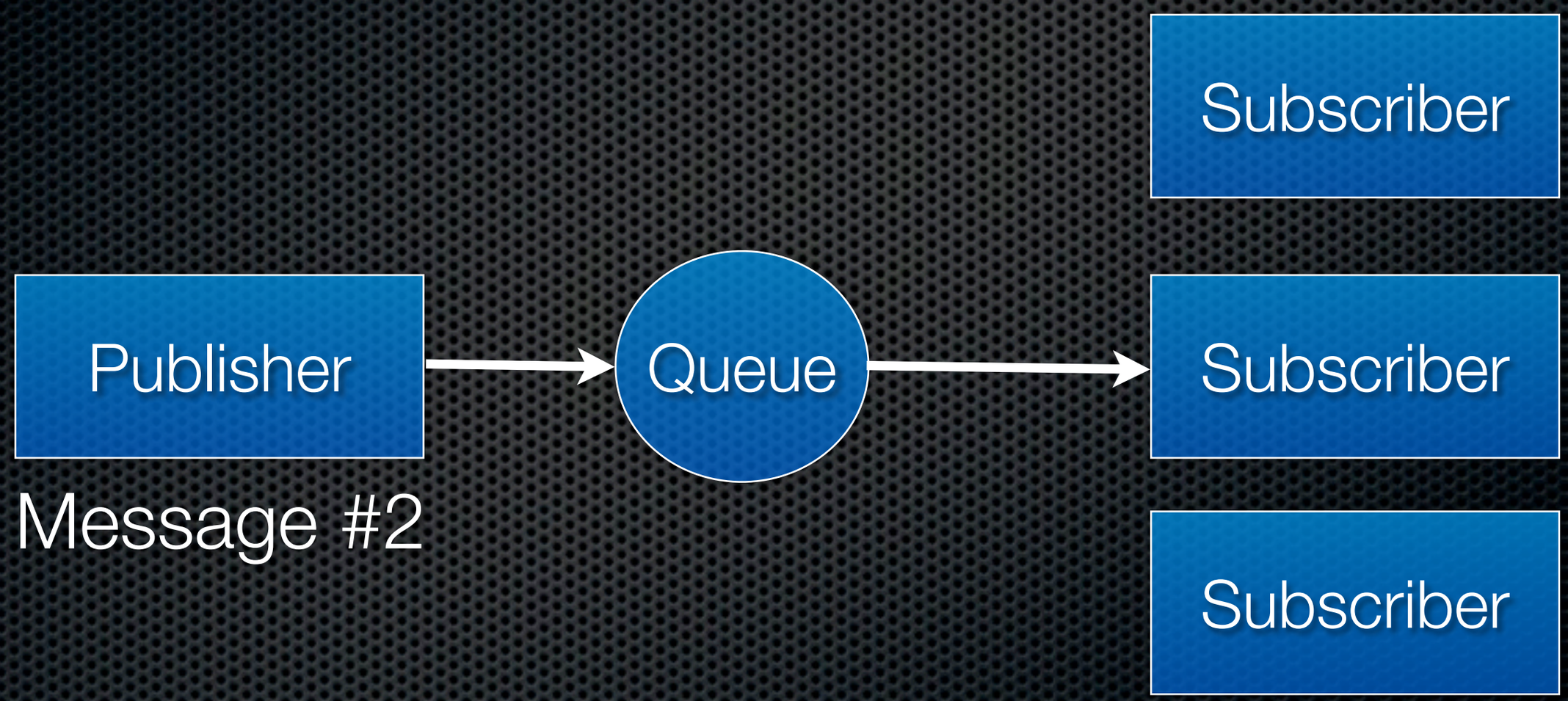
Messaging - Queuing

1 : 1



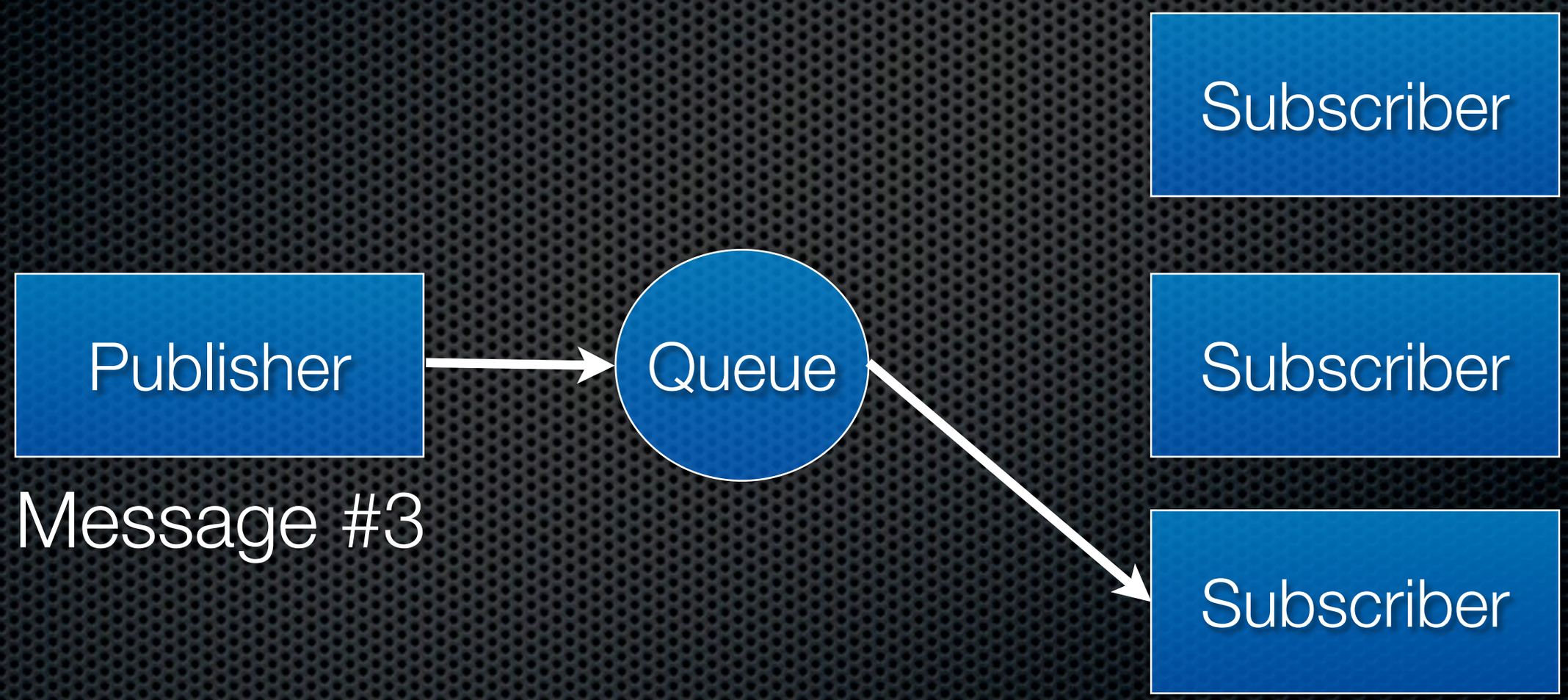
Messaging - Queuing

1 : 1



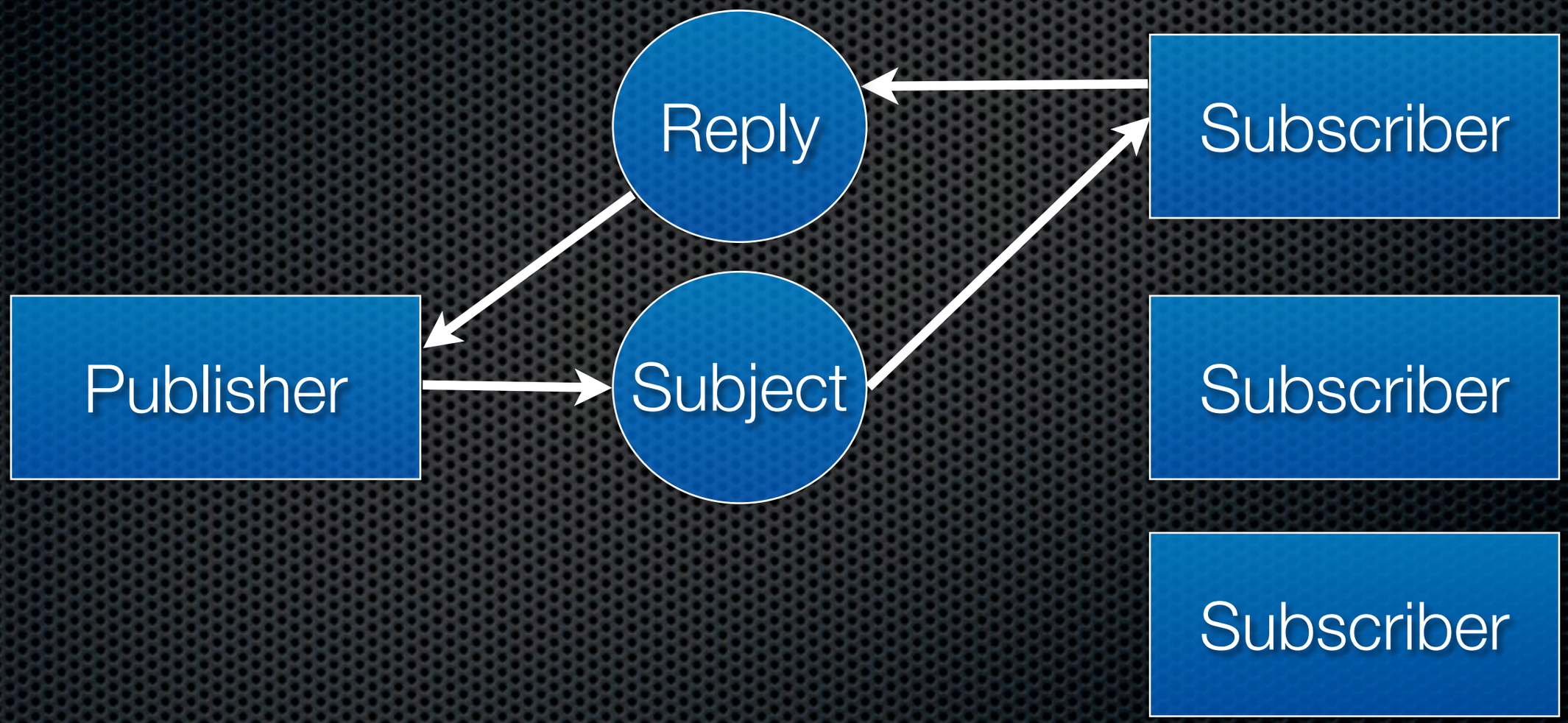
Messaging - Queuing

1 : 1



Messaging - Request Reply

1 : 1



Messaging Patterns

- ✓ Addressing, discovery
- ✓ Command and control
- ✓ Load-balancing
- ✓ N-way scalability

Messaging

- ✓ Standards
  - ✓ AMQP (wire)
  - ✓ JMS (api)
- ✓ Products
  - ✓ RabbitMQ
  - ✓ ZeroMQ
  - ✓ ActiveMQ
  - ✓ TIBCO
  - ✓ MQSeries

Asynchronous and Non-Blocking

- ✓ Don’t wait, go doing something else
- ✓ Never block
- ✓ All callbacks all the time can get messy!
- ✓ Good language/framework support
  - ✓ functional closures
  - ✓ co-routines

Load Balancing

- ✓ Multiple endpoints to perform work
- ✓ Can be semantically aware
- ✓ Chainable: DNS, hardware, software
- ✓ Endpoints can be Hardware, VM, process, thread, co-routine, fiber, etc.

Load Balancing
Selection

- ✓ Random
- ✓ Round Robin
- ✓ Weighted
- ✓ Dynamically “aware”
  - ✓ Least connections
  - ✓ Least loaded

Load Balancing
Technologies

- ✓ DNS Round Robin
- ✓ Anycast
- ✓ Reverse Proxies
- ✓ Clustering
- ✓ Hardware Load Balancers

Load Balancing
Reverse Proxies

- ✓ Nginx
- ✓ HAProxy
- ✓ Apache (mod_proxy)
- ✓ Squid

Parallel Computing

- ✓ Divide and Conquer
- ✓ Worker queues
- ✓ Map Reduce
- ✓ UE = Unit of Execution
  - ✓ VM, process, thread, co-routine, fiber, callback

Parallel Computing
Worker Queues

- ✓ Good for offloading tasks
- ✓ Need bounded time check in master
- ✓ Async result processing
- ✓ Fork/Join pattern

Parallel Computing
MapReduce

- ✓ Used internally at Google
- ✓ Variation of Fork and Join
- ✓ Distributed
- ✓ Originally used for logs processing

Parallel Computing
MapReduce

- ✓ Google's MapReduce
- ✓ Hadoop
- ✓ Amazon's Elastic MapReduce
- ✓ RIAK uses it internally for queries

Scalability Patterns

State

Scalability Patterns: State

Harder than scaling

behavior

Scalability Patterns: State

- ✓ Master Record
- ✓ Replication
- ✓ Sharding
- ✓ Caching
- ✓ NoSQL
- ✓ Concurrency

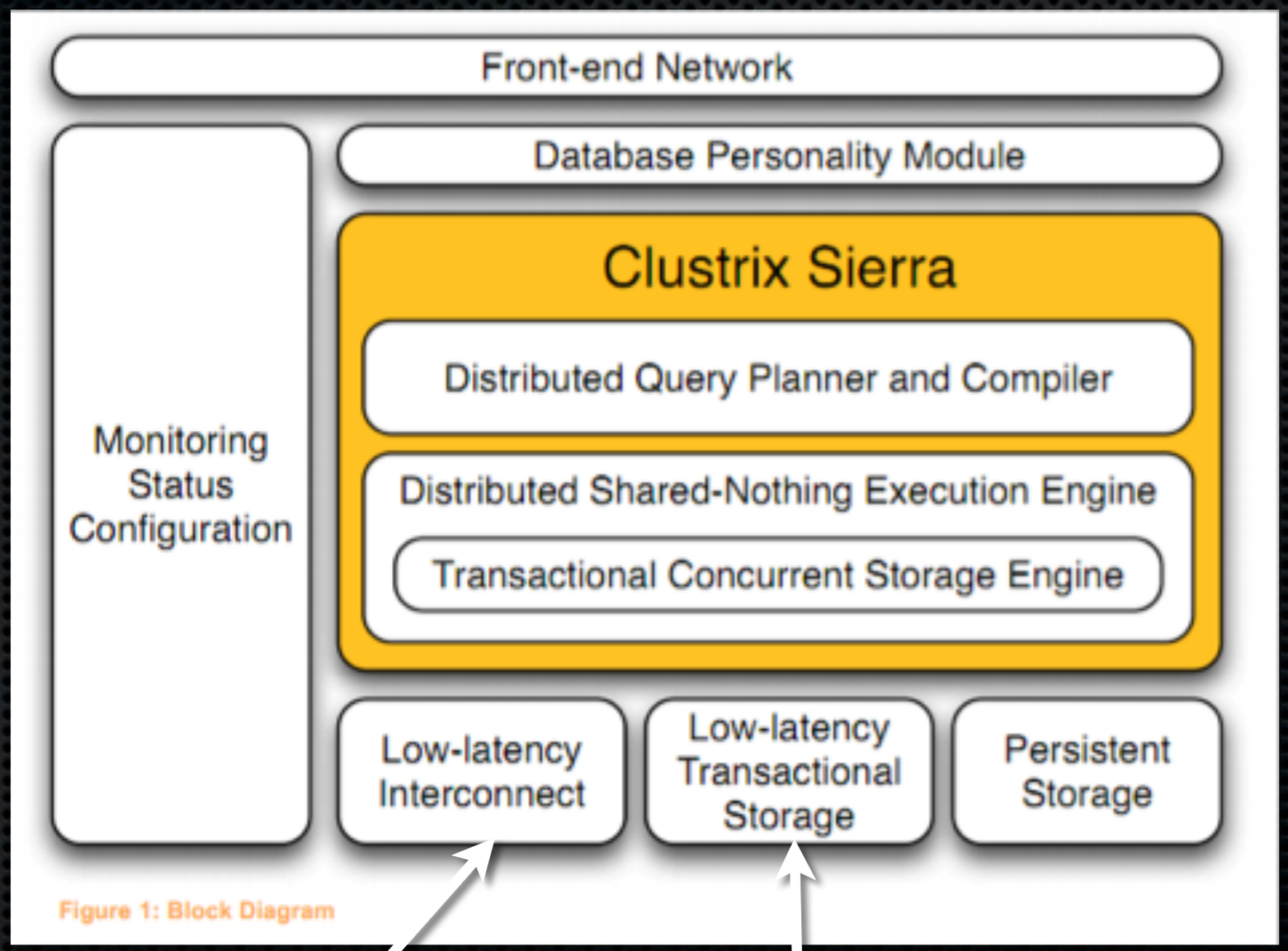
Master Record

- ✓ Normally Relational Databases (RDBMS)
- ✓ NoSQL Databases emerging
- ✓ Can’t lose this data
- ✓ Scaling can be a challenge

Master Record: Scaling

- ✓ Traditonally Scale Up
- ✓ **Technology** will help here
  - ✓ SSD (50k-100k IOPs)
  - ✓ More memory/cores per box
  - ✓ Faster network connectivity
  - ✓ Clustering Appliances

Clustering Appliances



64 bit

Infiniband

SSD

Master Record: Scaling

- ✓ Scaling Reads vs Writes?
- ✓ Scaling Reads with Slaves
  - ✓ Synchronous (Speed of Light)
  - ✓ Asynchronous

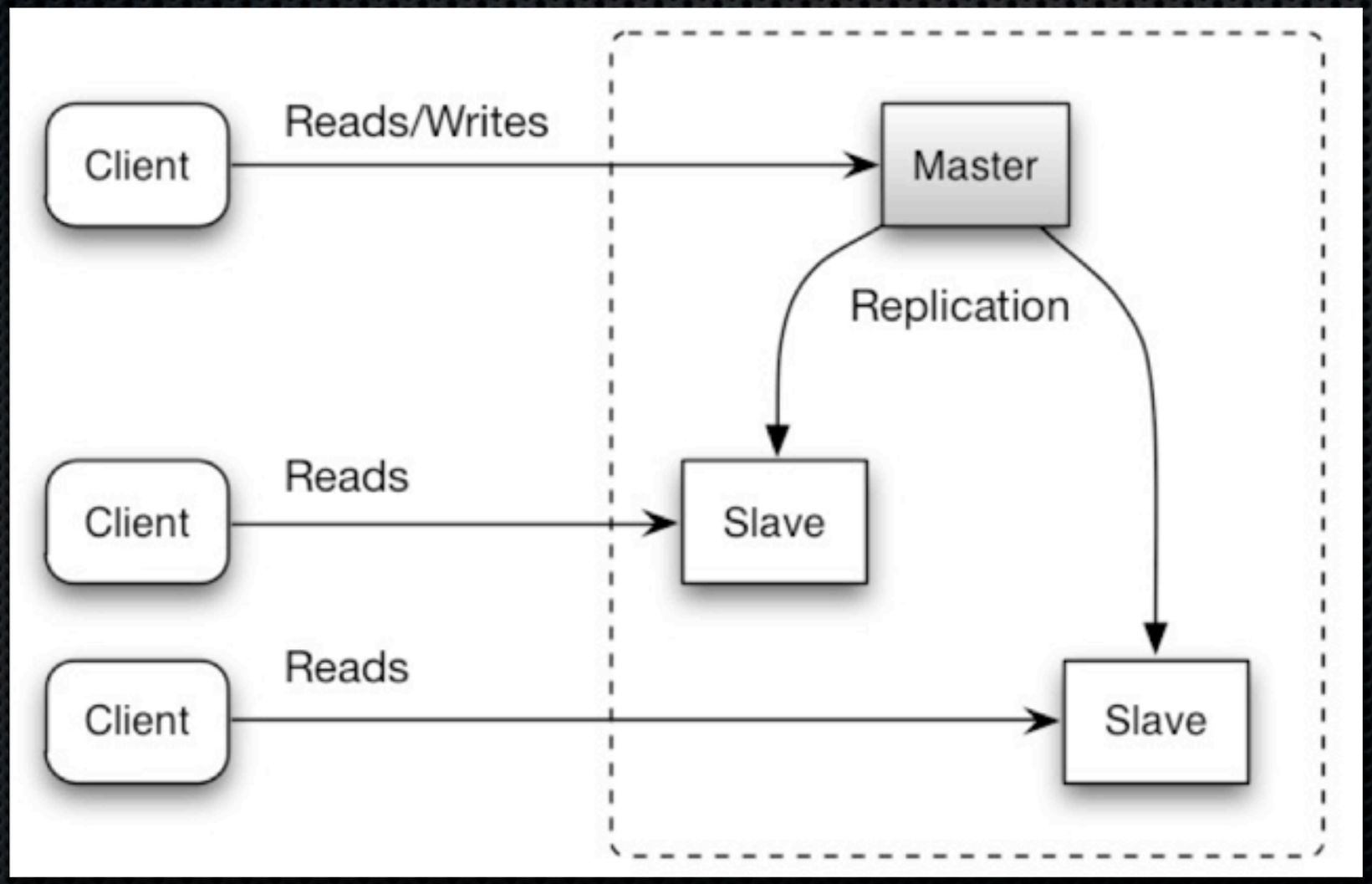
Master Record: Scaling

How do we scale OUT?

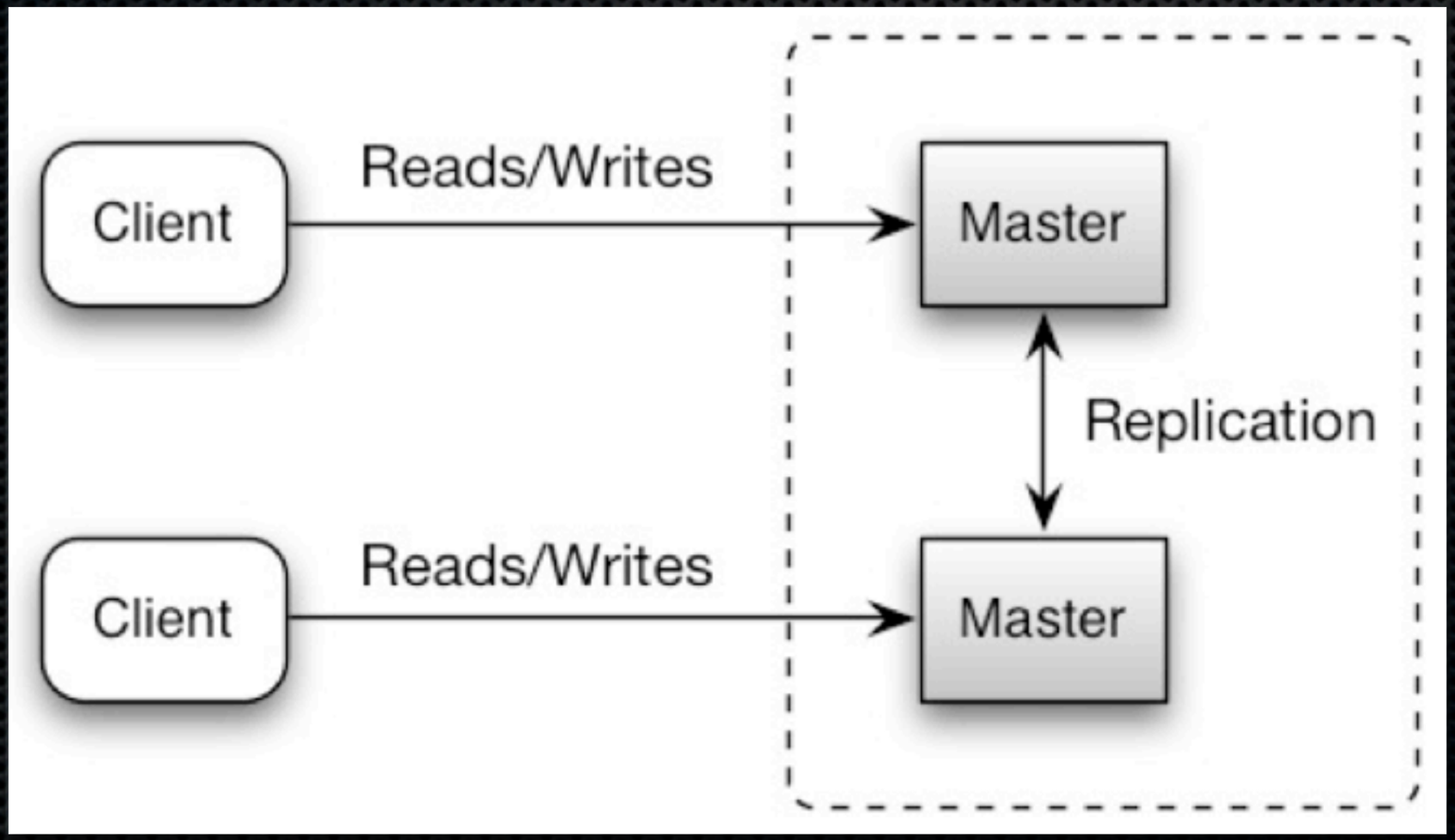
Master Record: Replication

- ✓ Synchronous vs Asynchronous
- ✓ Master / Slave Replication
- ✓ Master / Master Replication
- ✓ Tree Replication
- ✓ Buddy Replication

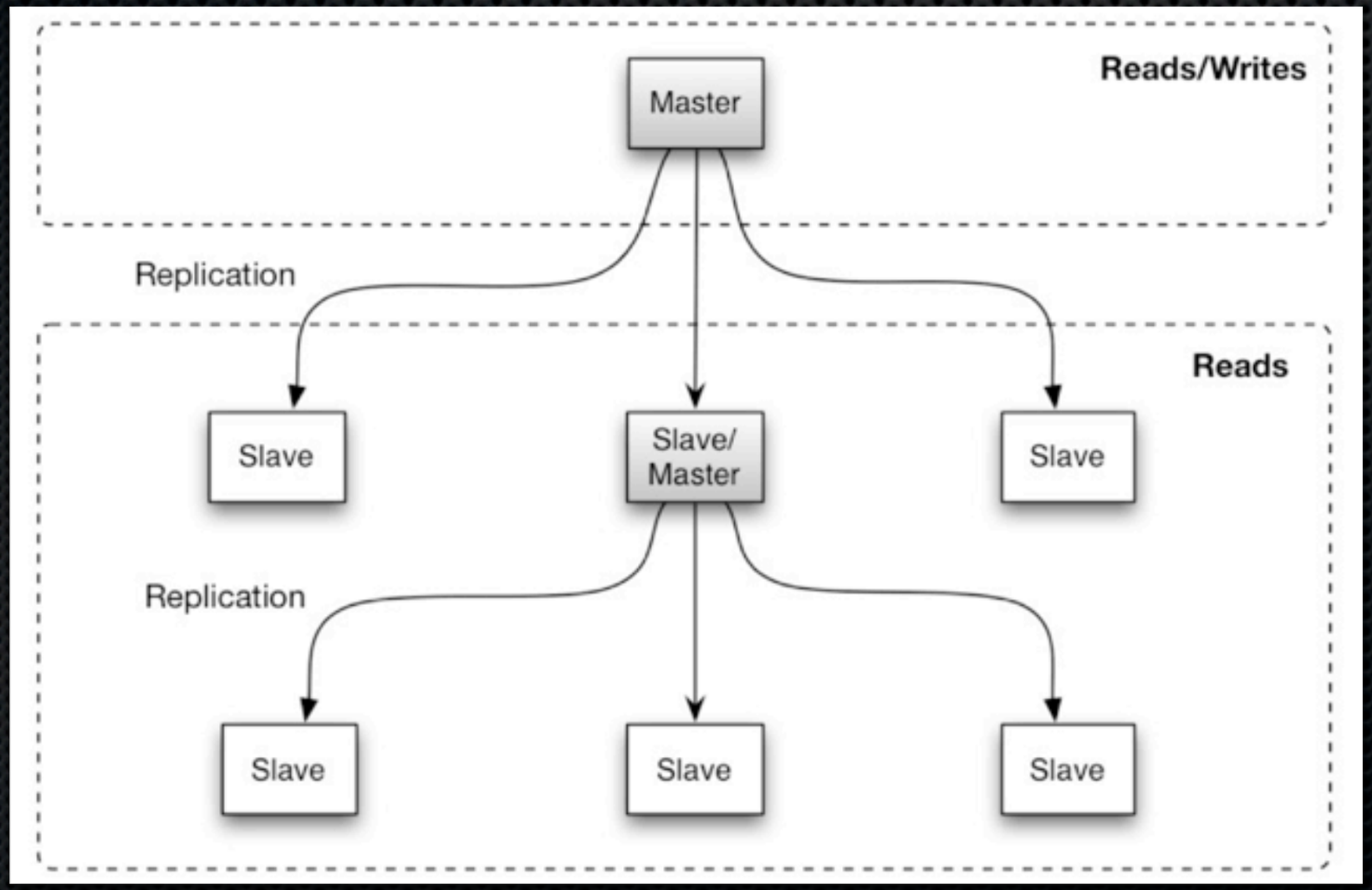
Replication: Master / Slave



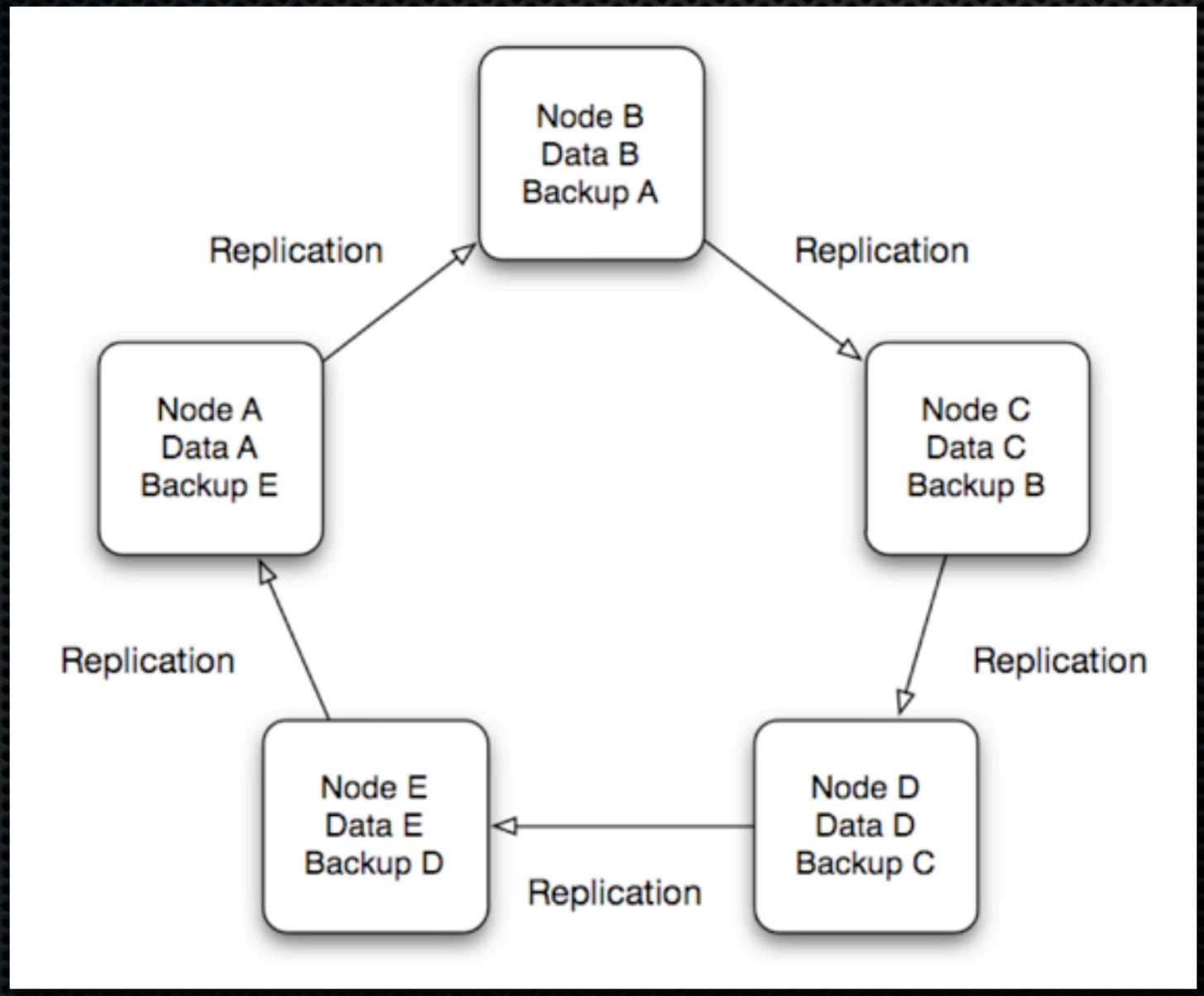
Replication: Master / Master



Replication: Tree



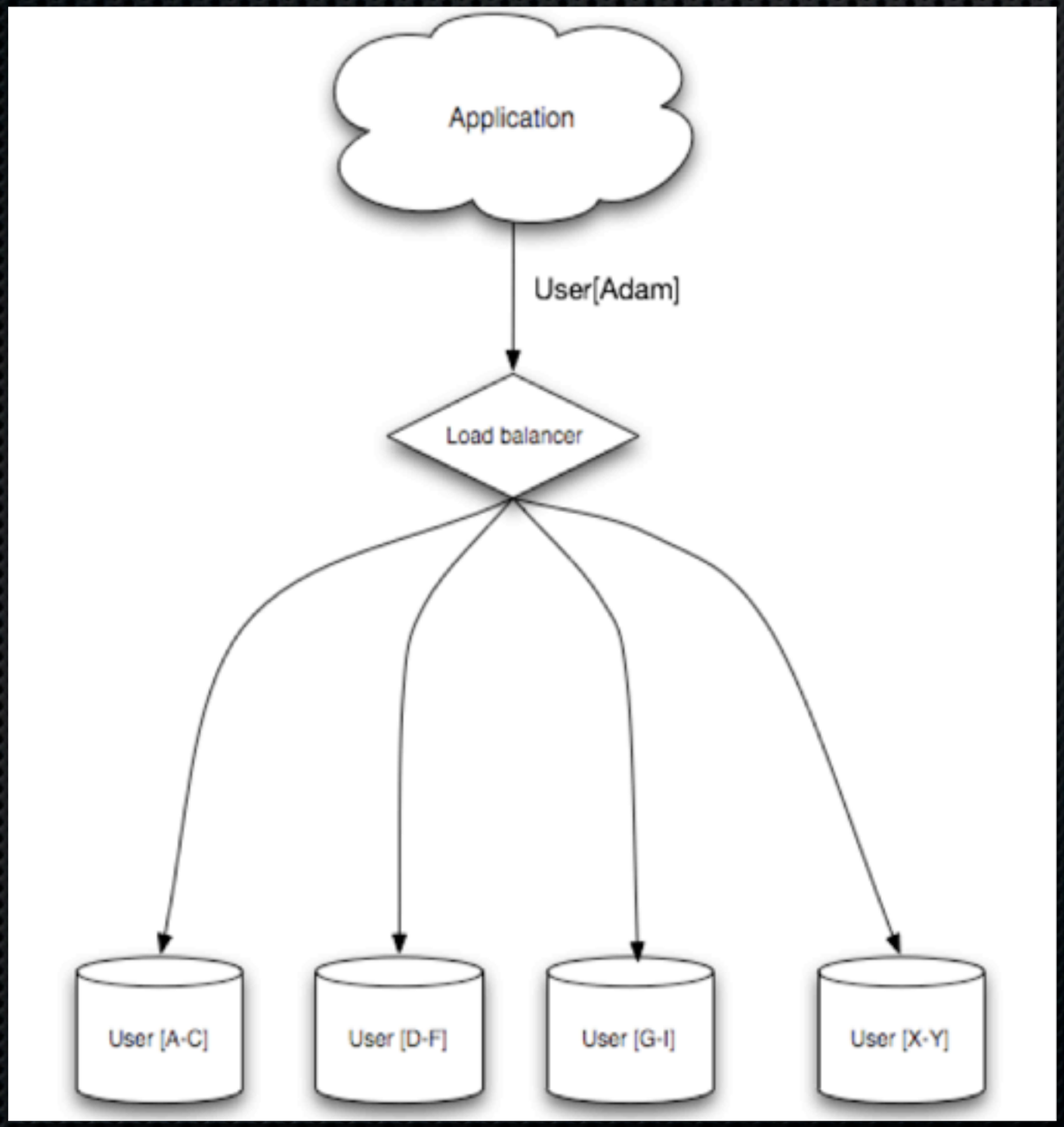
Replication: Buddy



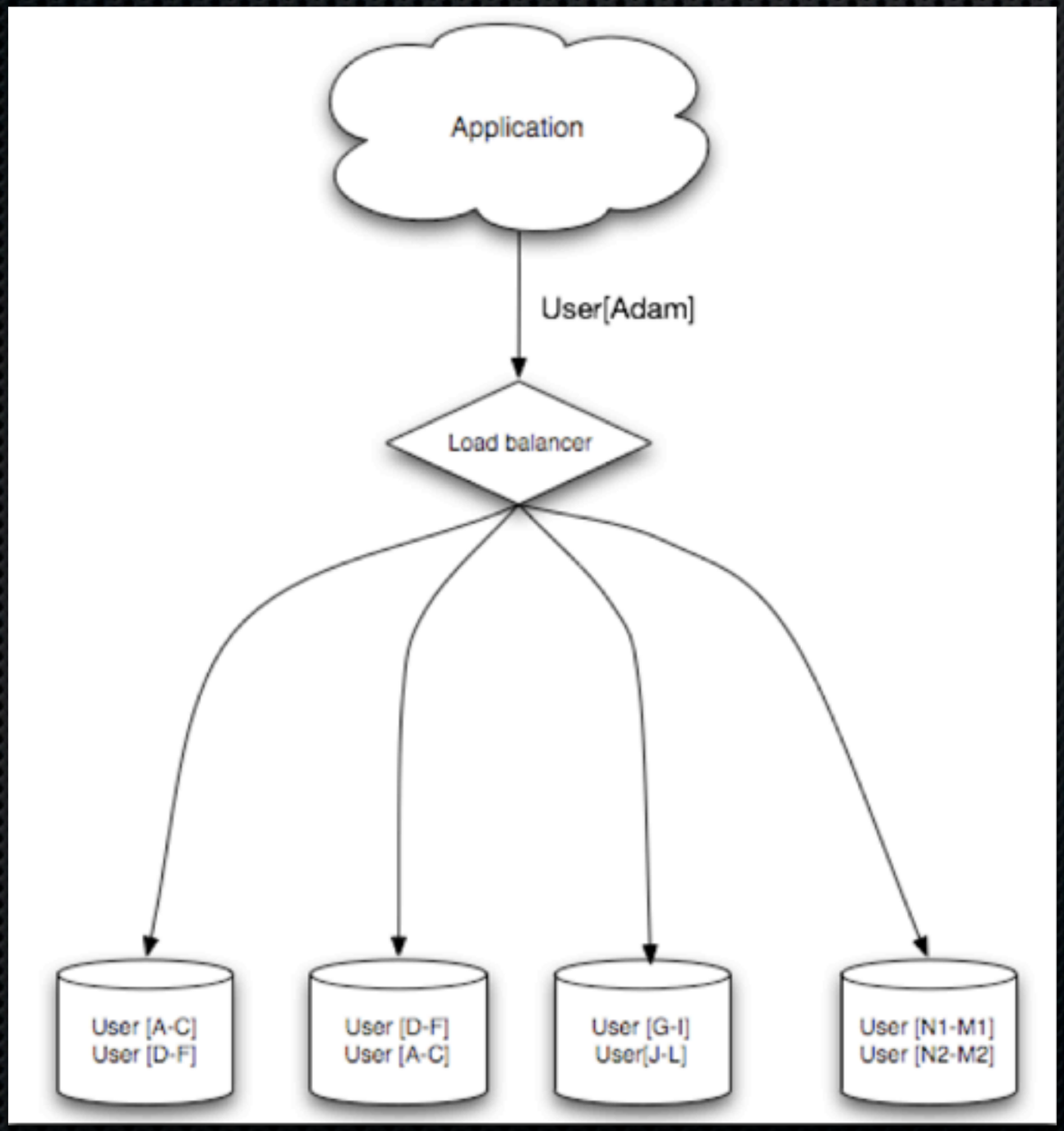
Sharding

- ✓ Partitioning state
- ✓ Requests need to know where to go
  - ✓ Distributed Hash
  - ✓ Load Balancer
  - ✓ Messaging

Sharding: Paritioning



Sharding: Replication



Sharding: Over-provision

- ✓ Use N partitions
- ✓ Use Y replicas
- ✓ Use message based requests
- ✓ First back wins
- ✓ Therefore user wins (Google Search)

Master Record: RDBMS

Do we really need an **RDBMS**?

Master Record: RDBMS

Don’t underestimate **RDBMS**

or

the ability of a single machine

Master Record: RDBMS

What about **alternatives**?

NoSQL

- ✓ Key-Value
- ✓ Column Databases
- ✓ Document Databases
- ✓ Graph Databases
- ✓ Datastructure Databases

NoSQL

- ✓ Key-Value: (Memcache, Redis, Riak)
- ✓ Column Databases: (Cassandra, Vertica)
- ✓ Document Databases: (MongoDB, CouchDB)
- ✓ Graph Databases: (Neo4J, AllegroGraph)
- ✓ Datastructure Databases: (Redis, Hazelcast)

NoSQL in the **wild**

- ✓ Google: Bigtable, Colossus
- ✓ Twitter: Redis
- ✓ Amazon: Dynamo, SimpleDB
- ✓ Yahoo: HBase (Hadoop)
- ✓ Facebook: Cassandra, HBase

Caching

- ✓ Cache early and often
- ✓ Usually biggest bang for the buck
- ✓ Referential Transparency
- ✓ Polyglot APIs coming
- ✓ NoSQL stores
- ✓ Cache invalidation is still **hard!**

Caching

- ✓HTTP (HTML, JS, CSS, Images, Media)
- ✓Key/Value Data
- ✓Semantic Data structures

HTTP Caching

- ✓ Varnish
- ✓ Squid
- ✓ Pound
- ✓ Nginx
- ✓ Rack-cache

HTTP Caching
CDN

- ✓ Akamai
- ✓ Limelight
- ✓ Level3
- ✓ Digital Fountain (Qualcomm)
- ✓ aiCache

HTTP Caching
CDN

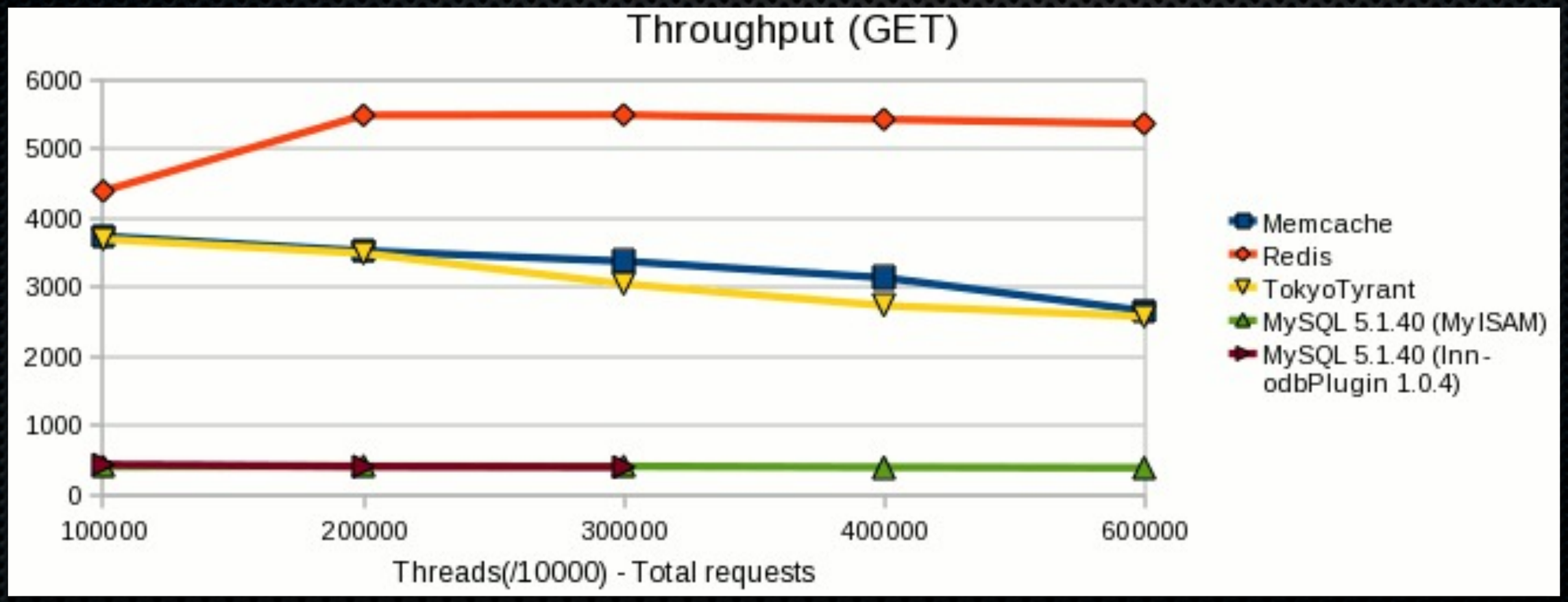
HTTP Caching

- ✓ Lives in browsers, proxies, CDNs, apps
- ✓ Hard to control, so do it right!
- ✓ Master page controls other resources
  - ✓ master page not cached (at least too far)
  - ✓ read-only resources
  - ✓ change link in master page

Key/Value Caching

- ✓ Memcache
- ✓ Redis
- ✓ Riak
- ✓ Voldemort

Data Structure Caching



Data Structure Caching

- ✓ Standalone
- ✓ Augment RDBMS
- ✓ In Memory or on Disk

Data Structure Caching

- ✓ Data Types
  - ✓ Strings, Hashes, Lists, Sets, Sorted Sets
- ✓ Atomic Operations
  - ✓ Push, pop, ranges, set operations (intersect, union)

Caching Patterns

- ✓ Write Through
- ✓ Write Behind
- ✓ Replicated
- ✓ P2P

Cache Invalidation

- ✓ TTL (Time to Live)
- ✓ Bounded FIFO or LIFO
- ✓ Explicit cache invalidation
- ✓ Explicit non-use of read-only resource
  - ✓ Harder problem the more master items used

Scalability Key Points

- ✓The problem is not where you think ;)
- ✓Autoscaling is a myth
- ✓Can’t fix what you can’t measure
- ✓Scaling master record writes is hard
- ✓Scaling reads is more tractable
- ✓What is the opex cost of your choices?

Availability Patterns

What do you do when things go bad?

Availability Patterns

Available vs Consistent

Availability Patterns

Available

We have been here

before, right?

Yes, we have been here before!?

Scalability Patterns

Behavior

Scalability Patterns: Behavior

- ✓ Event-Driven Architectures
- ✓ **Load-Balancing**
- ✓ Parallel Computing

Scalability Patterns

State

Scalability Patterns: State

- ✓ Master Record
- ✓ **Replication**
- ✓ **Sharding**
- ✓ **Caching**
- ✓ NoSQL
- ✓ Concurrency

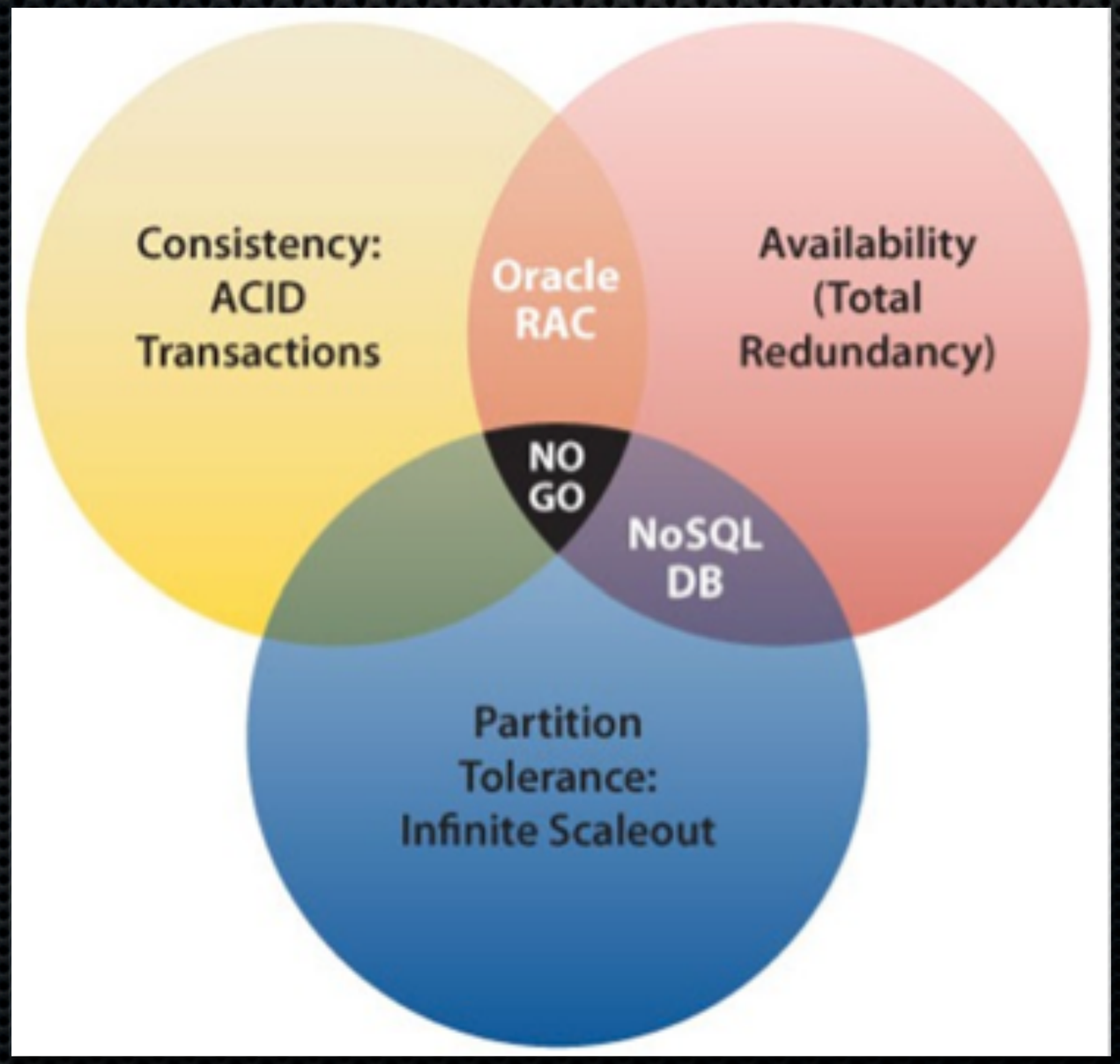
But let’s talk more about your data

Availability Patterns

Available vs Consistent

Brewer's CAP Theorem

Brewer's CAP Theorem



You can only pick 2

- **C**onsistency
- **A**vailability
- **P**artition Tolerance

Centralized Systems

- ✓ If the system is **centralized**
  - ✓ no **P** (network partitions)
- ✓ So you get **both**:
  - ✓ **A**vailability
  - ✓ **C**onsistency

Distributed Systems

- ✓ If the system is **distributed**
  - ✓ you will have **P!** (network partitions)
- ✓ So you get **pick one**:
  - ✓ **A**vailability
  - ✓ **C**onsistency

CAP in reality

- ✓There is only once choice to make:
- ✓When there is a **network partition**, which do you **sacrifice**?
  - ✓**A**vailability
  - ✓**C**onsistency

BASE

What is BASE?

BASE

- **B**asically
- **A**vailable
- **S**oft State
- **E**ventually Consistent

Eventually Consistent

- ✓ Great **tradeoff** for the right kind of **data**
- ✓ **Can't** be used everywhere
- ✓ Works in more places than you think
- ✓ Solved speed of light problem

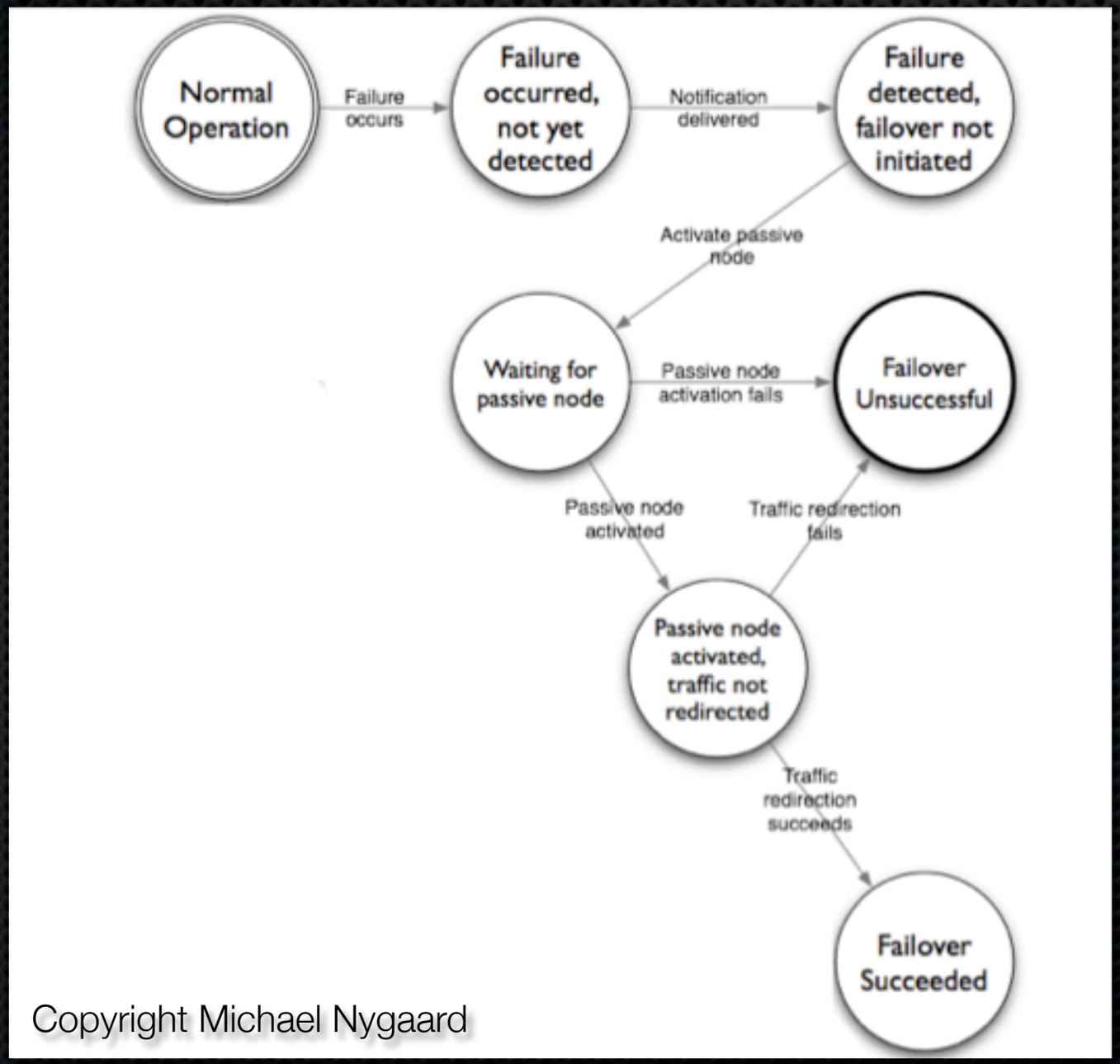
Availability Patterns

Failover

Availability Patterns: Failover

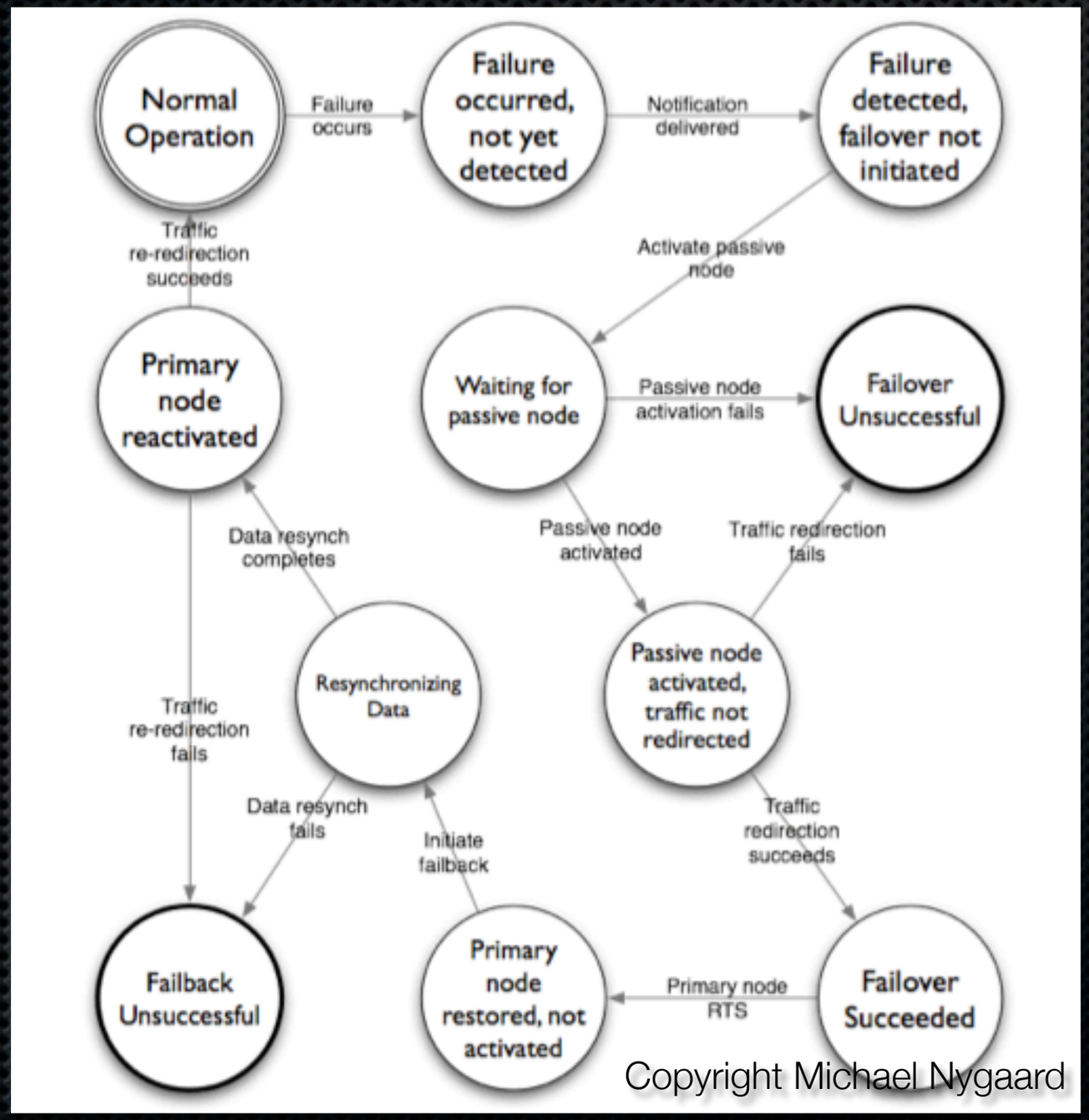
- ✓ Failover is complex
- ✓ Switch time is critical
- ✓ Failback is equally as complex

Availability Patterns: Failover



Copyright Michael Nygaard

Availability Patterns: Failback



Copyright Michael Nygaard

Availability Patterns: Replication

- ✓ Synchronous vs Asynchronous
- ✓ Master / Slave Replication
- ✓ Master / Master Replication

Availability Patterns: Redirection

- ✓ DNS
- ✓ Load Balancers
- ✓ Secondary Sites

Availability Key Points

- ✓ Always have a dial tone
- ✓ Syntactically correct is good
- ✓ Semantically correct is better
- ✓ Be transparent

Beating the dead **horse**

Understand your **data!**

Understand your **user!**

Understand the experience!

Master the **Tradeoffs**

(For your app and your data!)

Thank You

Thank You

Questions?

Derek Collison

@derekcollison

dcollison@vmware.com

derek.collison@gmail.com