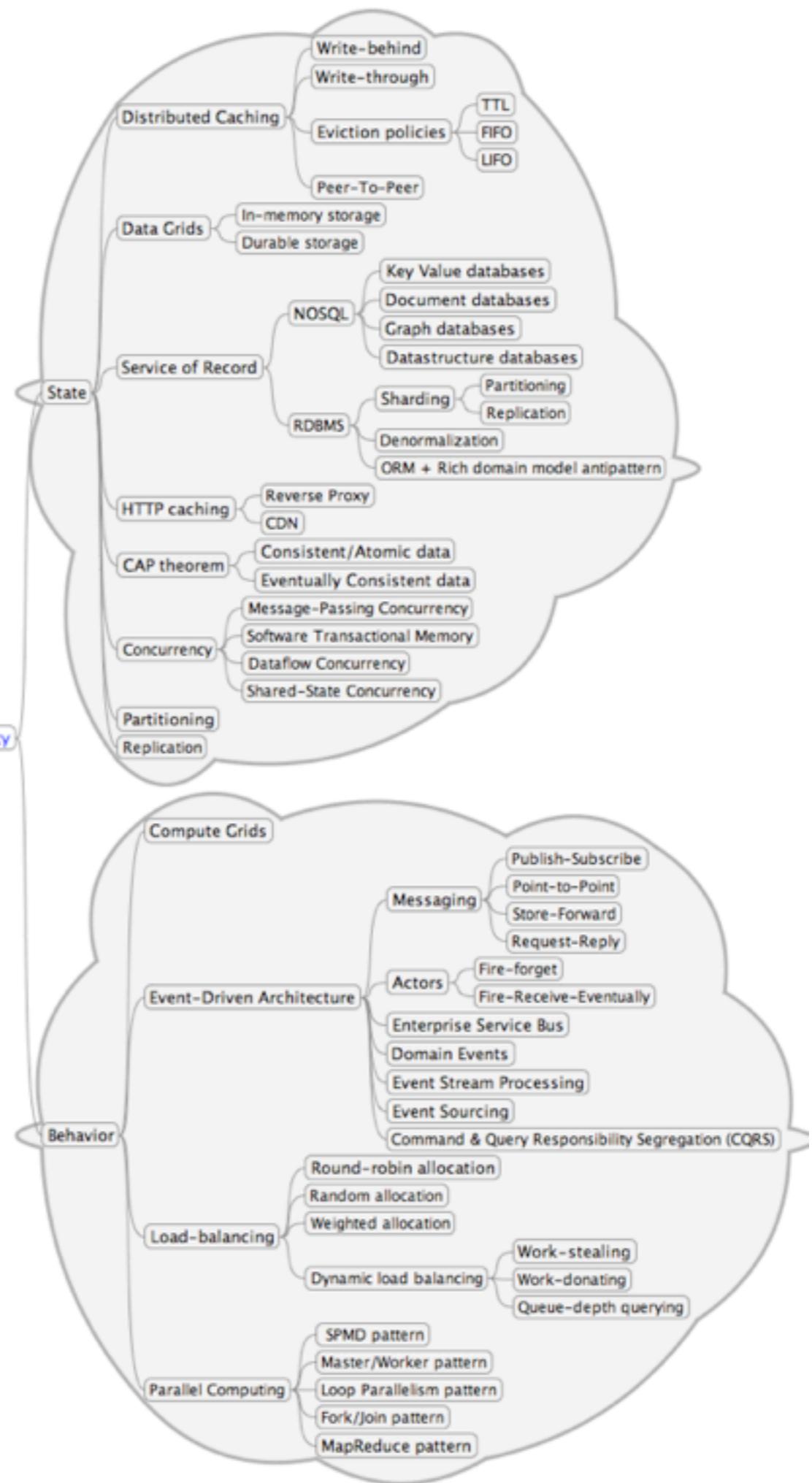
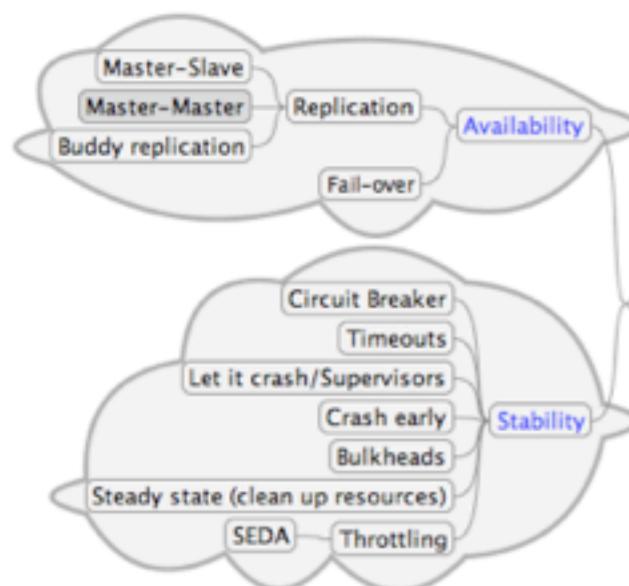


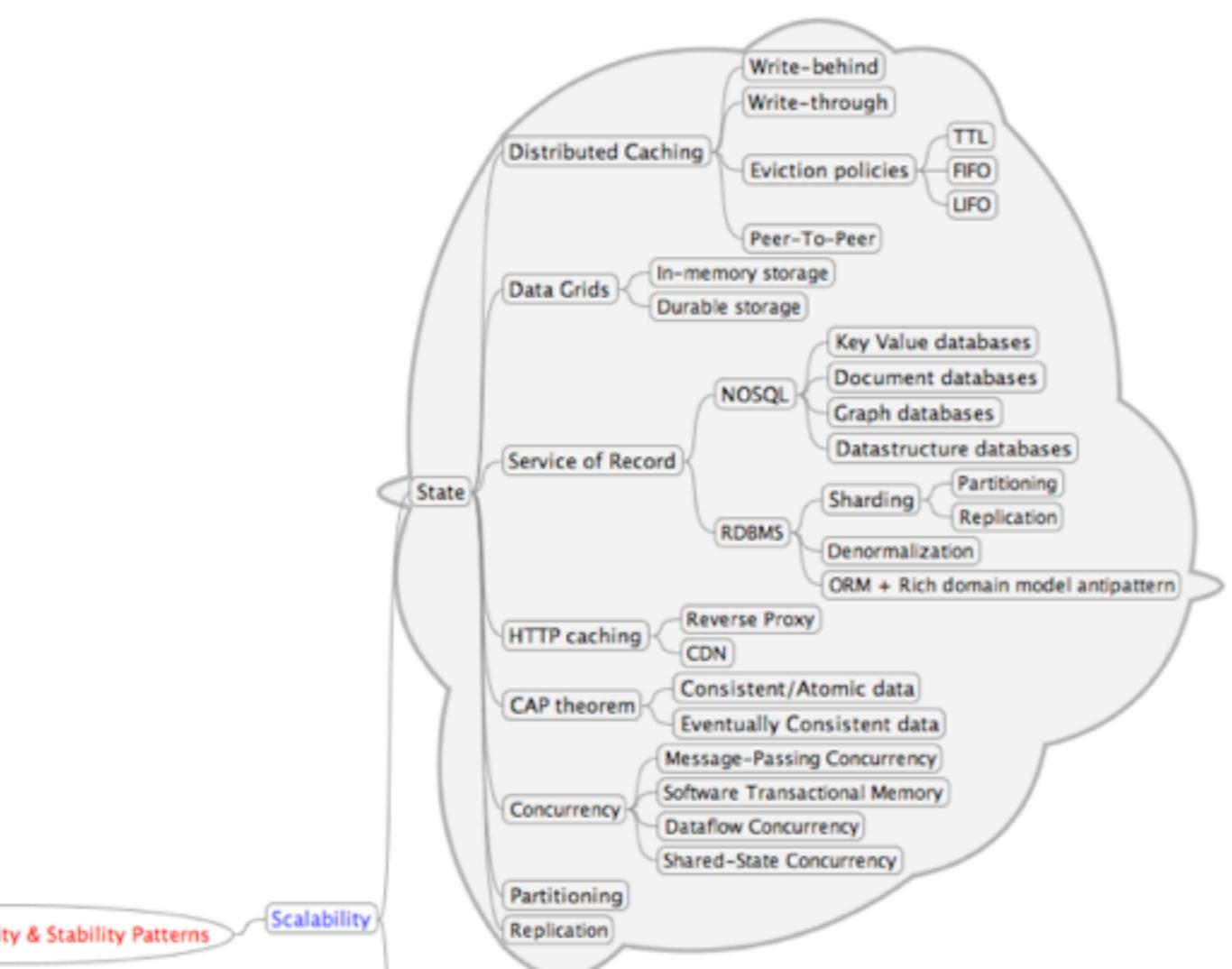
Scalability, Availability & Stability Patterns

Jonas Bonér¹
CTO Typesafe
twitter: @jboner

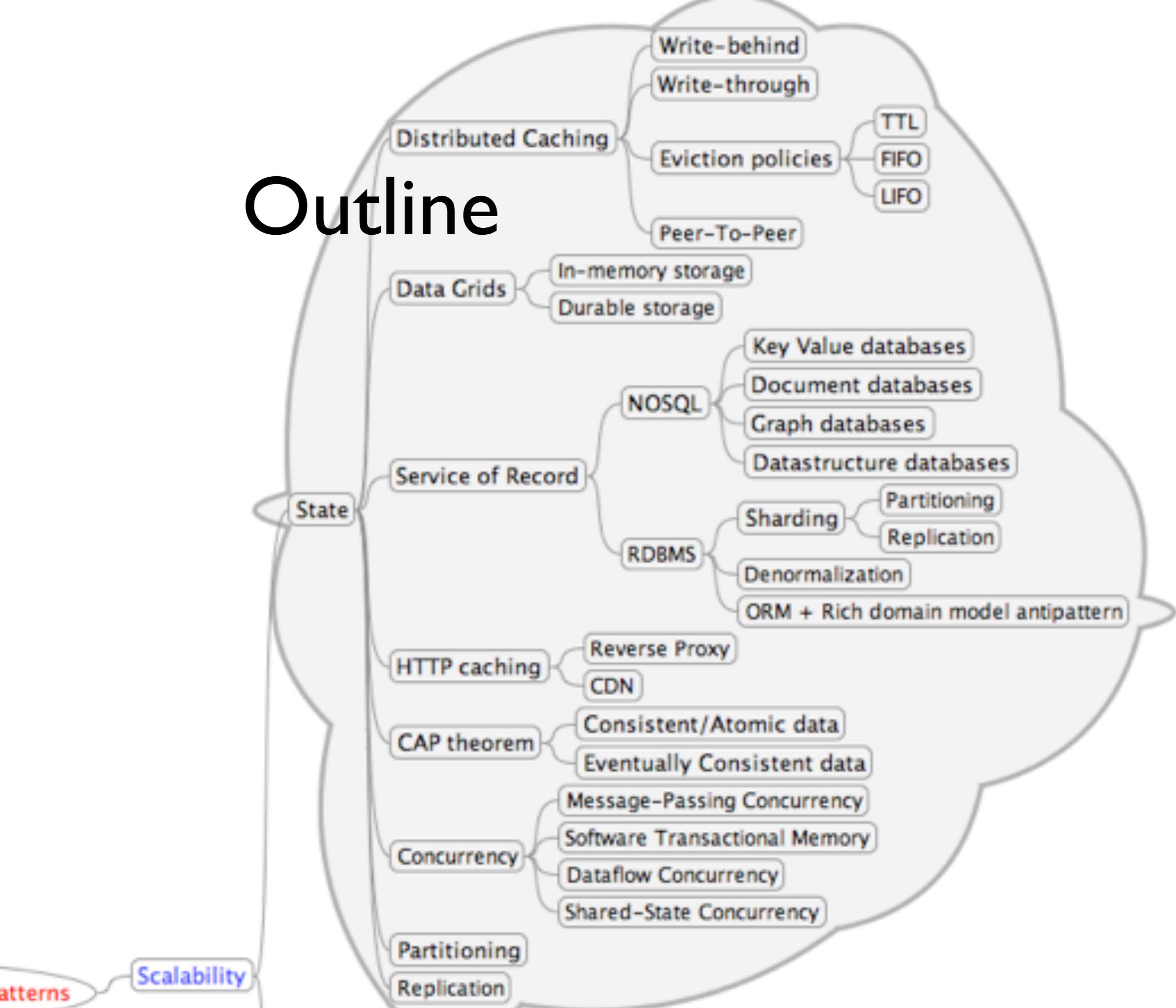
Outline



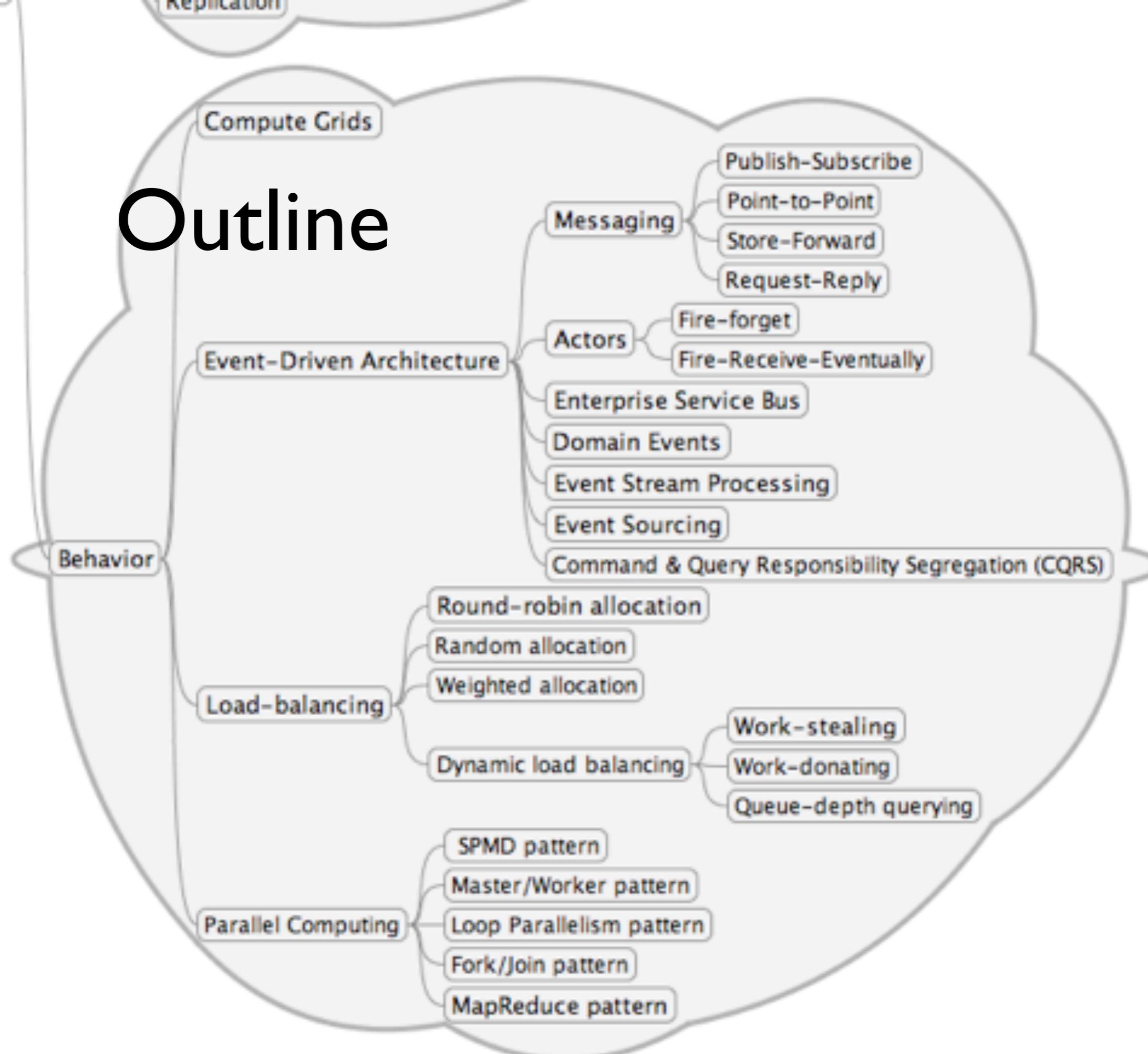
Outline



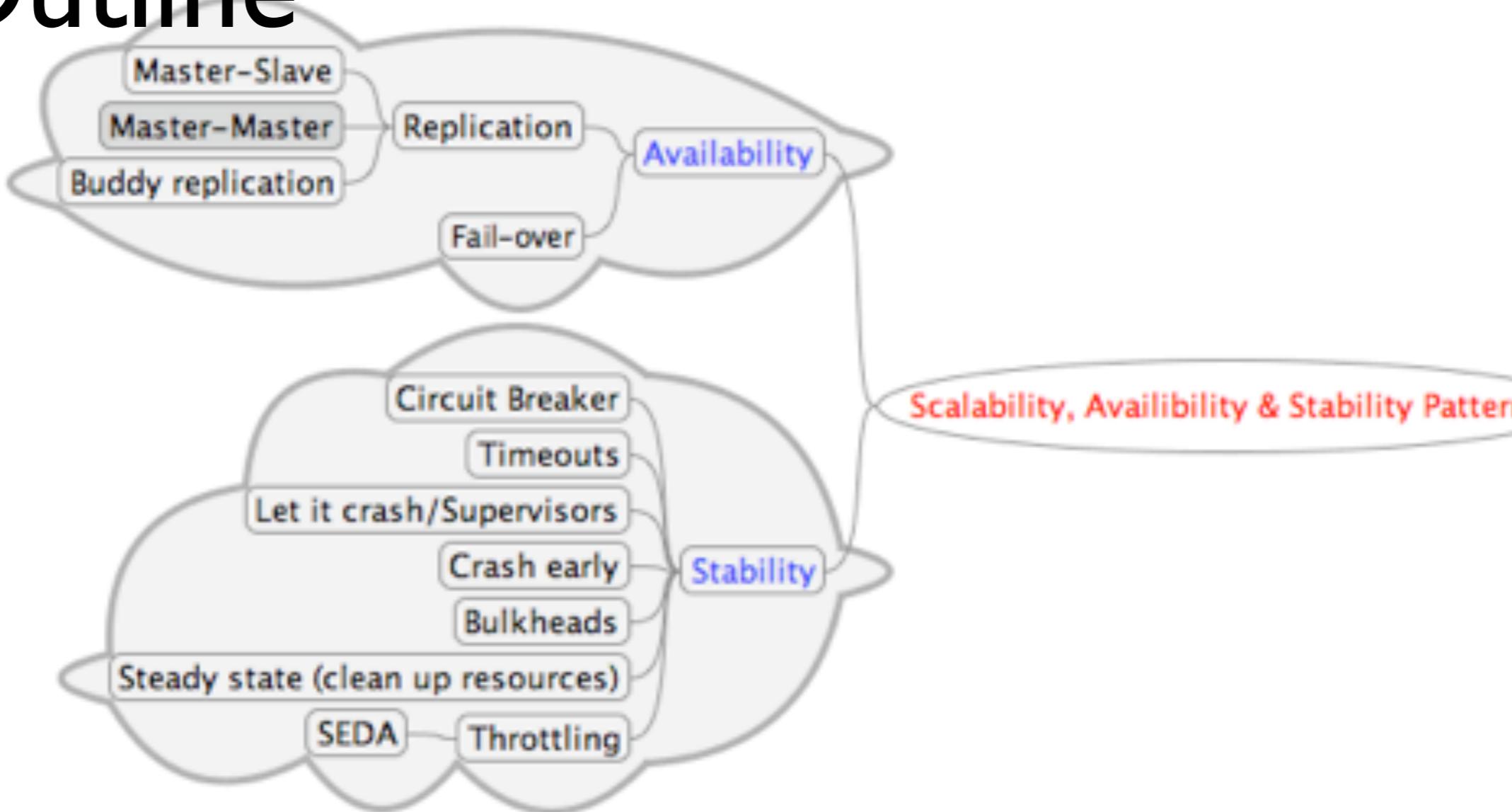
Outline



Outline



Outline



Introduction

Scalability Patterns



Managing Overload



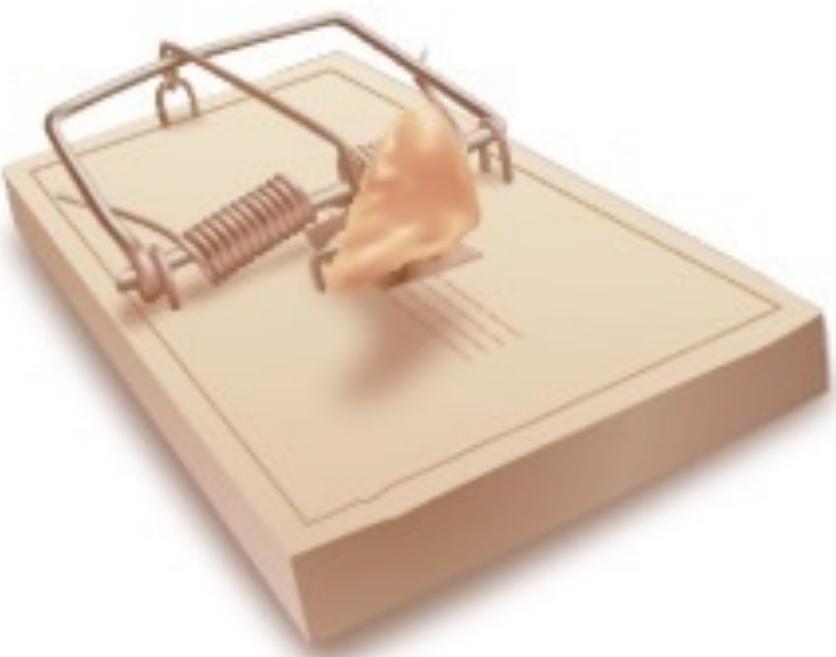
Scale up **vs** Scale out?



General recommendations

- Immutability as the default
- Referential Transparency (FP)
- Laziness
- Think about your data:
 - Different data need different guarantees

Scalability Trade-offs



***There is no
Free Lunch.***

Trade-offs

- Performance **vs** Scalability
- Latency **vs** Throughput
- Availability **vs** Consistency

Performance

vs

Scalability

How do I know if I have a
performance problem?

How do I know if I have a performance problem?

If your system is
slow for a **single user**

How do I know if I have a
scalability problem?

How do I know if I have a **scalability** problem?

If your system is
fast for a **single user**
but **slow** under **heavy load**

Latency

vs

Throughput

You should strive for
maximal throughput
with
acceptable latency

Availability

VS

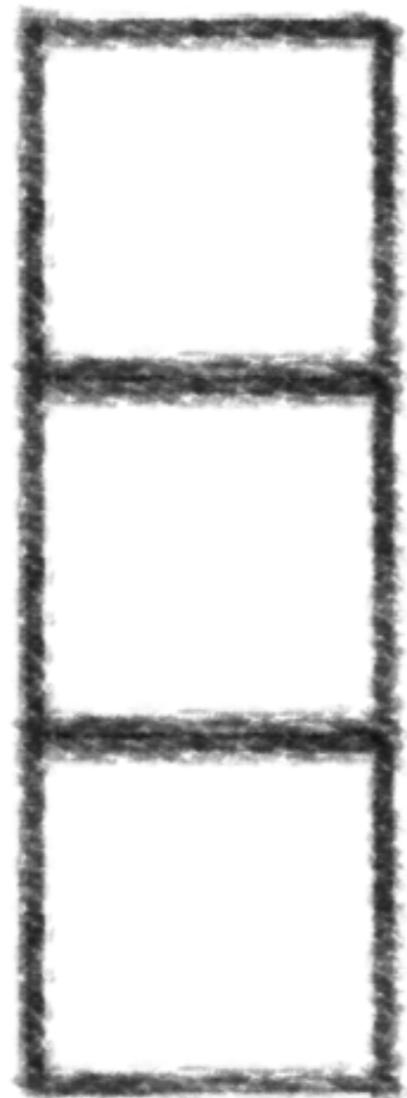
Consistency

Brewer's

CAP

theorem

You can only pick **2**



Consistency

Availability

Partition tolerance

At a given point in time

Centralized system

- In a **centralized system** (RDBMS etc.) we don't have network partitions, e.g. **P** in CAP
- So you **get both**:
 - **A**vailability
 - **C**onsistency

[Atomic

[Consistent

[Isolated

[Durable

Distributed system

- In a **distributed system** we (will) have network partitions, e.g. **P** in CAP
- So you get to **only pick one**:
 - **A**vailability
 - **C**onsistency

CAP in practice:

- ...there are **only two types** of systems:
 1. **CP**
 2. **AP**
- ...there is **only one choice** to make. In case of a network partition, what do you sacrifice?
 1. **C: Consistency**
 2. **A: Availability**

Basically **A**vailable

Soft state

Eventually consistent

Eventual Consistency

...is an interesting trade-off

Eventual Consistency

...is an interesting trade-off

But let's get back to that later

Availability Patterns

Availability Patterns

- Fail-over
- Replication
 - Master-Slave
 - Tree replication
 - Master-Master
 - Buddy Replication

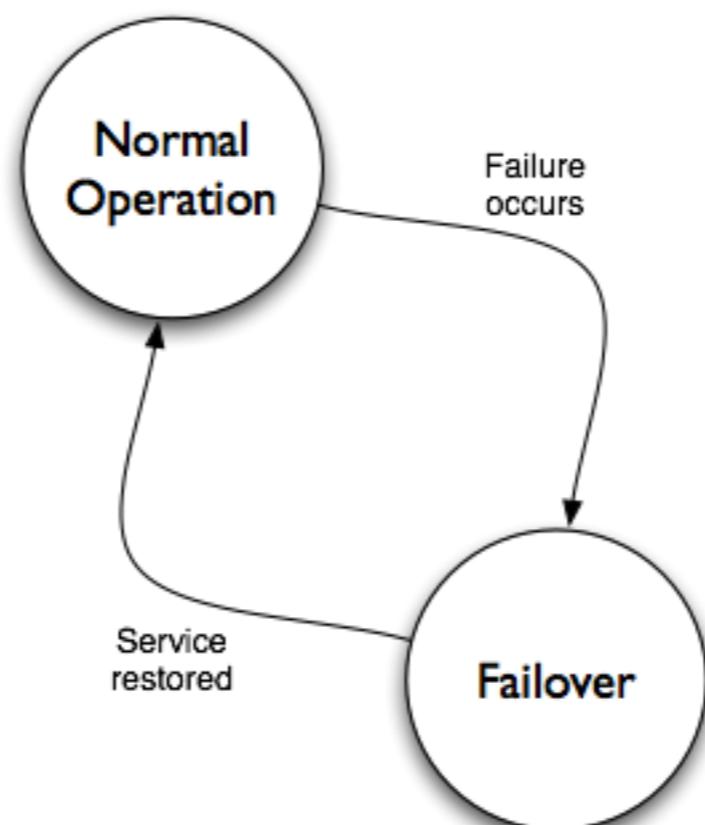
What do we mean with Availability?



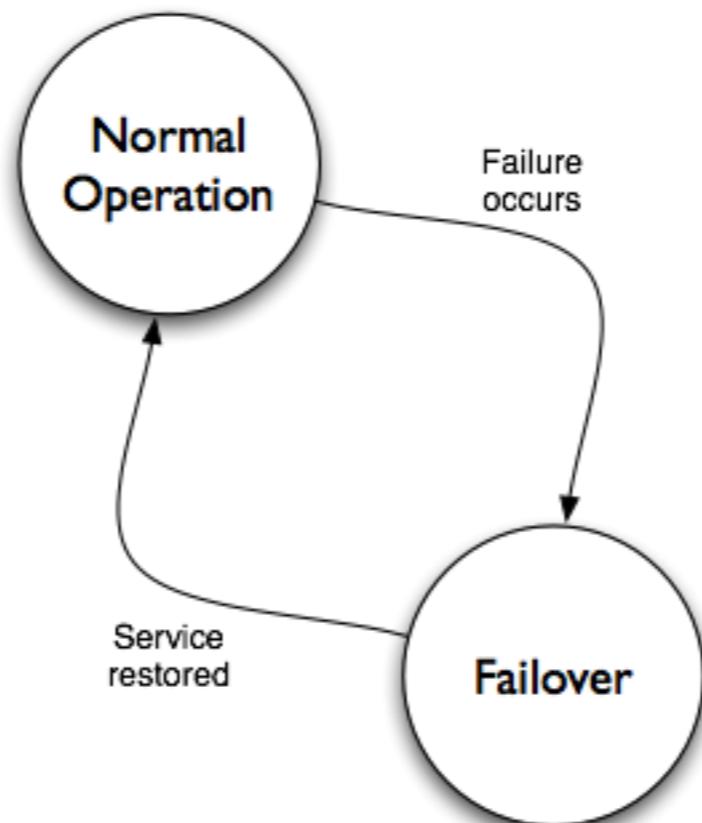
Fail-over



Fail-over

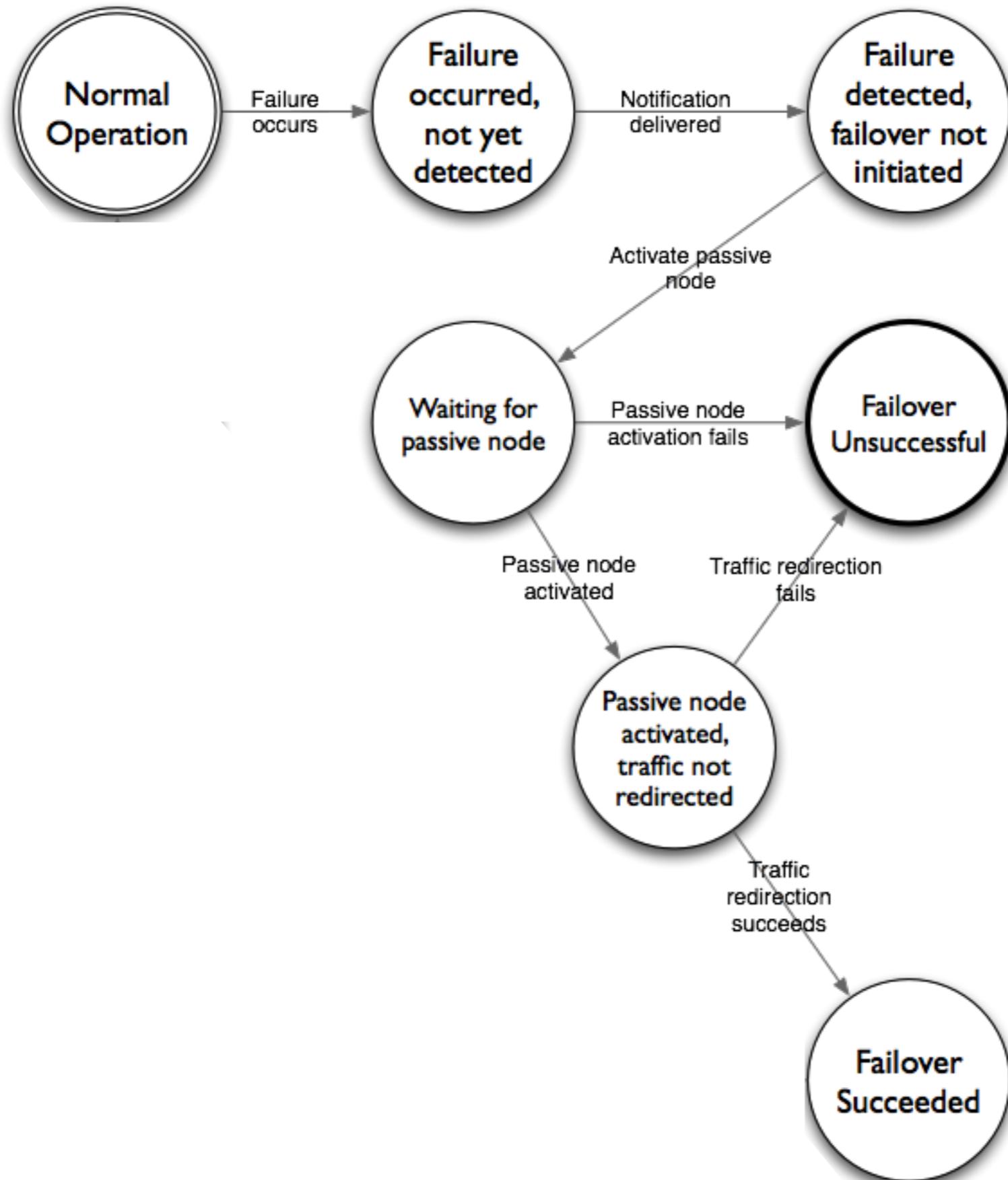


Fail-over

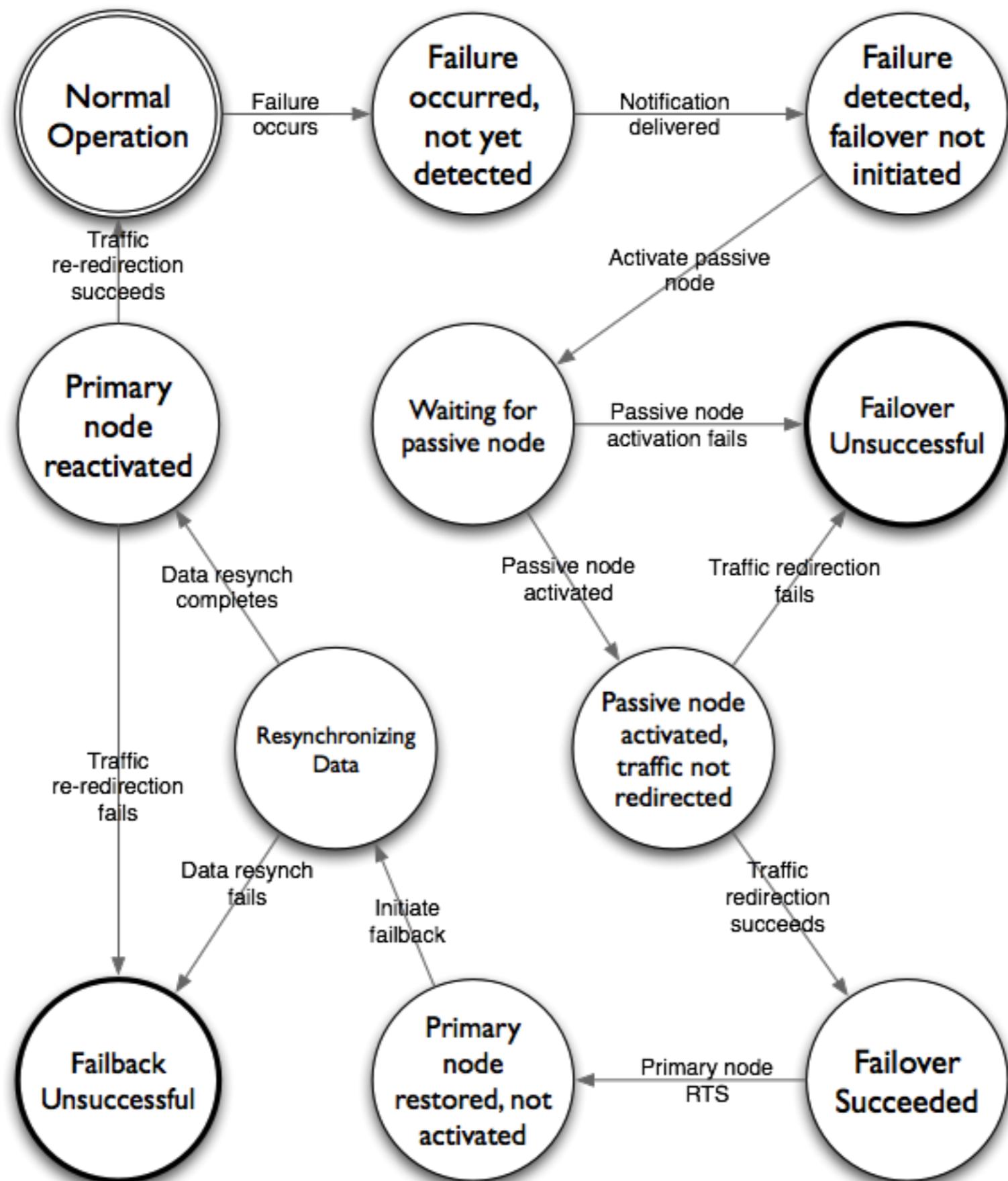


But fail-over is not always this simple

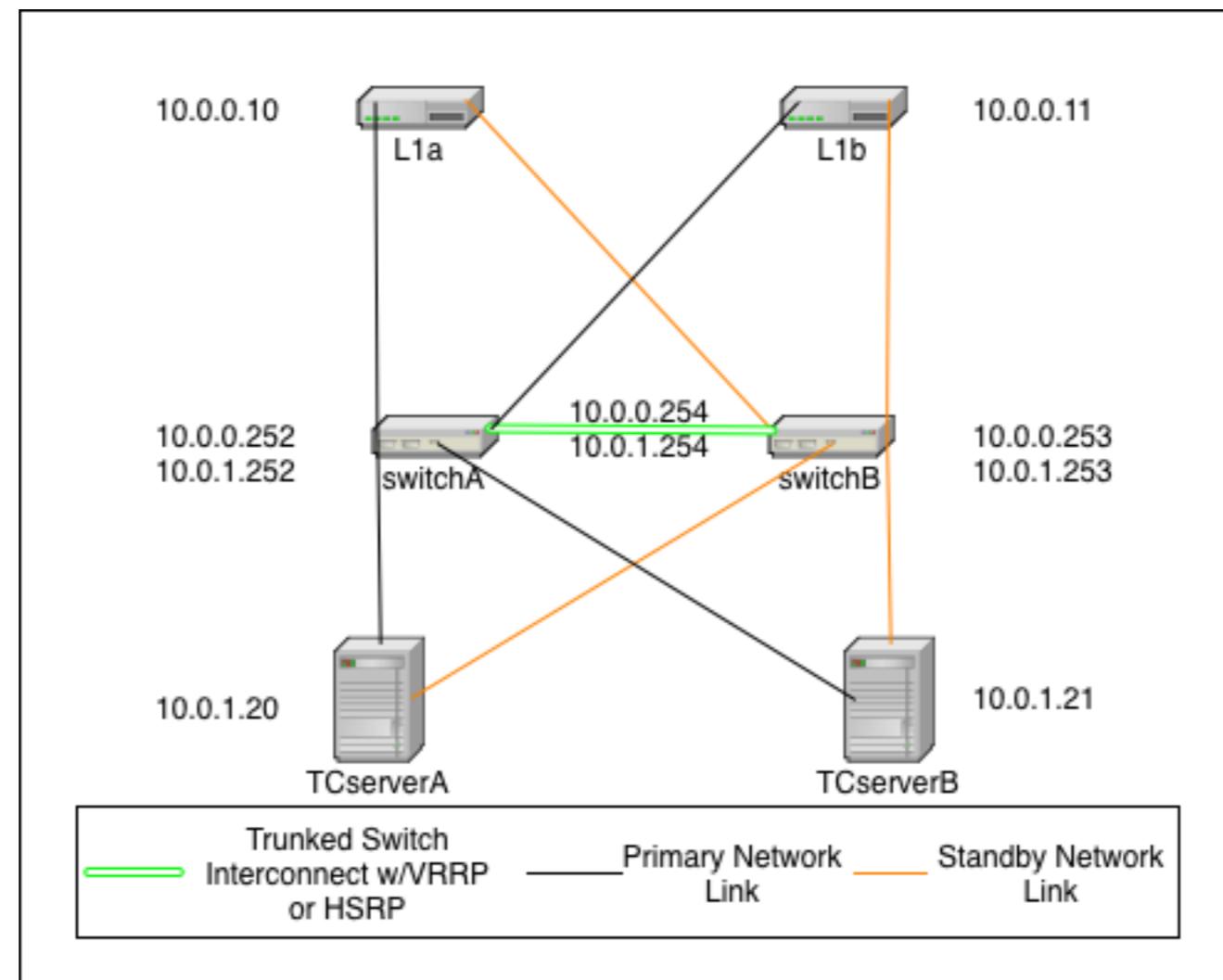
Fail-over

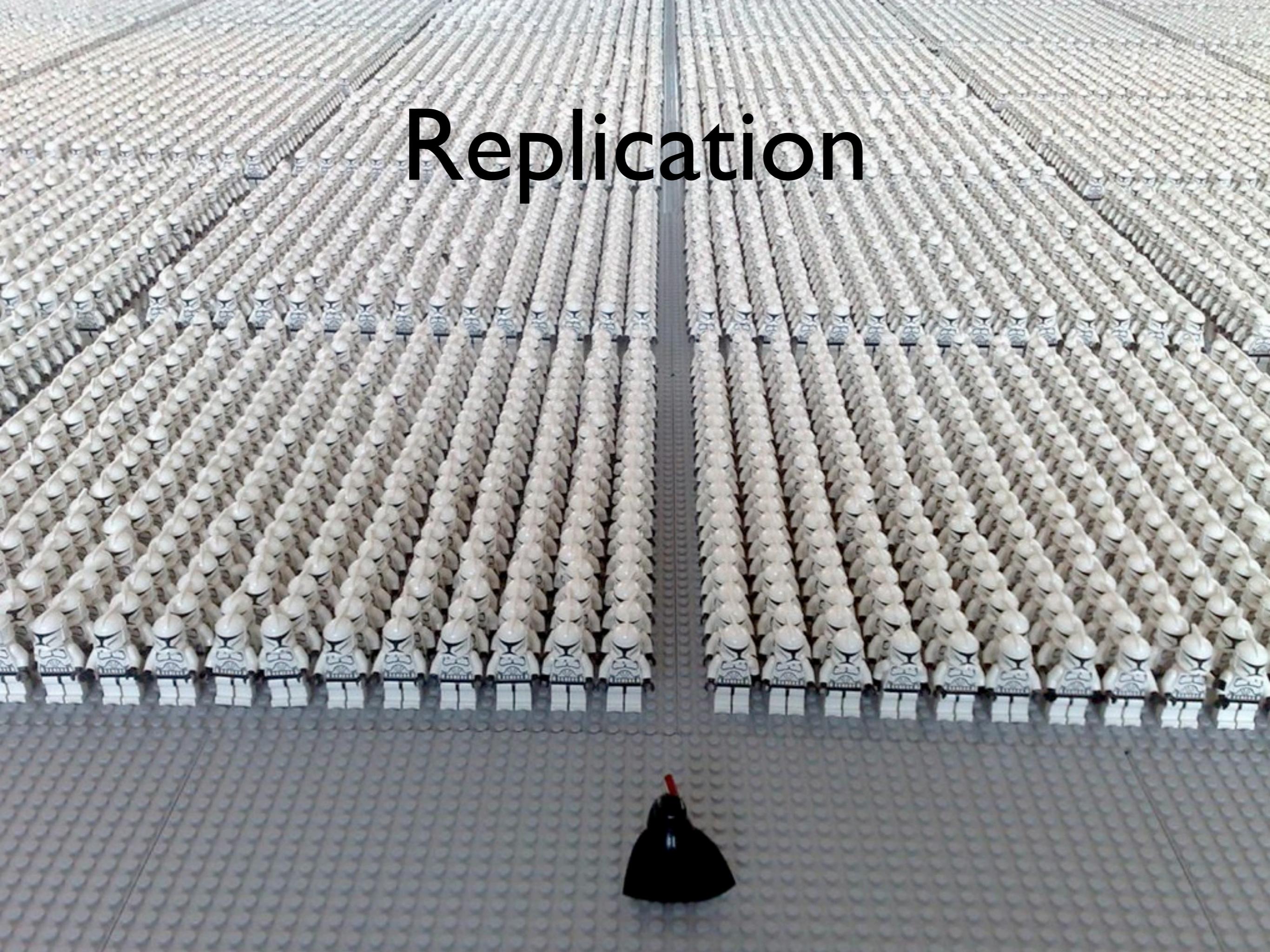


Fail-back



Network fail-over





Replication

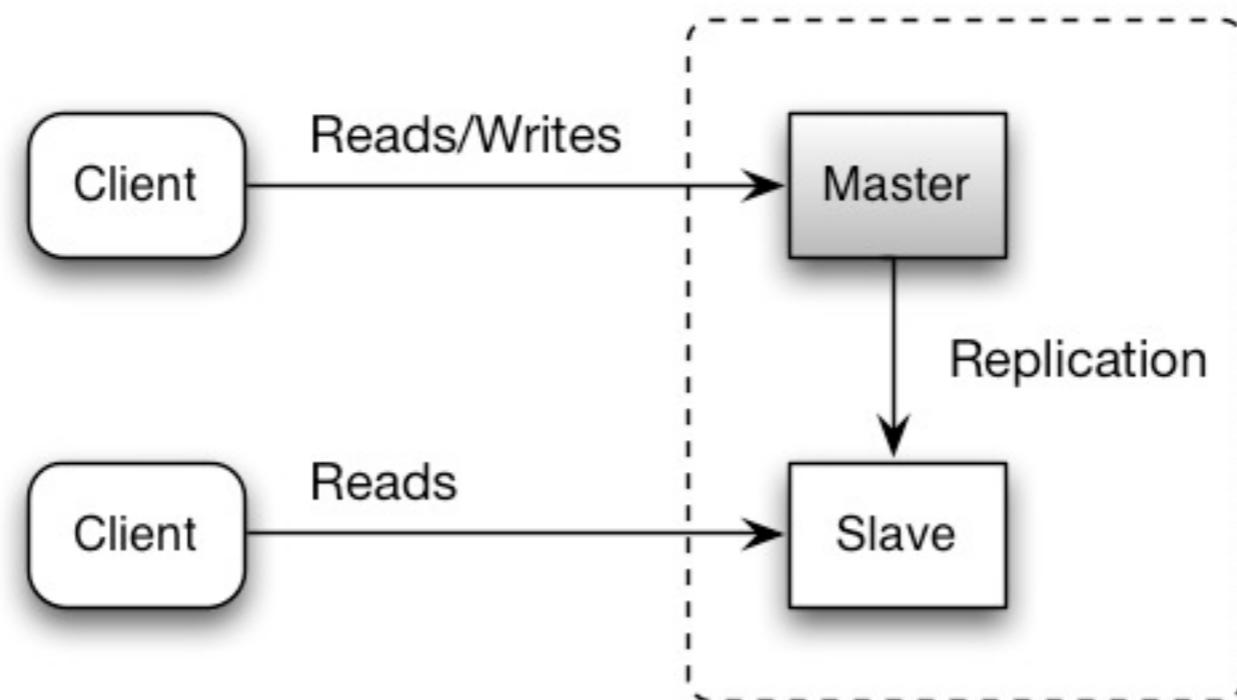
Replication

- Active replication - Push
- Passive replication - Pull
 - Data not available, read from peer, then store it locally
 - Works well with timeout-based caches

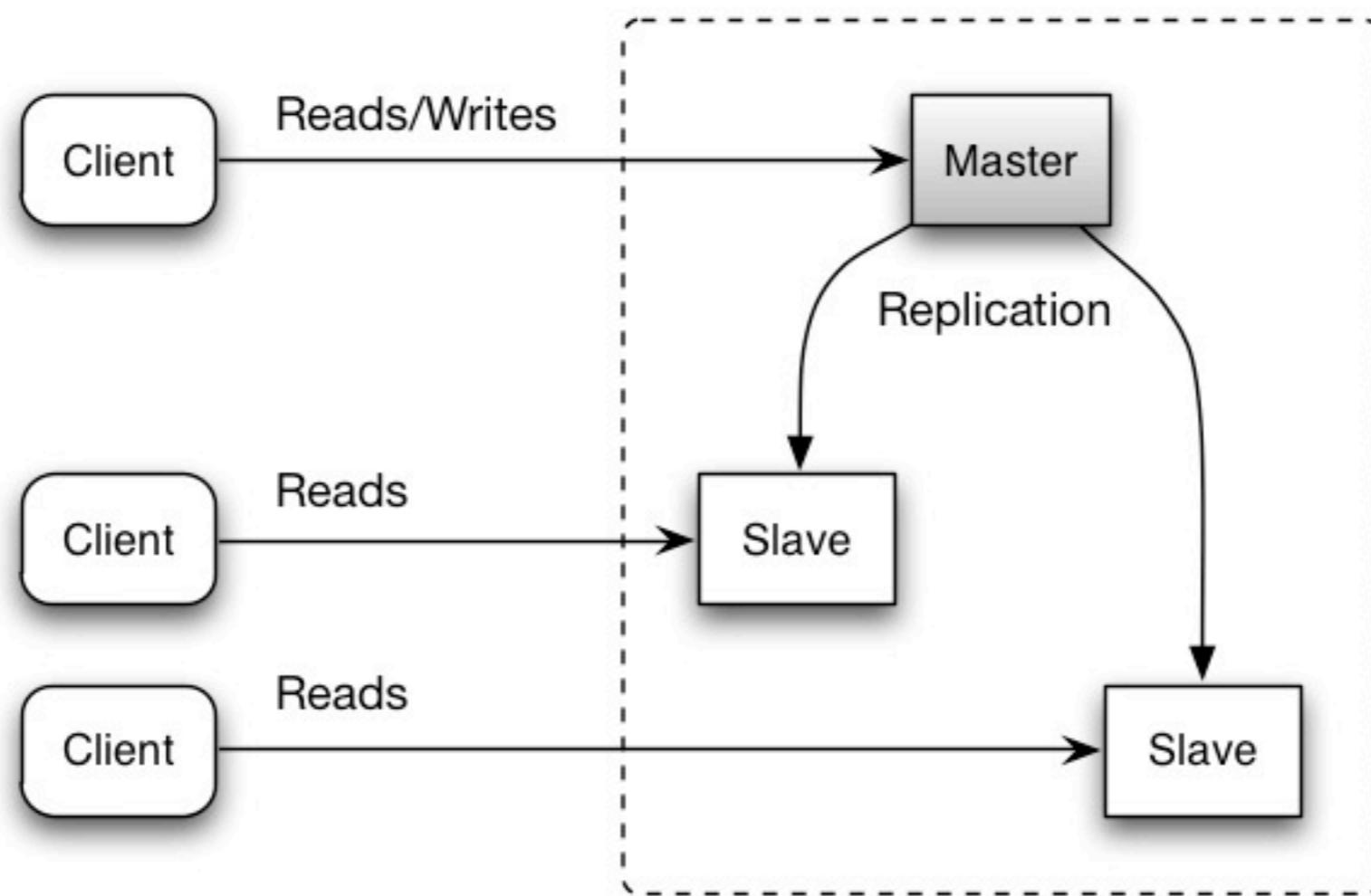
Replication

- Master-Slave replication
- Tree Replication
- Master-Master replication
- Buddy replication

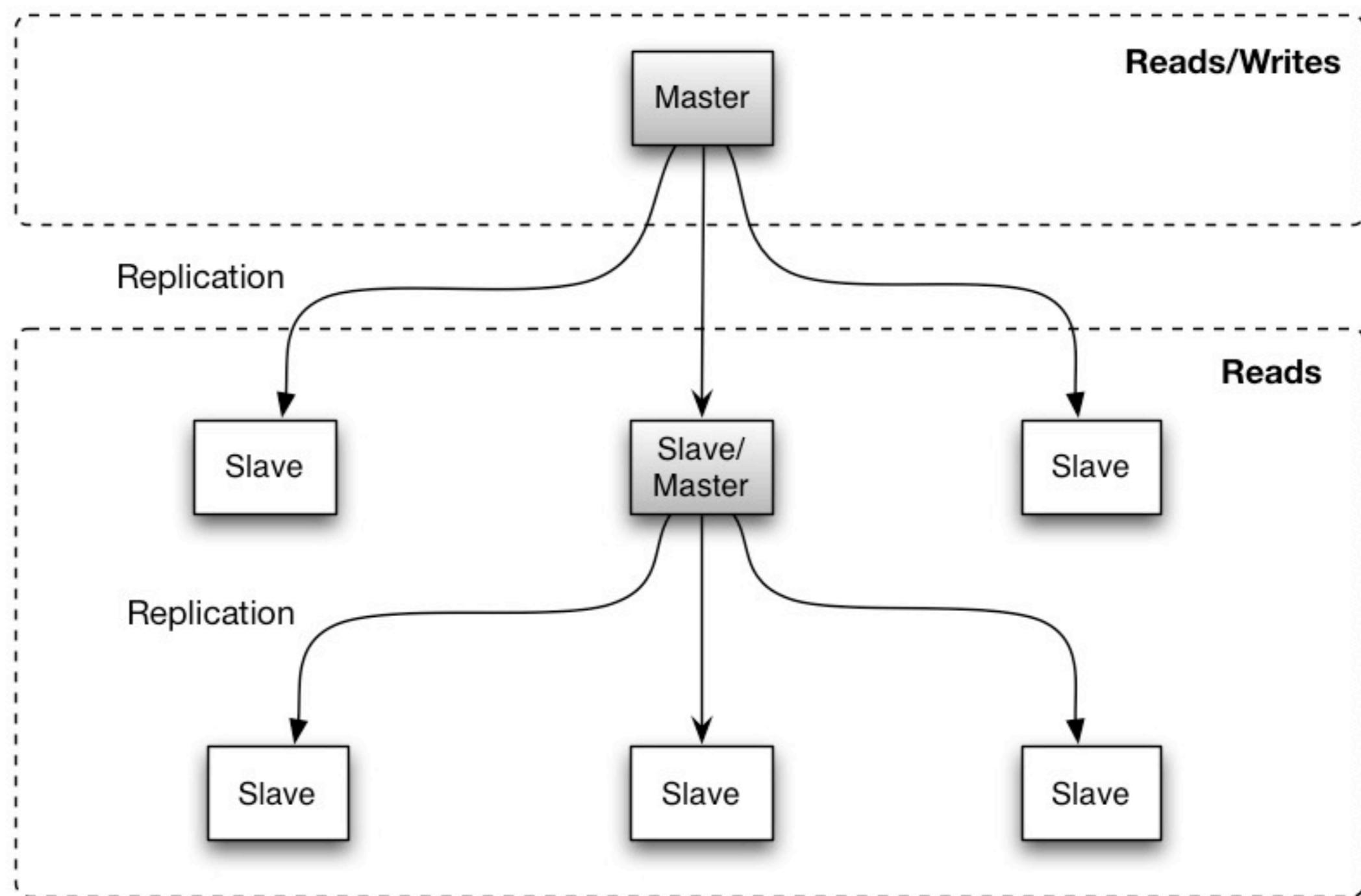
Master-Slave Replication



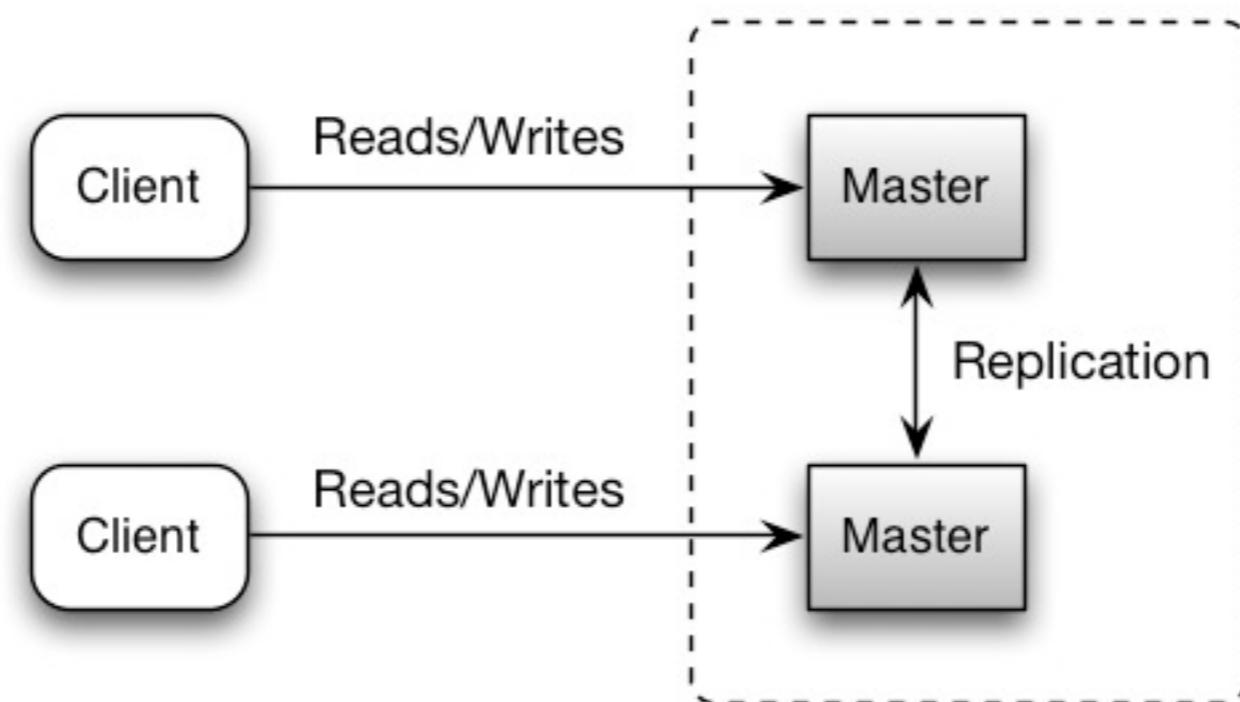
Master-Slave Replication



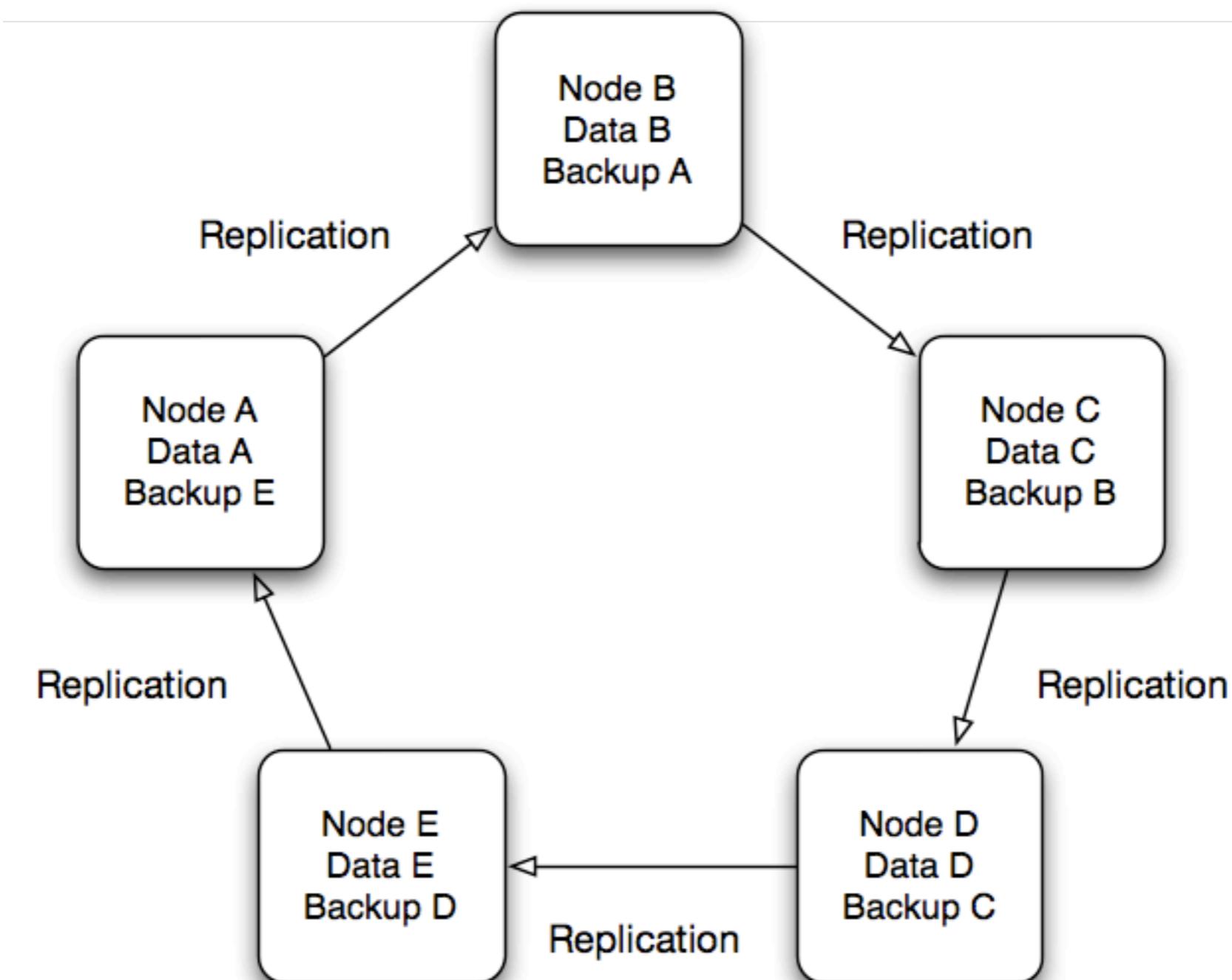
Tree Replication



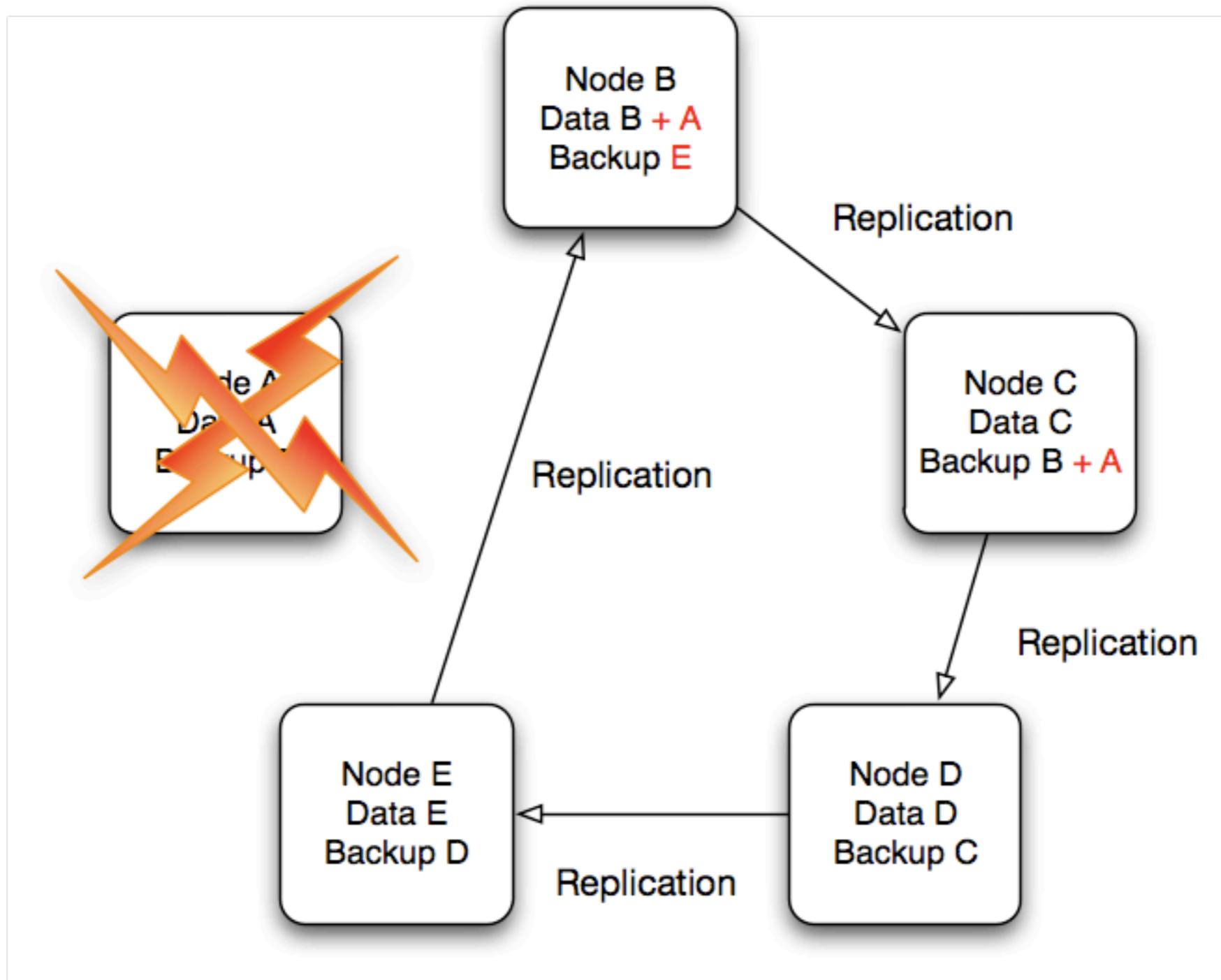
Master-Master Replication



Buddy Replication



Buddy Replication

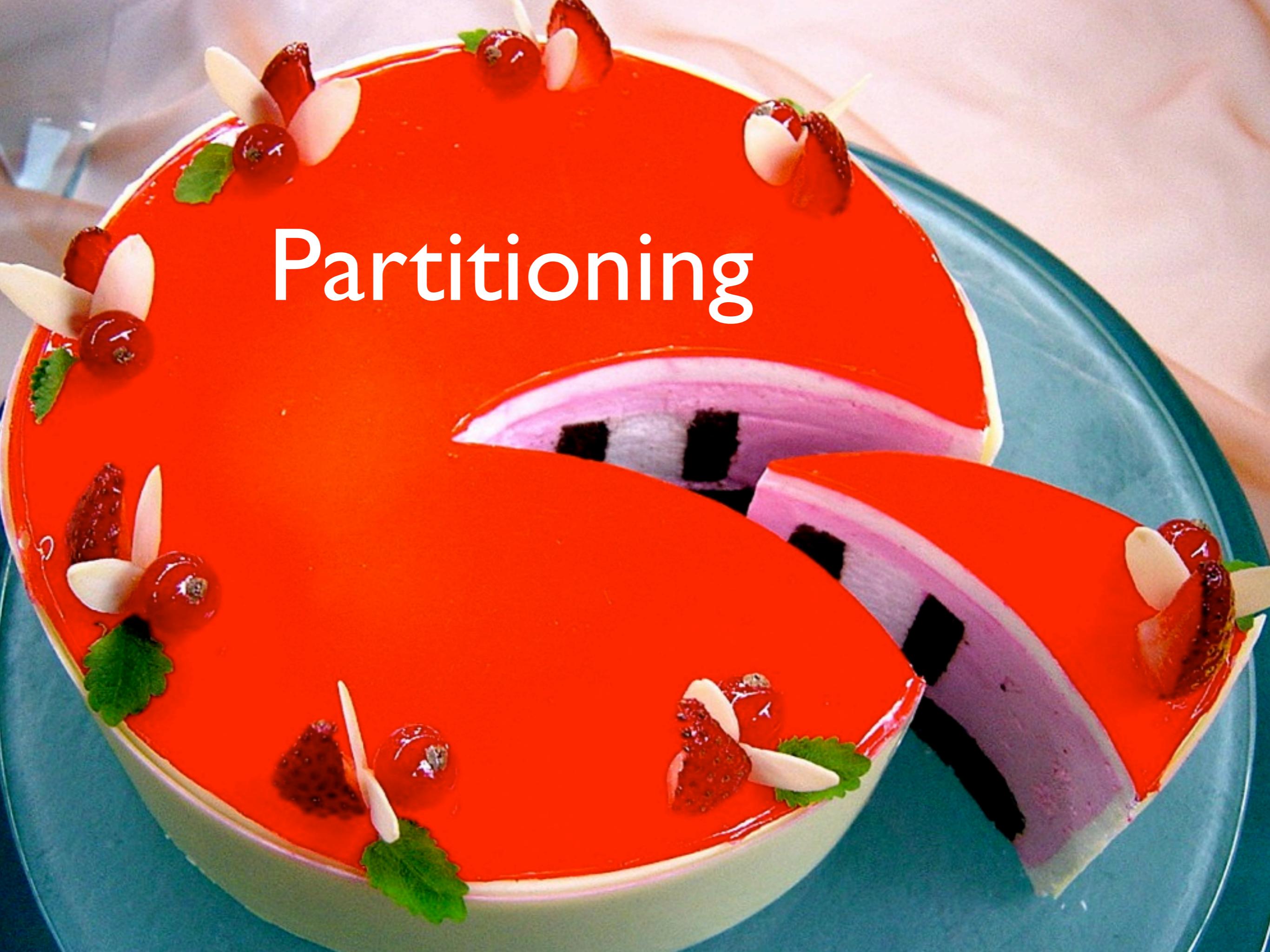


Scalability Patterns: State

Scalability Patterns: State

- Partitioning
- HTTP Caching
- RDBMS Sharding
- NOSQL
- Distributed Caching
- Data Grids
- Concurrency

Partitioning



HTTP Caching

Reverse Proxy

- Varnish
- Squid
- rack-cache
- Pound
- Nginx
- Apache mod_proxy
- Traffic Server

HTTP Caching

CDN, Akamai



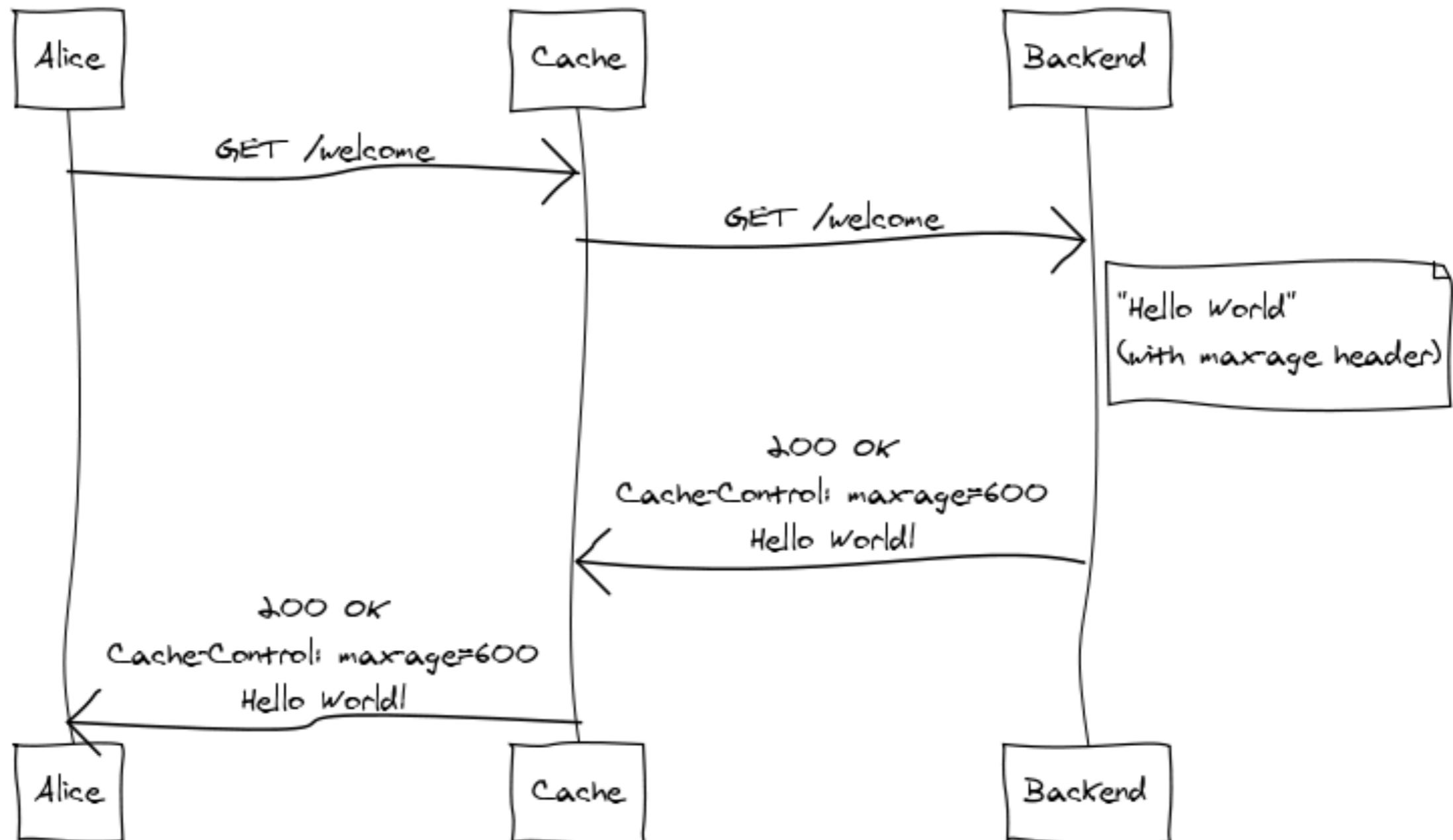
Generate Static Content

Precompute content

- Homegrown + cron or Quartz
- Spring Batch
- Gearman
- Hadoop
- Google Data Protocol
- Amazon Elastic MapReduce

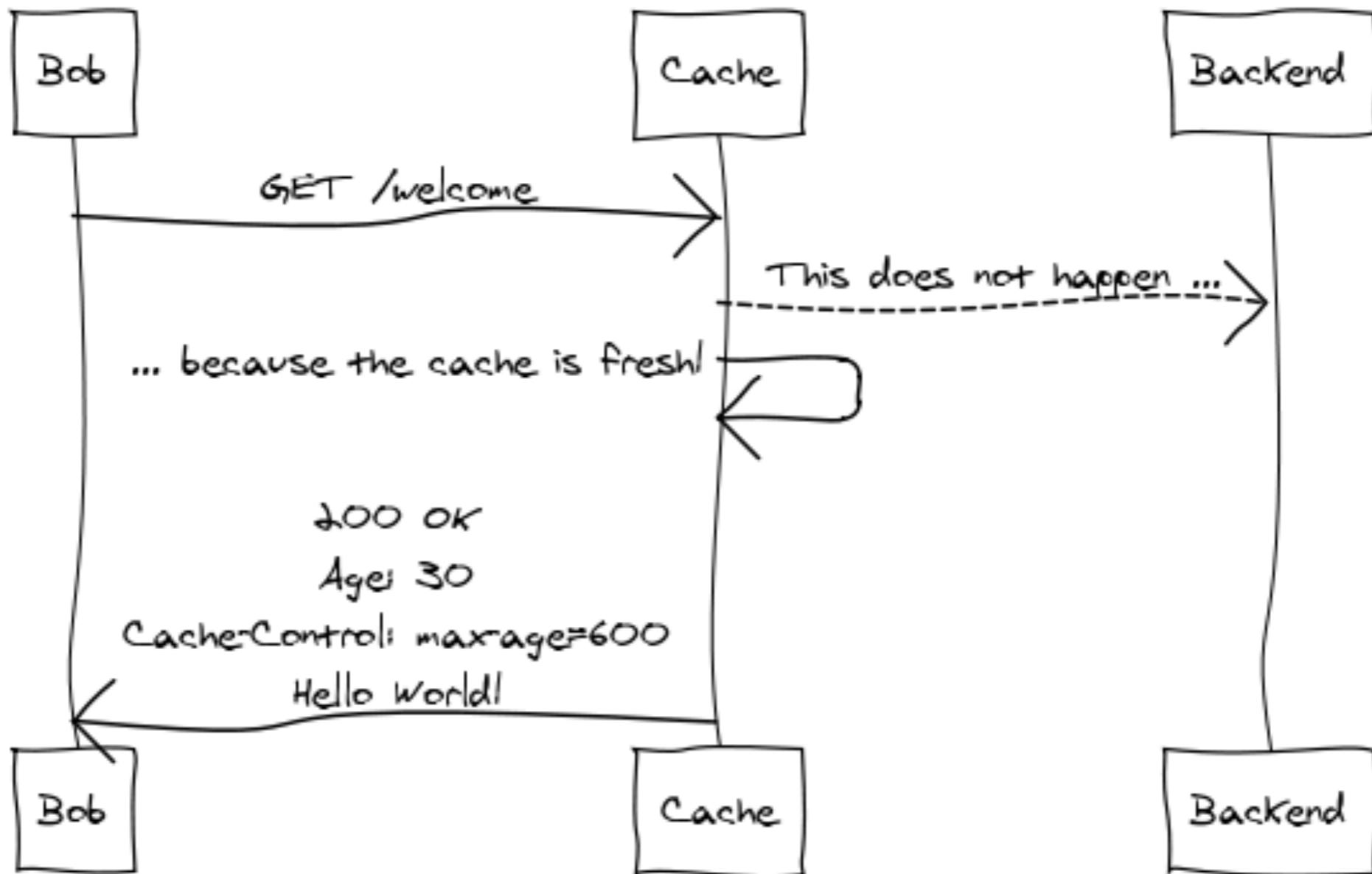
HTTP Caching

First request



HTTP Caching

Subsequent request

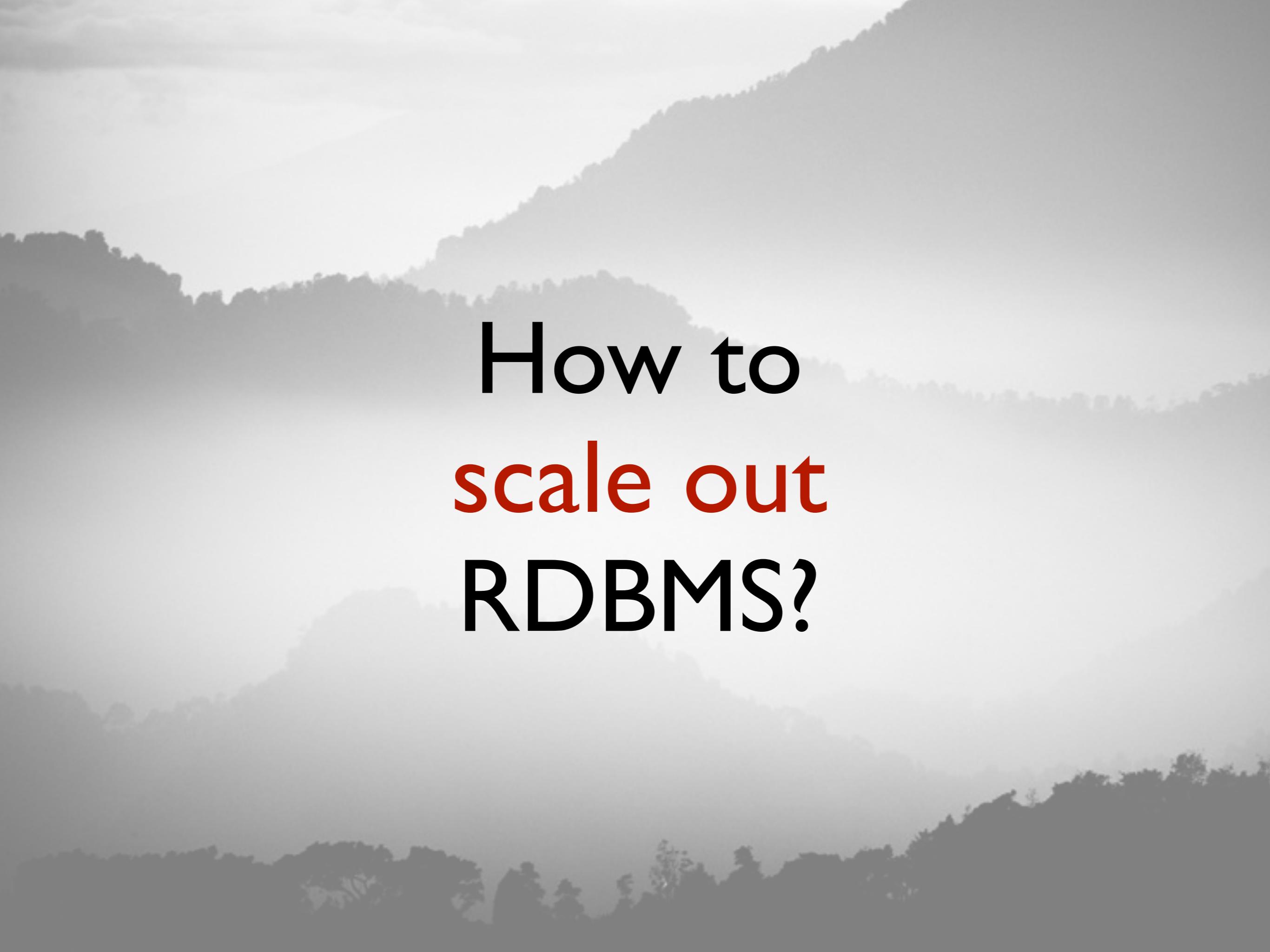


Service of Record

SoR

Service of Record

- Relational Databases (RDBMS)
- NOSQL Databases

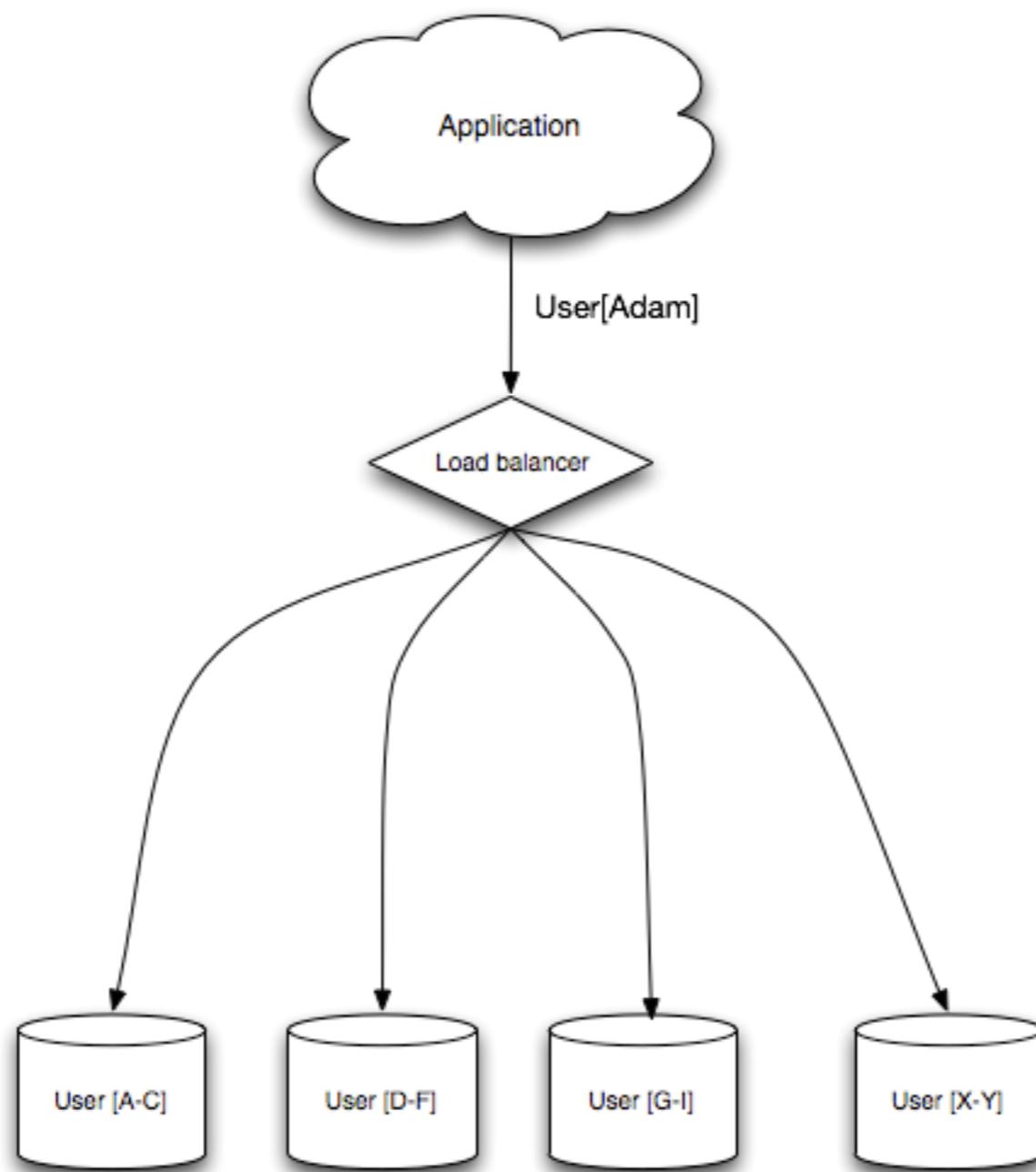
A grayscale photograph of a misty, mountainous landscape. The foreground is dominated by dark, silhouetted tree tops. In the middle ground, several layers of mountains are visible, their peaks obscured by thick fog or low-hanging clouds. The overall atmosphere is hazy and ethereal.

How to
scale out
RDBMS?

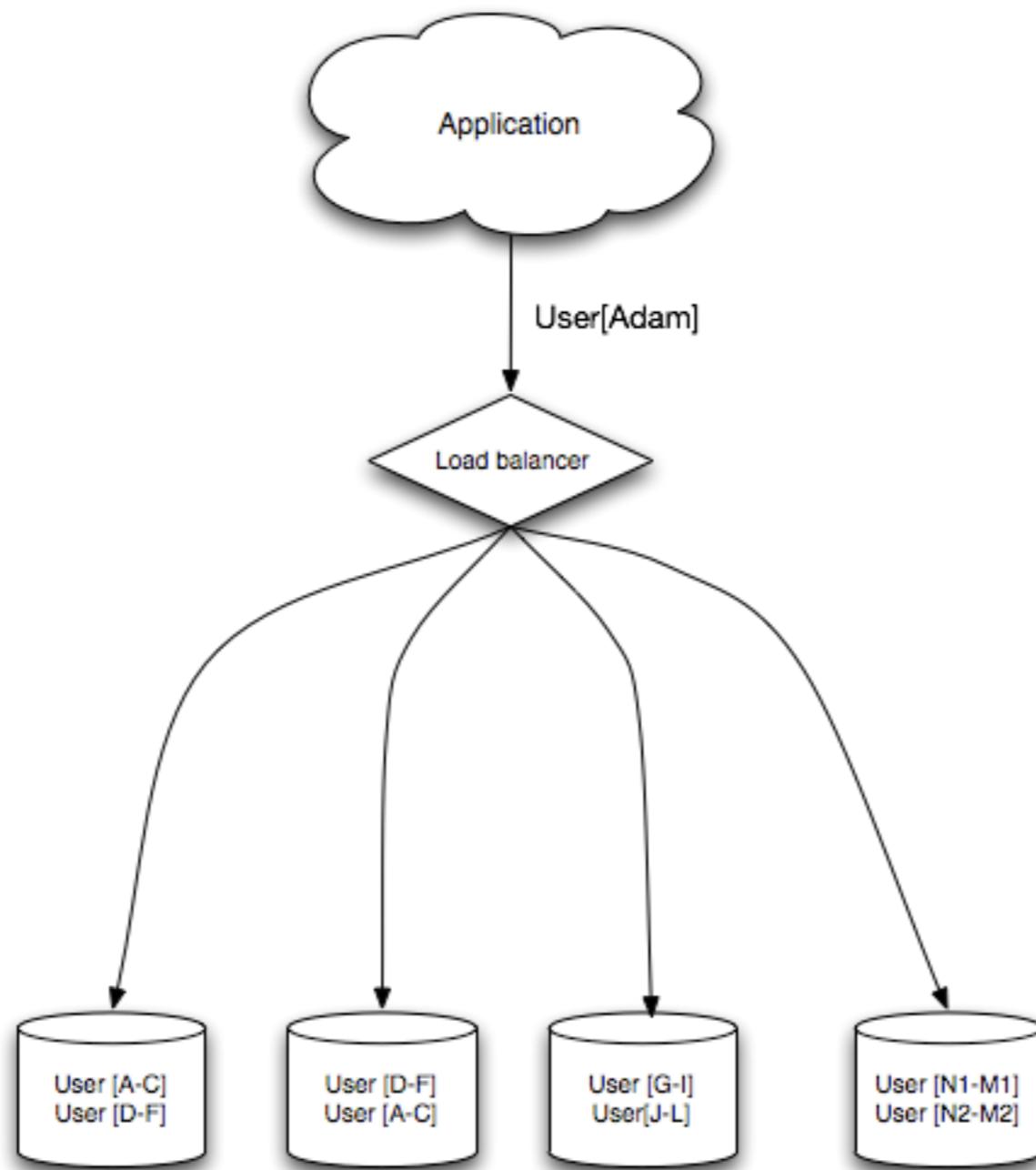
Sharding

- Partitioning
- Replication

Sharding: Partitioning



Sharding: Replication



ORM + rich domain model anti-pattern

- Attempt:
 - Read an object from DB
- Result:
 - You sit with your whole database in your lap

Think about your data

Think again

- When do you need ACID?
- When is Eventually Consistent a better fit?
- Different kinds of data has different needs

When is
a RDBMS

not

good enough?

Scaling reads
to a RDBMS
is hard

Scaling writes
to a RDBMS

is impossible

Do we
really need
a RDBMS?

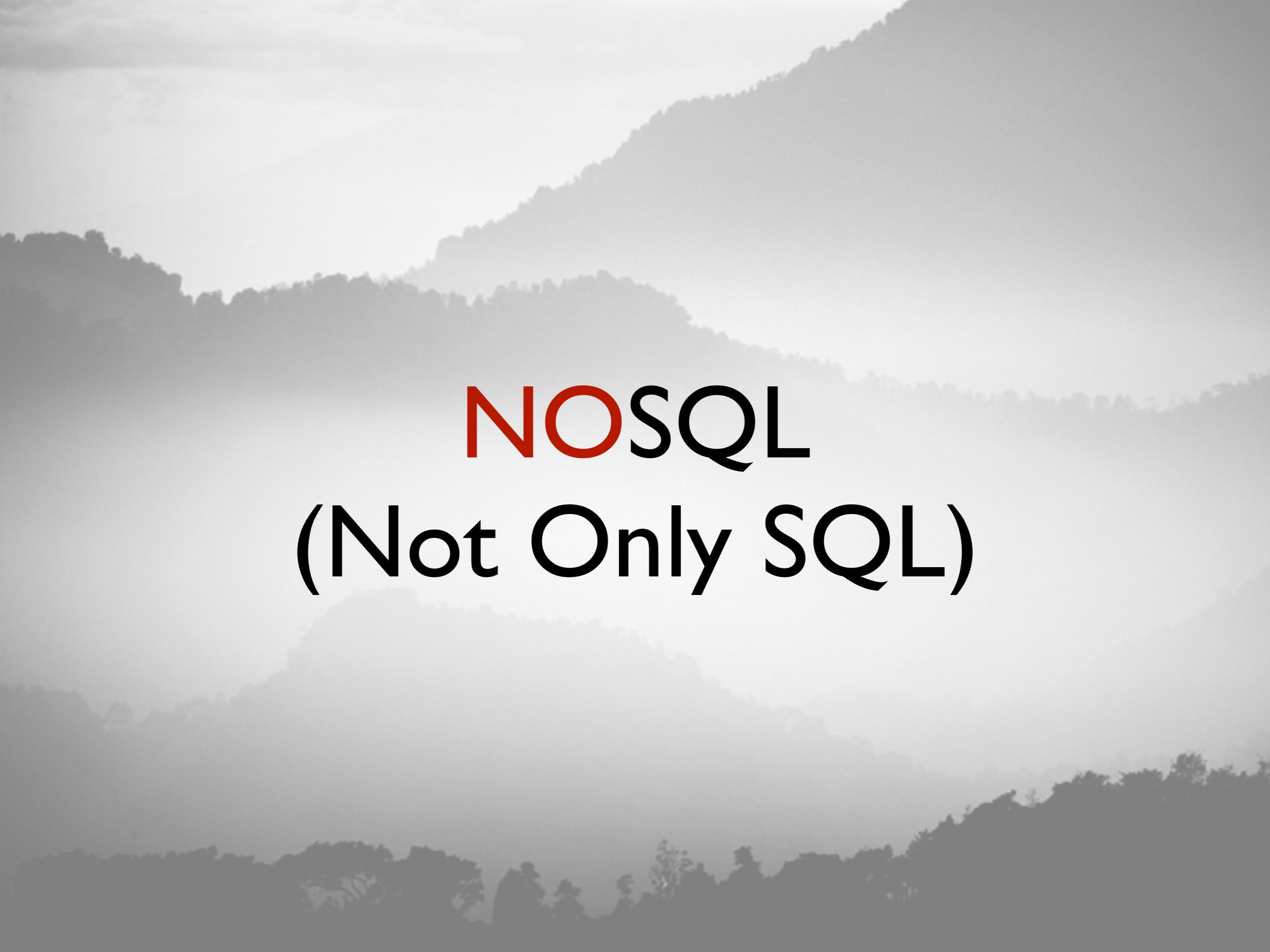
Do we
really need
a RDBMS?

Sometimes...

Do we
really need
a RDBMS?

Do we
really need
a RDBMS?

But many times we don't

A grayscale photograph of a misty mountain range. The peaks are dark and silhouetted against a lighter sky, with layers of fog or clouds obscuring the mid-ground and foreground.

NOSQL

(Not Only SQL)

NOSQL

- Key-Value databases
- Column databases
- Document databases
- Graph databases
- Datastructure databases

Who's ACID?

- Relational DBs (MySQL, Oracle, Postgres)
- Object DBs (Gemstone, db4o)
- Clustering products (Coherence, Terracotta)
- Most caching products (ehcache)

Who's BASE?

Distributed databases

- Cassandra
- Riak
- Voldemort
- Dynomite,
- SimpleDB
- etc.

NOSQL in the wild

- Google: **Bigtable**
- Amazon: **Dynamo**
- Amazon: **SimpleDB**
- Yahoo: **HBase**
- Facebook: **Cassandra**
- LinkedIn: **Voldemort**

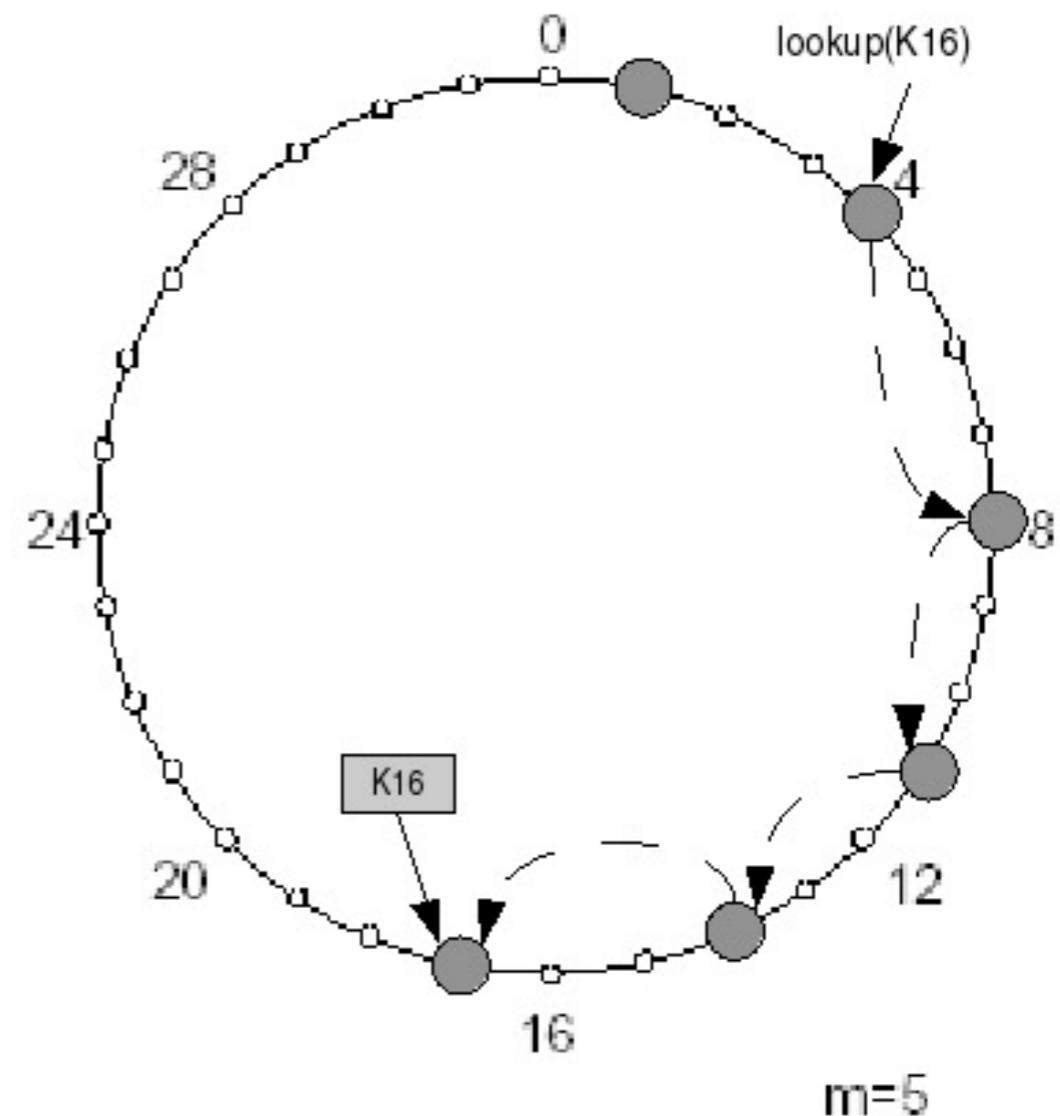
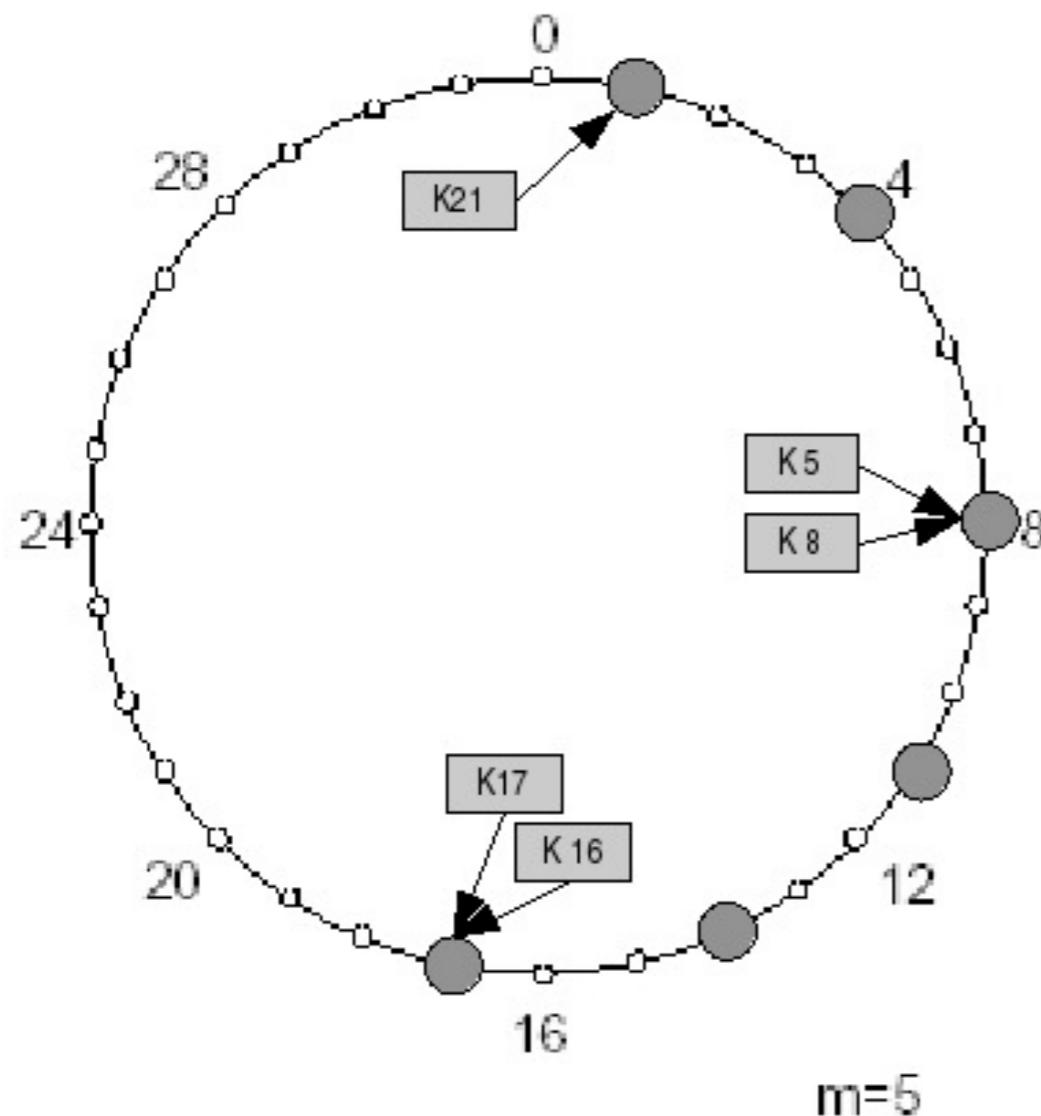
The background of the image is a soft, monochromatic landscape. It features several layers of mountains or hills, all shrouded in a thick, white fog or mist. In the immediate foreground, there's a dark silhouette of what appears to be a forest or a row of trees. The overall atmosphere is hazy and ethereal.

But first some background...

Chord & Pastry

- Distributed Hash Tables (DHT)
- Scalable
- Partitioned
- Fault-tolerant
- Decentralized
- Peer to peer
- Popularized
 - Node ring
 - Consistent Hashing

Node ring with Consistent Hashing



Find data in $\log(N)$ jumps

Bigtable

“How can we build a DB on top of Google File System?”

- Paper: Bigtable: A distributed storage system for structured data, 2006
- Rich data-model, structured storage
- Clones:

HBase

Hypertable

Neptune

Dynamo

“How can we build a distributed hash table for the data center?”

- Paper: Dynamo: Amazon’s highly available key-value store, 2007
- Focus: partitioning, replication and availability
- Eventually Consistent
- Clones:

Voldemort

Dynomite

Types of NOSQL stores

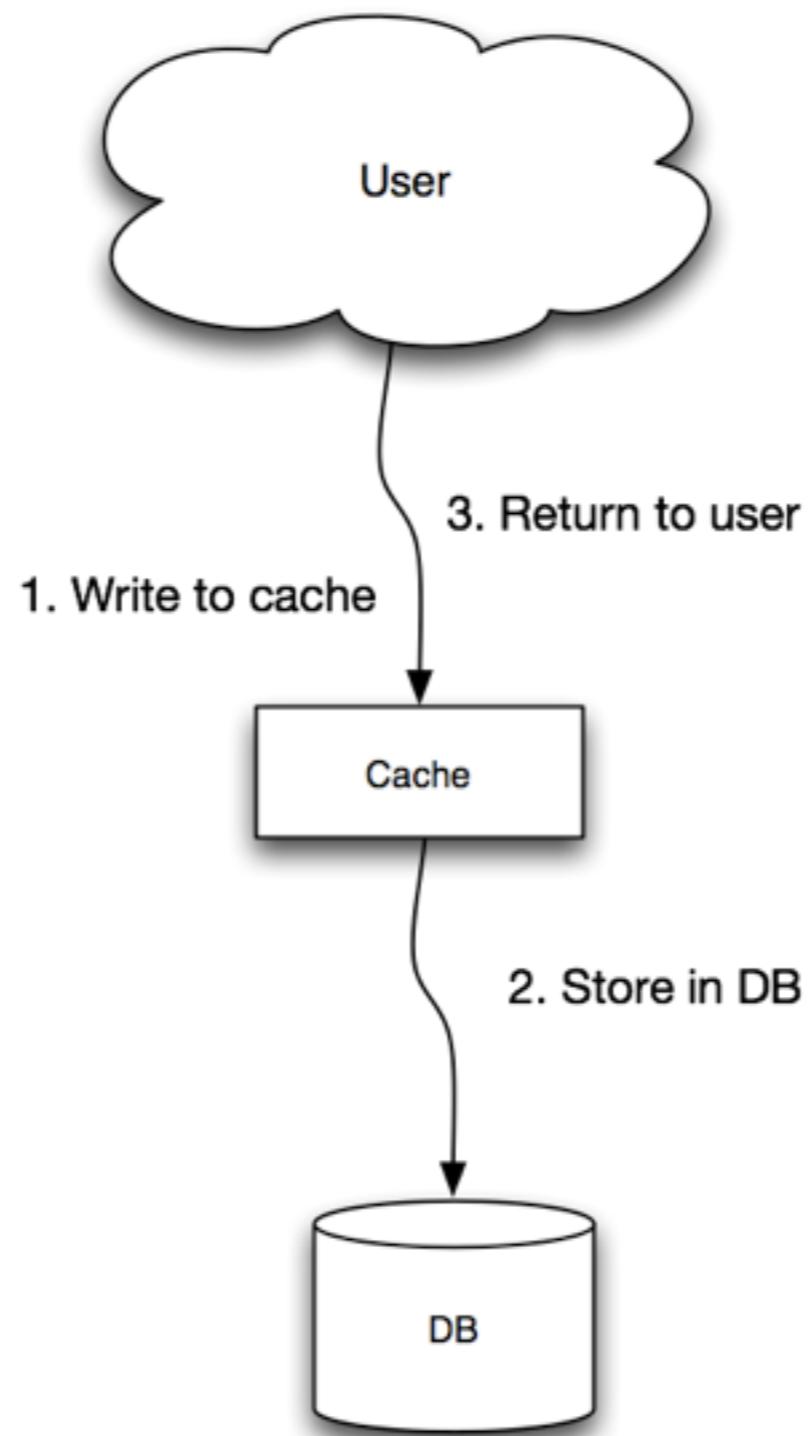
- **Key-Value** databases (Voldemort, Dynomite)
- **Column** databases (Cassandra, Vertica, Sybase IQ)
- **Document** databases (MongoDB, CouchDB)
- **Graph** databases (Neo4J, AllegroGraph)
- **Datastructure** databases (Redis, Hazelcast)

Distributed Caching

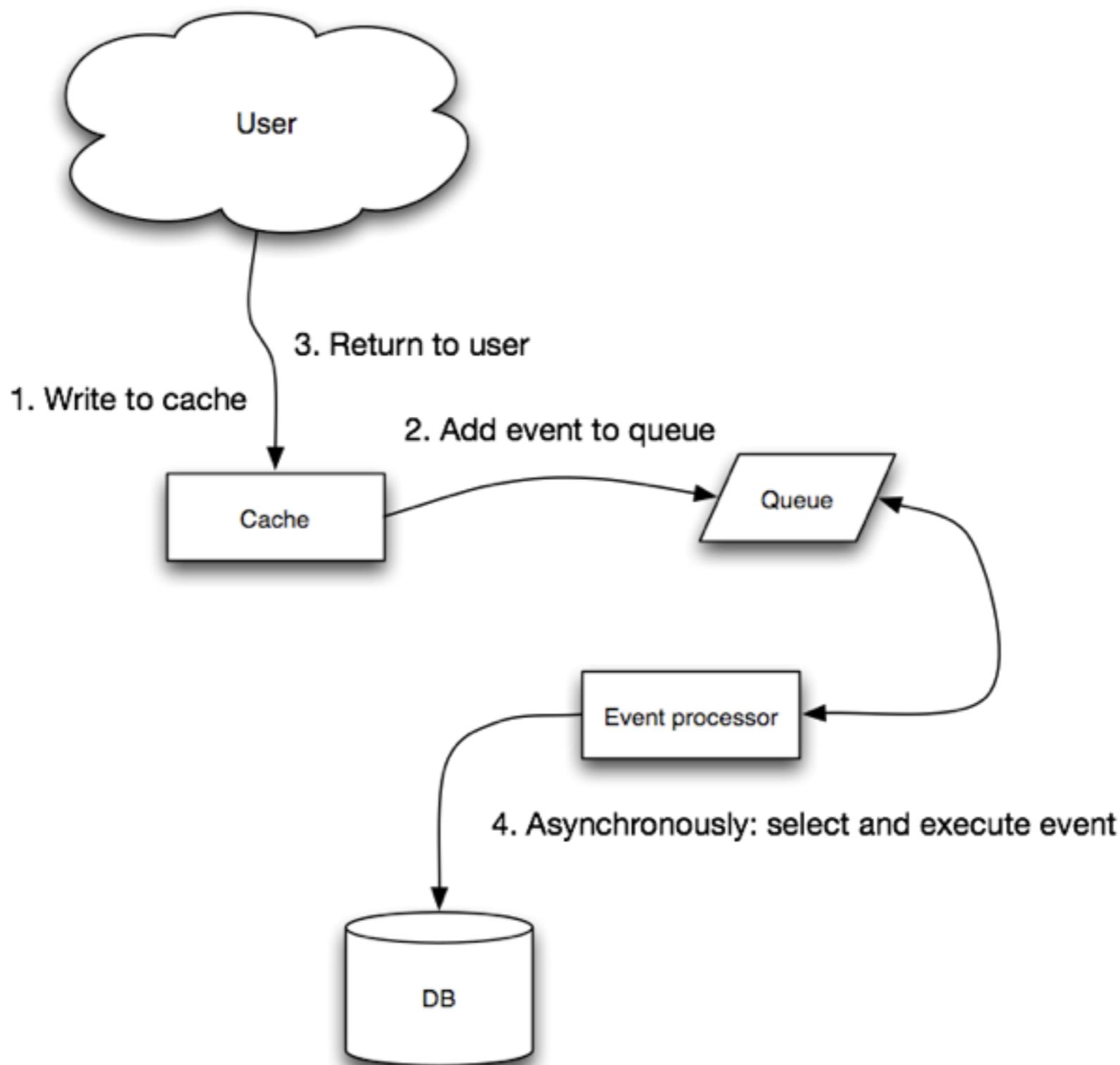
Distributed Caching

- Write-through
- Write-behind
- Eviction Policies
- Replication
- Peer-To-Peer (P2P)

Write-through



Write-behind



Eviction policies

- TTL (time to live)
- Bounded FIFO (first in first out)
- Bounded LIFO (last in first out)
- Explicit cache invalidation

Peer-To-Peer

- Decentralized
- No “special” or “blessed” nodes
- Nodes can join and leave as they please

Distributed Caching Products

- EHCache
- JBoss Cache
- OSCache
- memcached

memcached

- Very fast
- Simple
- Key-Value (string -> binary)
- Clients for most languages
- Distributed
- Not replicated - so I/N chance
for local access in cluster

Data Grids / Clustering

Data Grids/Clustering

Parallel data storage

- Data replication
- Data partitioning
- Continuous availability
- Data invalidation
- Fail-over
- C + P in CAP

Data Grids/Clustering Products

- Coherence
- Terracotta
- GigaSpaces
- GemStone
- Tibco Active Matrix
- Hazelcast

Concurrency

Concurrency

- Shared-State Concurrency
- Message-Passing Concurrency
- Dataflow Concurrency
- Software Transactional Memory

Shared-State Concurrency

Shared-State Concurrency

- Everyone can access anything anytime
- Totally indeterministic
- Introduce determinism at well-defined places...
- ...using locks

Shared-State Concurrency

- Problems with locks:
 - Locks do not compose
 - Taking too few locks
 - Taking too many locks
 - Taking the wrong locks
 - Taking locks in the wrong order
 - Error recovery is hard

Shared-State Concurrency

Please use `java.util.concurrent.*`

- `ConcurrentHashMap`
- `BlockingQueue`
- `ConcurrentQueue`
- `ExecutorService`
- `ReentrantReadWriteLock`
- `CountDownLatch`
- `ParallelArray`
- and much much more..

Message-Passing Concurrency

Actors

- Originates in a 1973 paper by Carl Hewitt
- Implemented in Erlang, Occam, Oz
- Encapsulates state and behavior
- Closer to the definition of OO than classes

Actors

- Share **NOTHING**
- Isolated **lightweight** processes
- Communicates through **messages**
- **Asynchronous** and **non-blocking**
- **No shared state**
 - ... hence, nothing to synchronize.
- Each actor has a **mailbox** (message queue)

Actors

- Easier to reason about
- Raised abstraction level
- Easier to avoid
 - Race conditions
 - Deadlocks
 - Starvation
 - Live locks

Actor libs for the JVM

- Akka (Java/Scala)
- scalaz actors (Scala)
- Lift Actors (Scala)
- Scala Actors (Scala)
- Kilim (Java)
- Jetlang (Java)
- Actor's Guild (Java)
- Actorom (Java)
- FunctionalJava (Java)
- GPars (Groovy)

Dataflow Concurrency

Dataflow Concurrency

- Declarative
- No observable non-determinism
- Data-driven – threads block until data is available
- On-demand, lazy
- No difference between:
 - Concurrent &
 - Sequential code
- Limitations: can't have side-effects

A grayscale photograph of a misty mountain range. The peaks are dark and silhouetted against a lighter sky, with layers of fog or clouds visible between the mountains.

STM:

Software

Transactional Memory

STM: overview

- See the memory (heap and stack) as a transactional dataset
- Similar to a database
 - begin
 - commit
 - abort/rollback
- Transactions are retried automatically upon collision
- Rolls back the memory on abort

STM: overview

- Transactions can nest
- Transactions compose (yipee!!)

```
atomic {
```

```
    ...
```

```
    atomic {
```

```
        ...
```

```
    }
```

```
}
```

STM: restrictions

All operations in scope of
a transaction:

- Need to be **idempotent**

STM libs for the JVM

- Akka (Java/Scala)
- Multiverse (Java)
- Clojure STM (Clojure)
- CCSTM (Scala)
- Deuce STM (Java)

Scalability Patterns: Behavior

Scalability Patterns: Behavior

- Event-Driven Architecture
- Compute Grids
- Load-balancing
- Parallel Computing

Event-Driven Architecture

“Four years from now, ‘mere mortals’ will begin to adopt an event-driven architecture (EDA) for the sort of complex event processing that has been attempted only by software gurus [until now]”

--Roy Schulte (Gartner), 2003

Event-Driven Architecture

- Domain Events
- Event Sourcing
- Command and Query Responsibility Segregation (CQRS) pattern
- Event Stream Processing
- Messaging
- Enterprise Service Bus
- Actors
- Enterprise Integration Architecture (EIA)

Domain Events

“It's really become clear to me in the last couple of years that we need a new building block and that is the Domain Events”

-- Eric Evans, 2009

Domain Events

“Domain Events represent the state of entities at a given time when an important event occurred and decouple subsystems with event streams. Domain Events give us clearer, more expressive models in those cases.”

-- Eric Evans, 2009

Domain Events

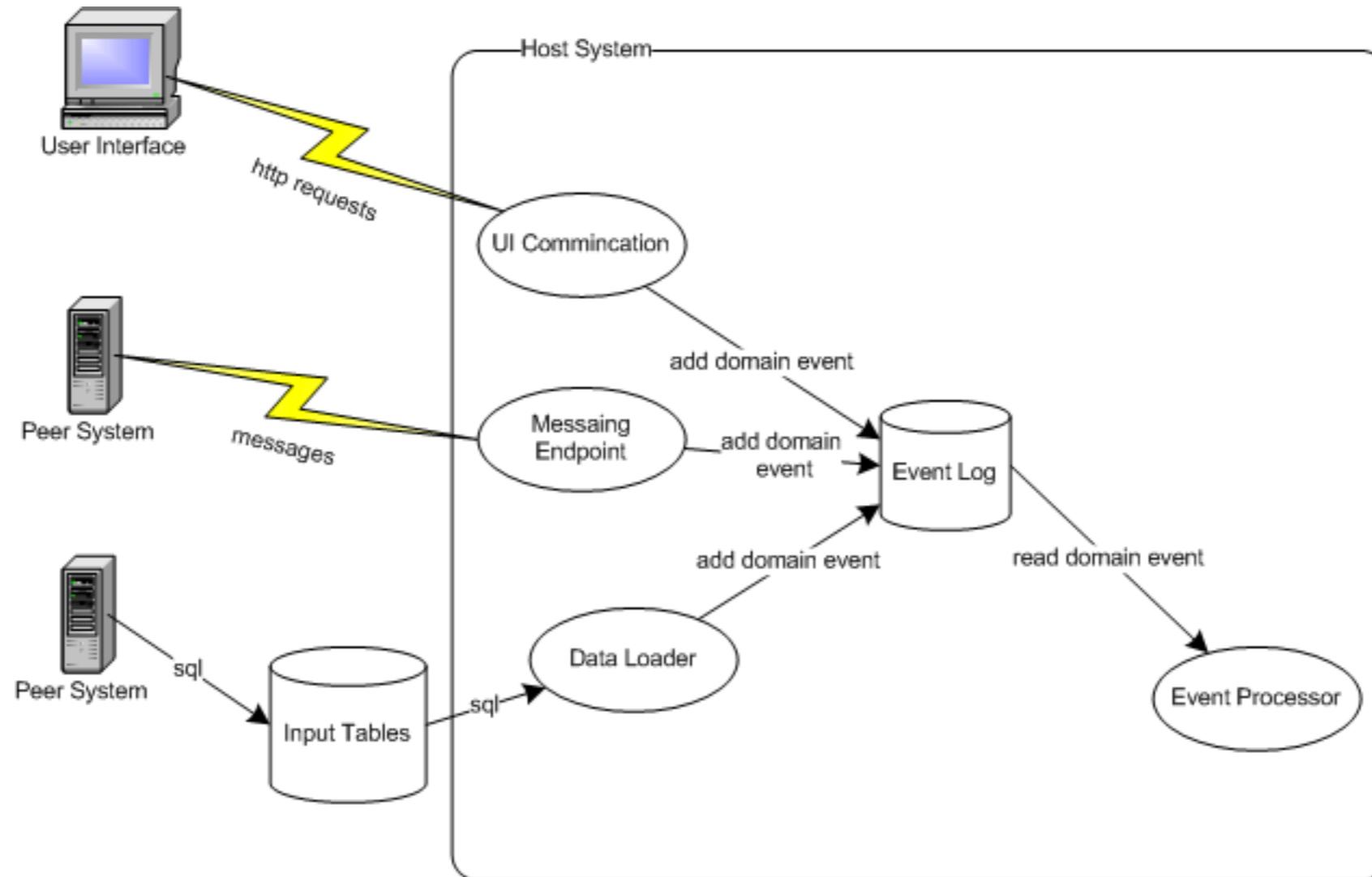
“State transitions are an important part of our problem space and should be modeled within our domain.”

-- Greg Young, 2008

Event Sourcing

- Every state change is materialized in an **Event**
- All Events are sent to an **EventProcessor**
- EventProcessor stores all events in an **Event Log**
- System can be reset and **Event Log** replayed
- No need for ORM, just persist the Events
- Many different **EventListeners** can be added to EventProcessor (or listen directly on the Event log)

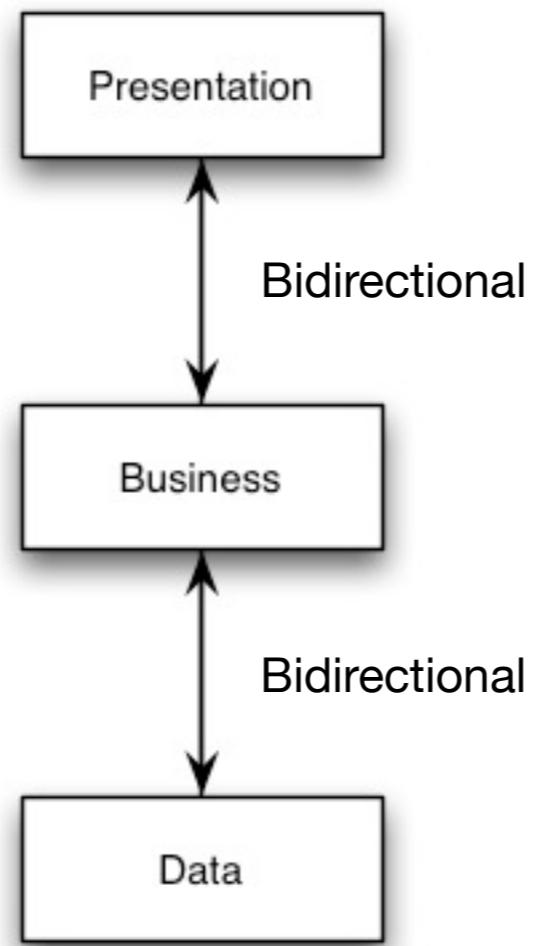
Event Sourcing

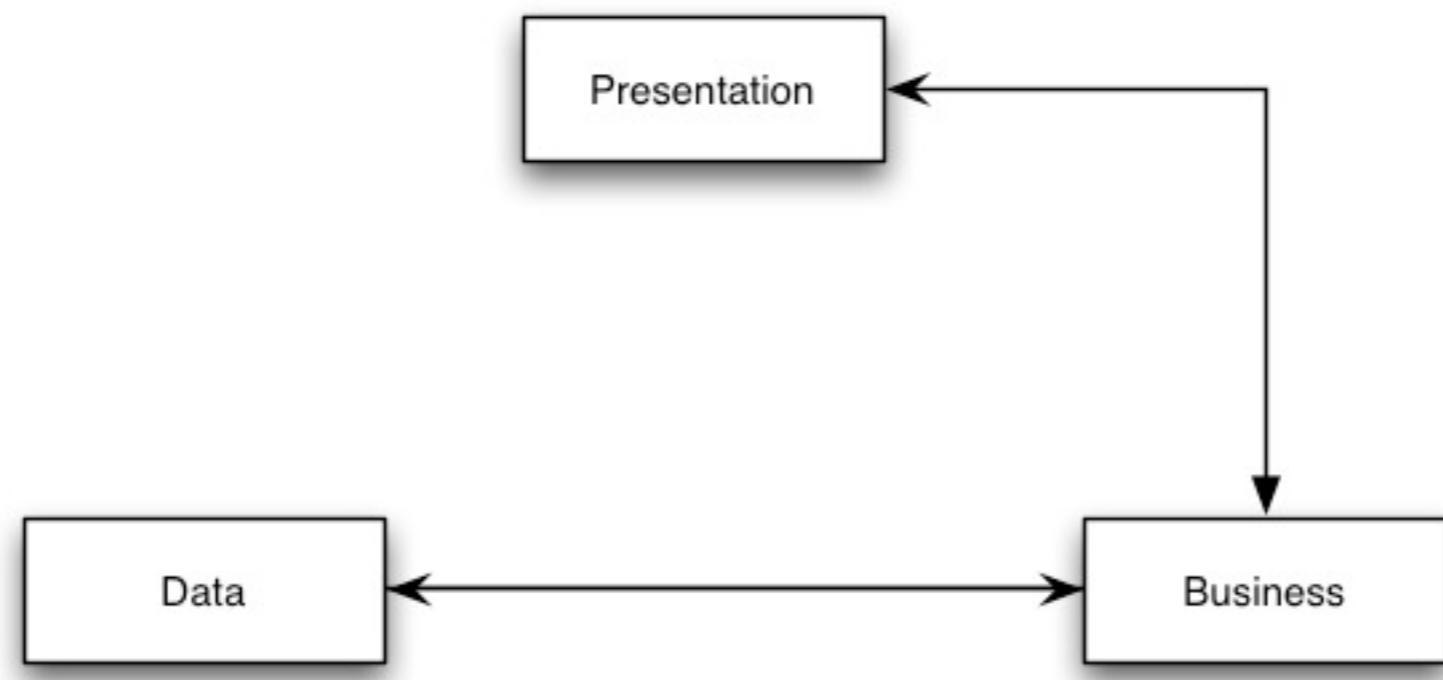


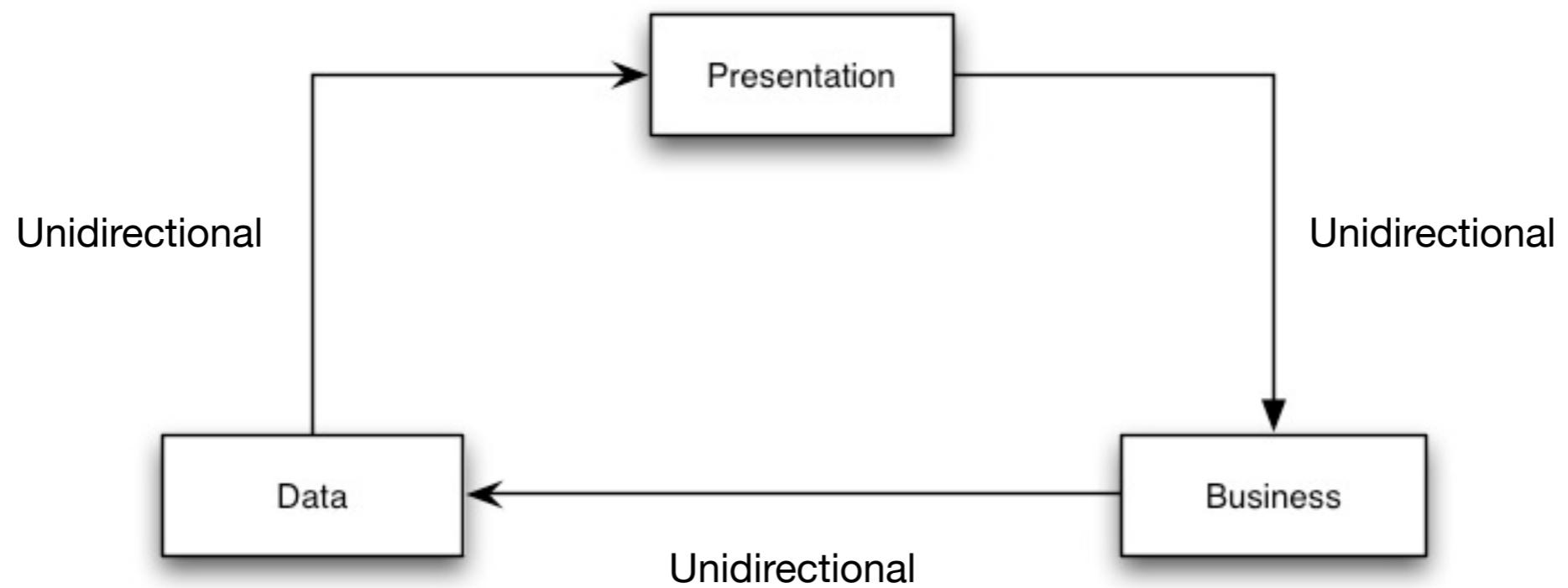
Command and Query Responsibility Segregation (CQRS) pattern

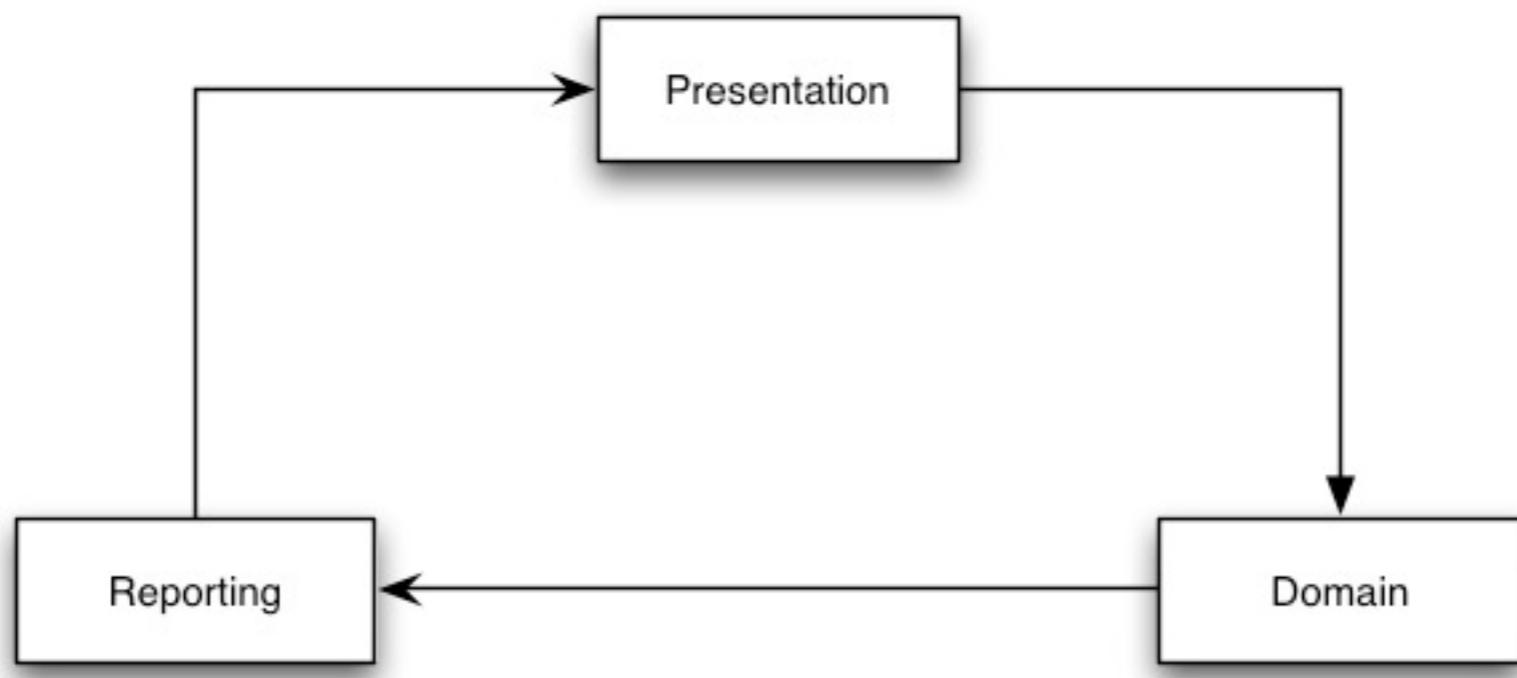
“A single model cannot be appropriate for reporting, searching and transactional behavior.”

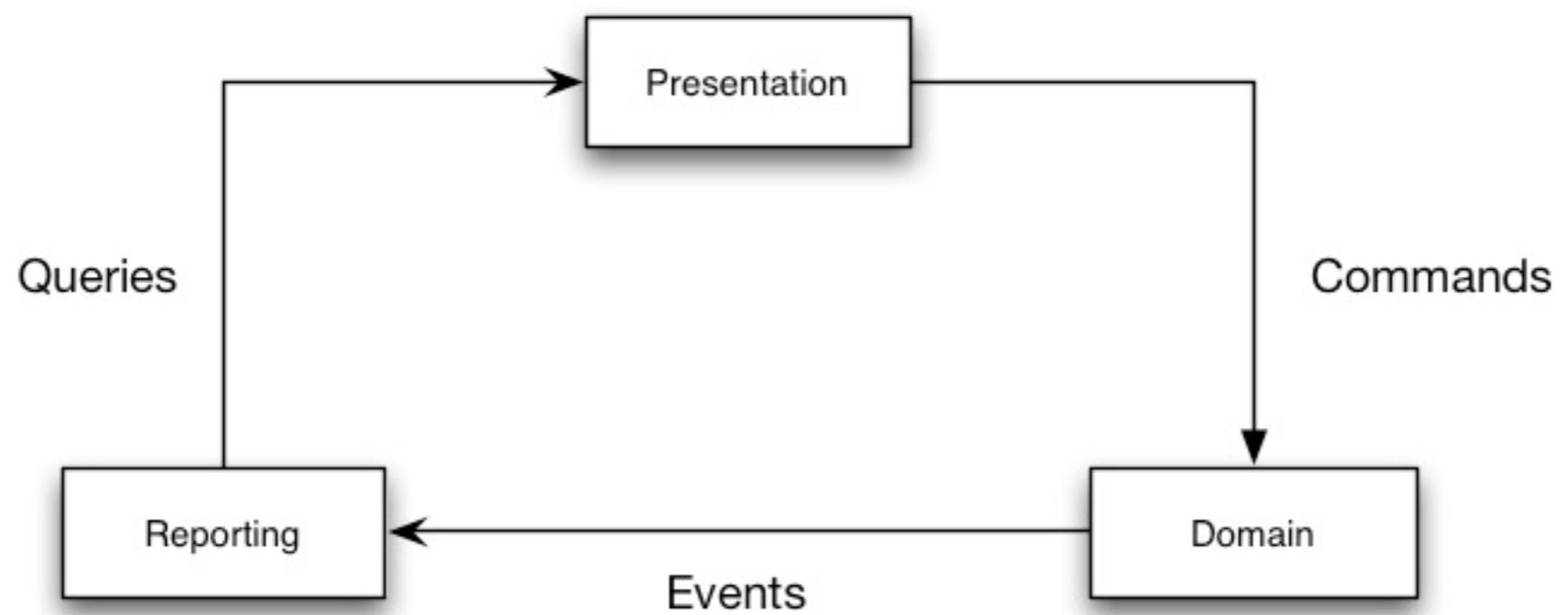
-- Greg Young, 2008

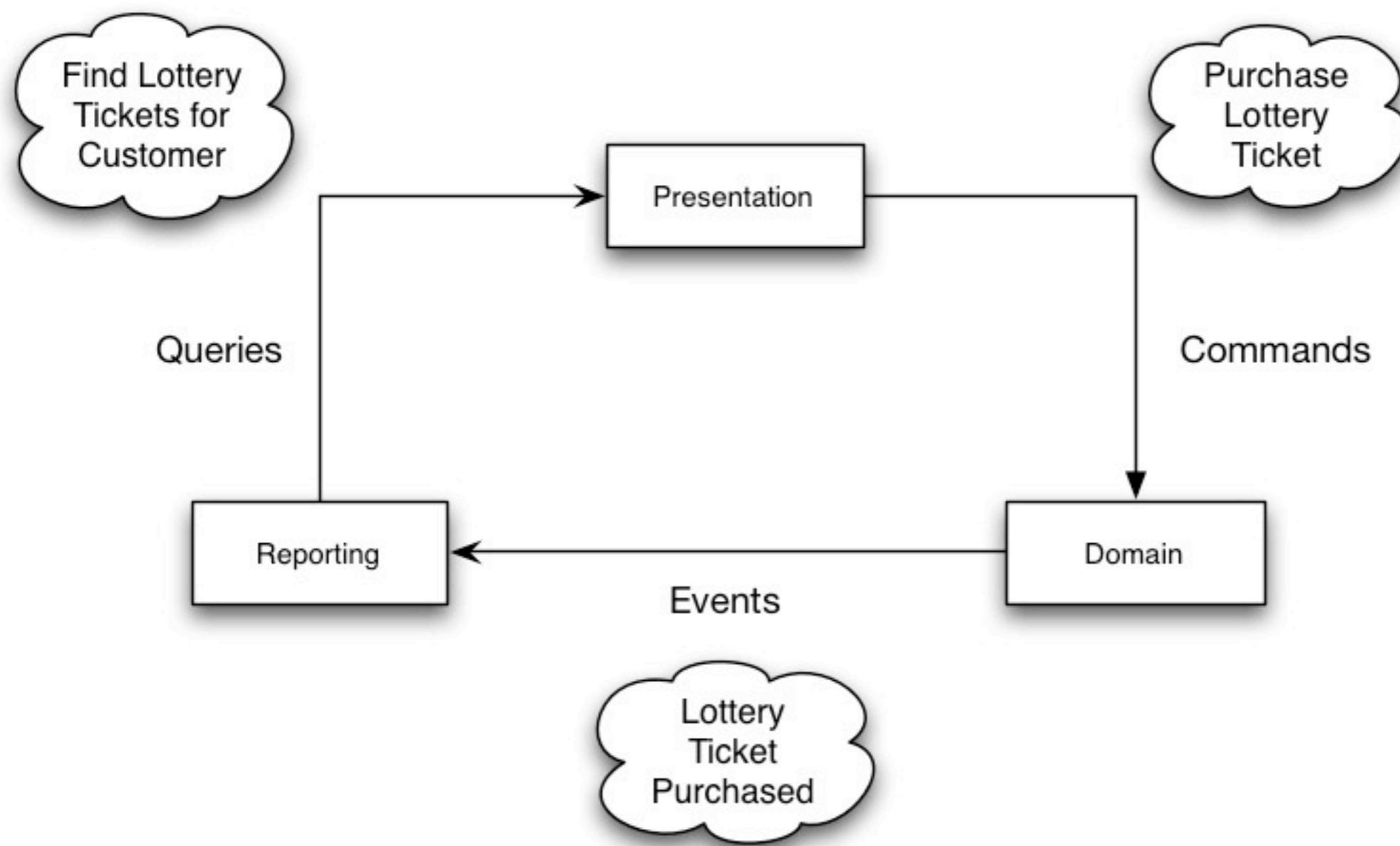










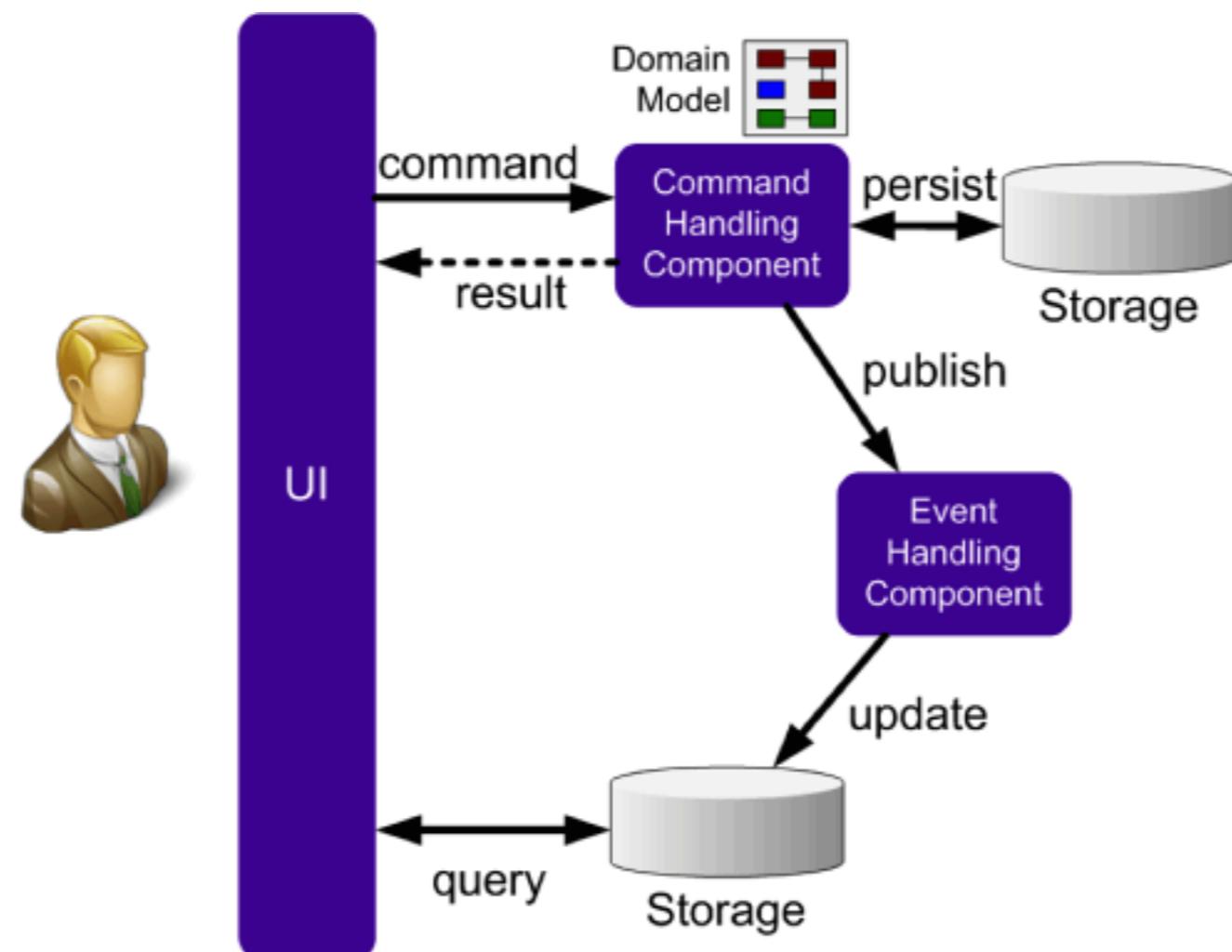


CQRS

in a nutshell

- All state changes are represented by Domain Events
- Aggregate roots **receive Commands** and **publish Events**
- *Reporting* (query database) is **updated** as a **result** of the published *Events*
- All *Queries* from *Presentation* go **directly** to *Reporting* and the *Domain* is **not involved**

CQRS

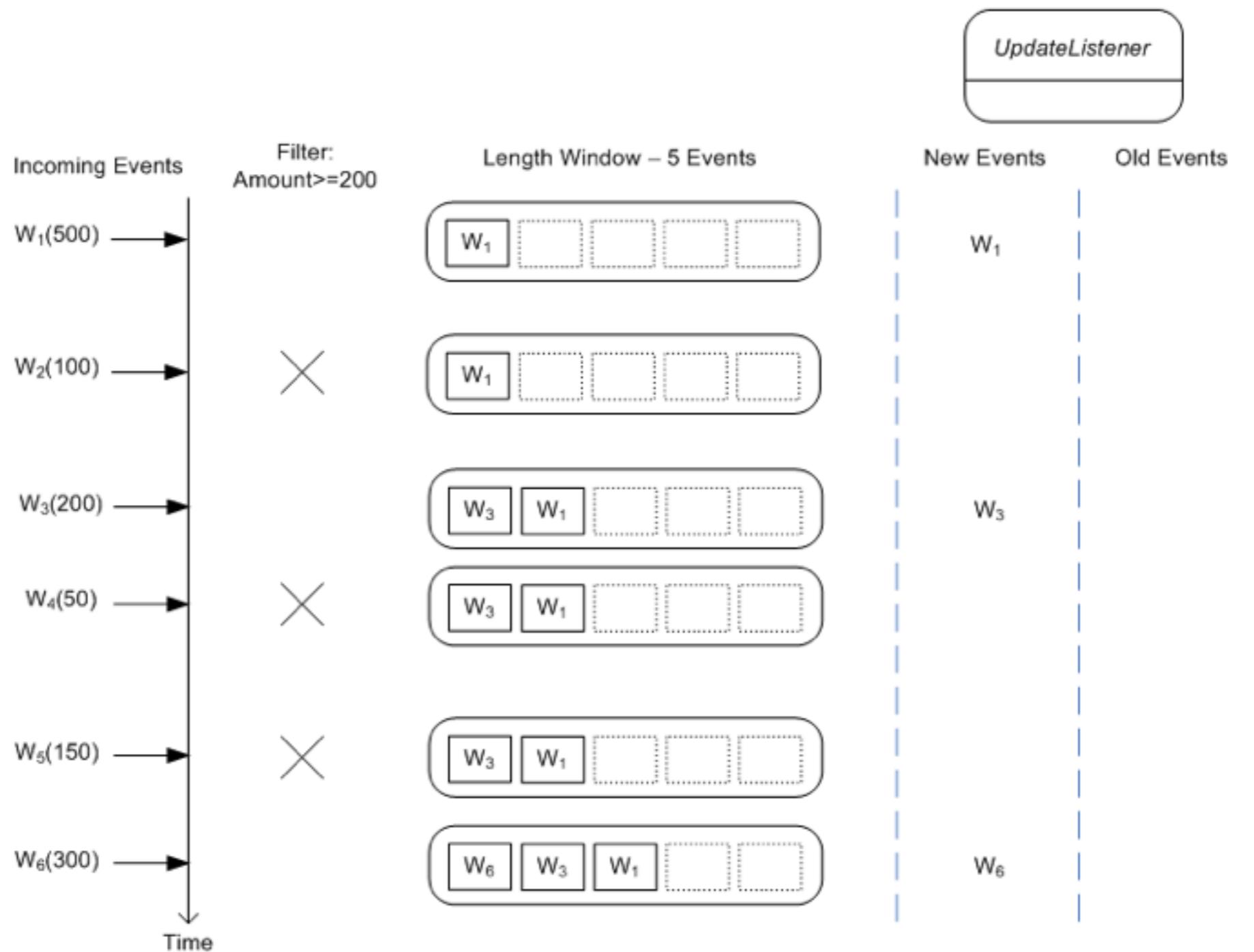


Copyright by Axis Framework

CQRS: Benefits

- Fully encapsulated domain that only exposes behavior
- Queries do not use the domain model
- No object-relational impedance mismatch
- Bullet-proof auditing and historical tracing
- Easy integration with external systems
- Performance and scalability

Event Stream Processing



select * from
Withdrawal(amount>=200).win:length(5)

Event Stream Processing

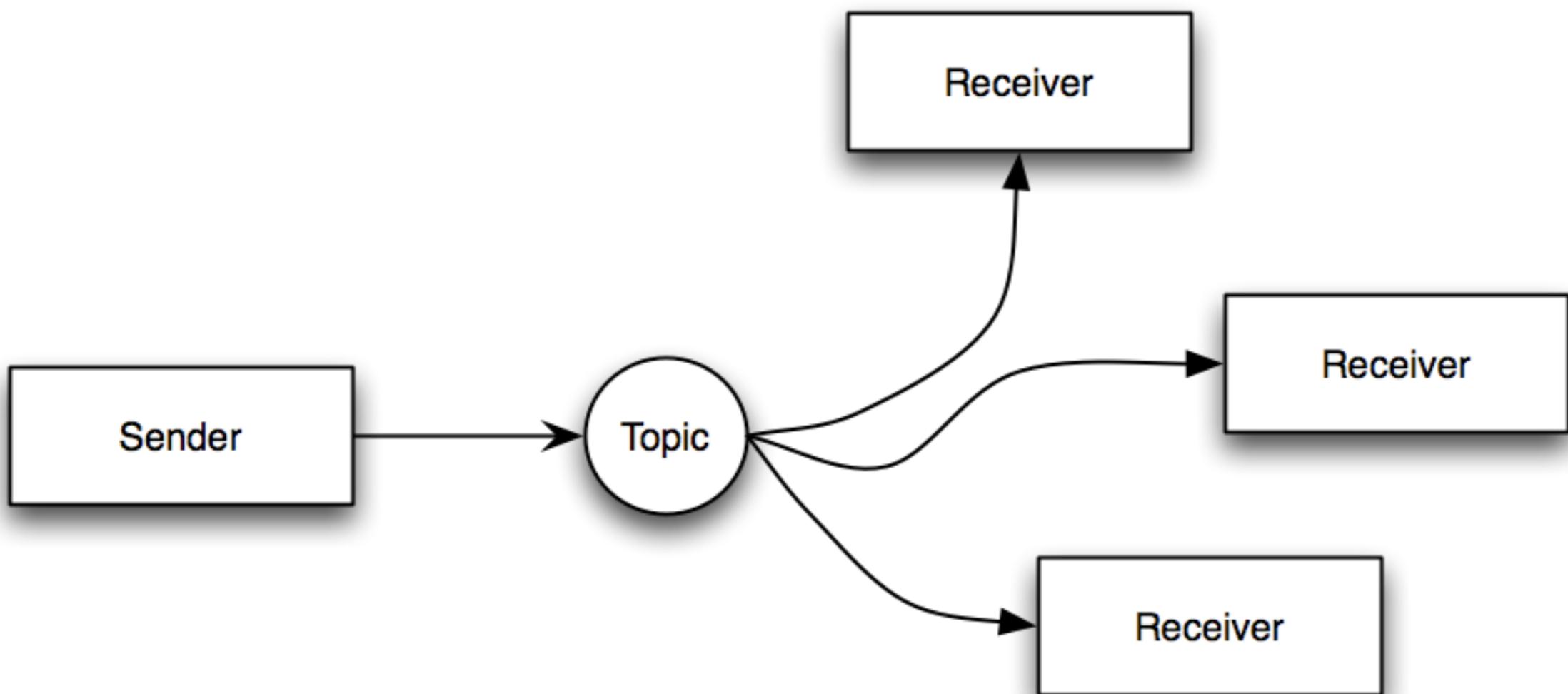
Products

- Esper (Open Source)
- StreamBase
- RuleCast

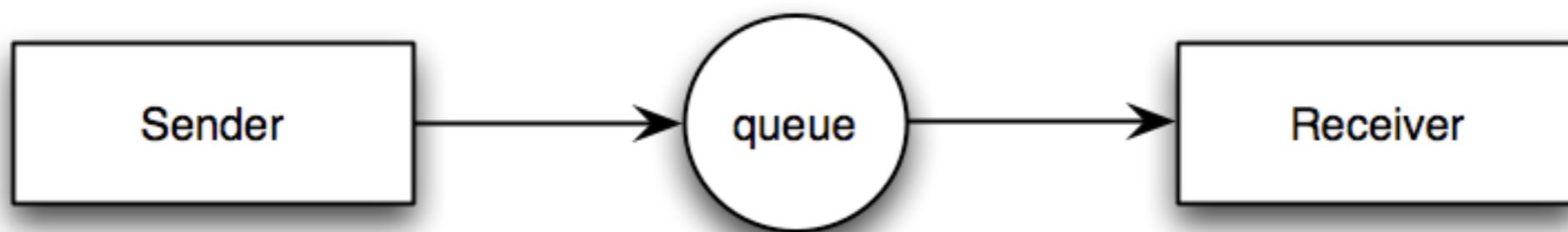
Messaging

- Publish-Subscribe
- Point-to-Point
- Store-forward
- Request-Reply

Publish-Subscribe

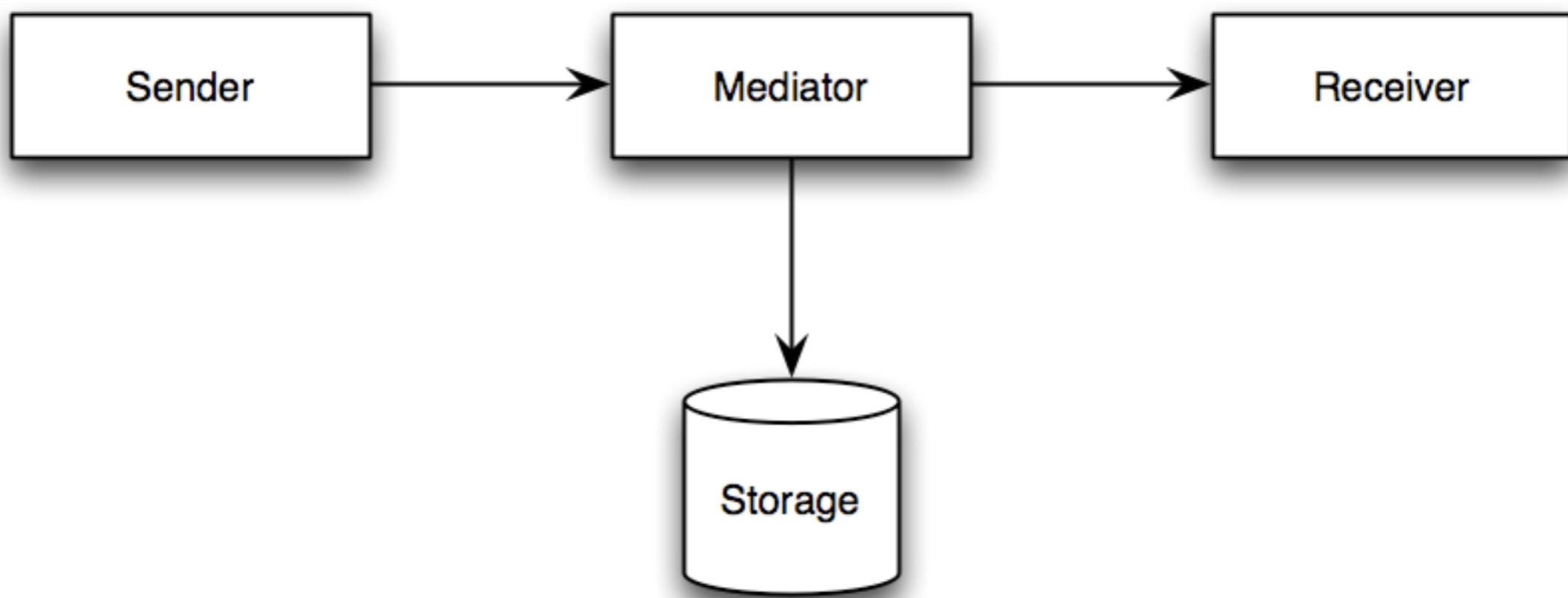


Point-to-Point



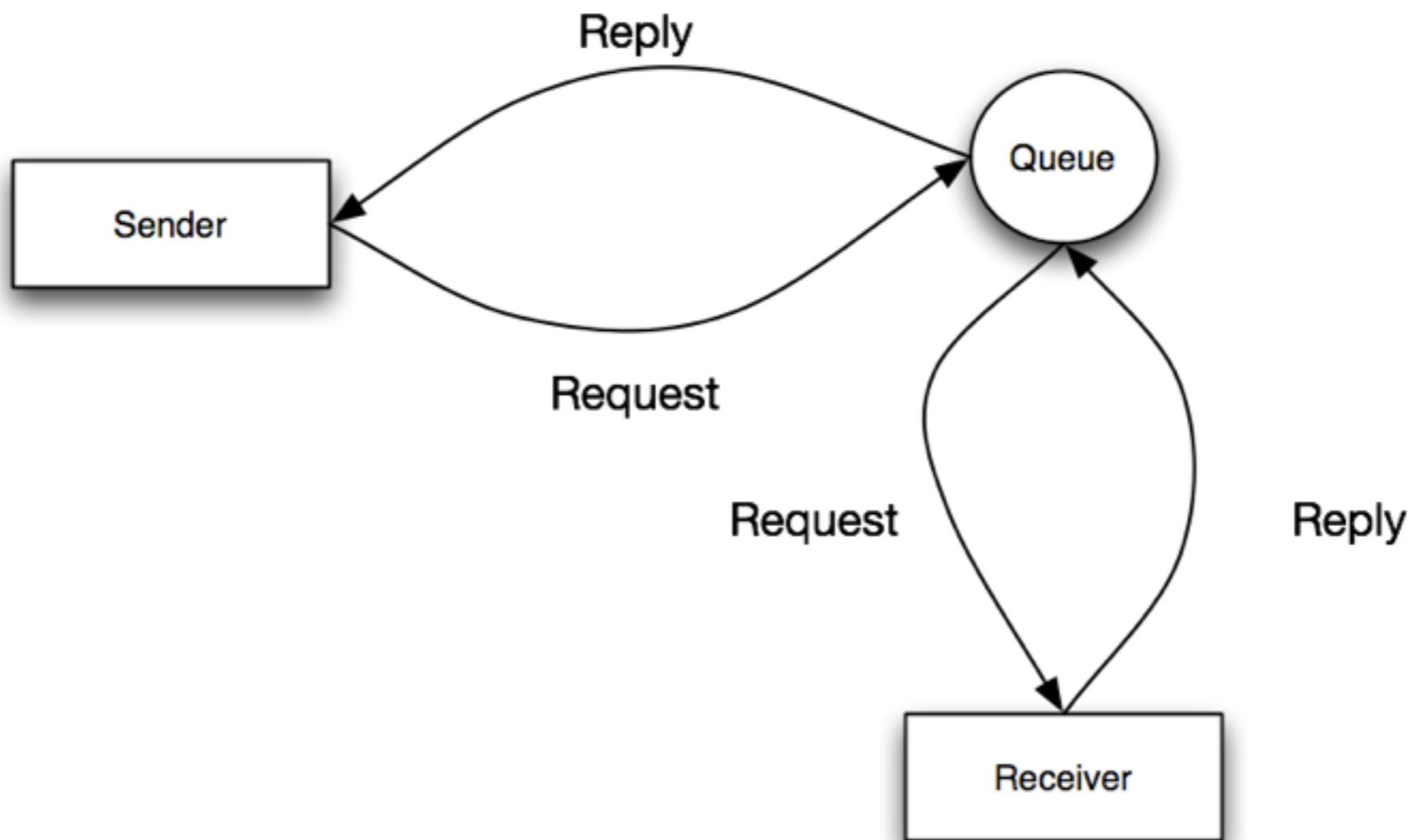
Store-Forward

Durability, event log, auditing etc.



Request-Reply

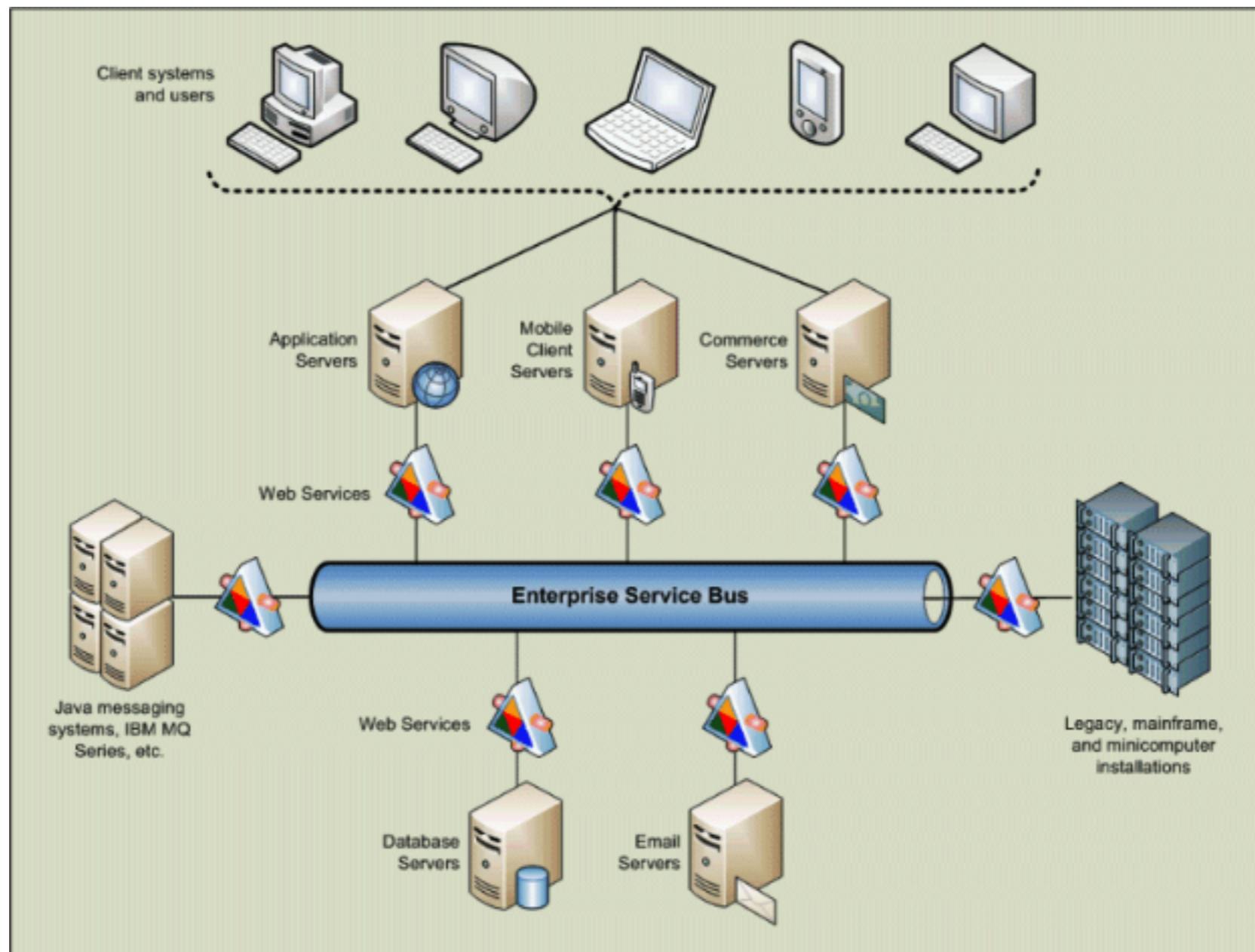
F.e. AMQP's 'replyTo' header



Messaging

- Standards:
 - AMQP
 - JMS
- Products:
 - RabbitMQ (AMQP)
 - ActiveMQ (JMS)
 - Tibco
 - MQSeries
 - etc

ESB



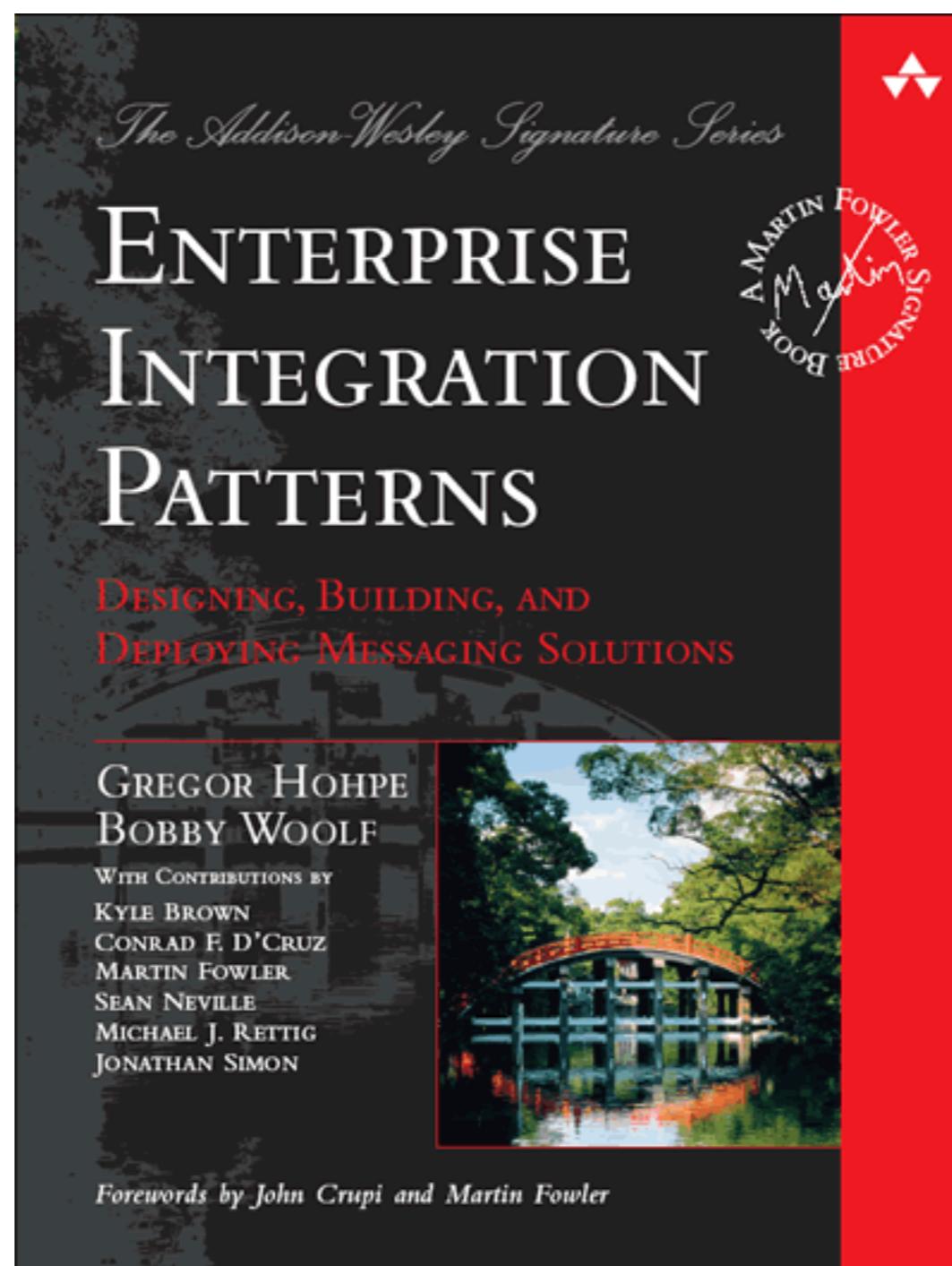
ESB products

- ServiceMix (Open Source)
- Mule (Open Source)
- Open ESB (Open Source)
- Sonic ESB
- WebSphere ESB
- Oracle ESB
- Tibco
- BizTalk Server

Actors

- Fire-forget
 - Async send
- Fire-And-Receive-Eventually
 - Async send + wait on Future for reply

Enterprise Integration Patterns



Enterprise Integration Patterns

Apache Camel

- More than 80 endpoints
- XML (Spring) DSL
- Scala DSL

Compute Grids

Compute Grids

Parallel execution

- Divide and conquer
 1. Split up job in independent tasks
 2. Execute tasks in parallel
 3. Aggregate and return result
- MapReduce - Master/Worker

Compute Grids

Parallel execution

- Automatic provisioning
- Load balancing
- Fail-over
- Topology resolution

Compute Grids

Products

- Platform
- DataSynapse
- Google MapReduce
- Hadoop
- GigaSpaces
- GridGain



Load balancing

Load balancing

- Random allocation
- Round robin allocation
- Weighted allocation
- Dynamic load balancing
 - Least connections
 - Least server CPU
 - etc.

Load balancing

- DNS Round Robin (simplest)
 - Ask DNS for IP for host
 - Get a new IP every time
- Reverse Proxy (better)
- Hardware Load Balancing

Load balancing products

- Reverse Proxies:
 - Apache mod_proxy (OSS)
 - HAProxy (OSS)
 - Squid (OSS)
 - Nginx (OSS)
- Hardware Load Balancers:
 - BIG-IP
 - Cisco

Parallel Computing

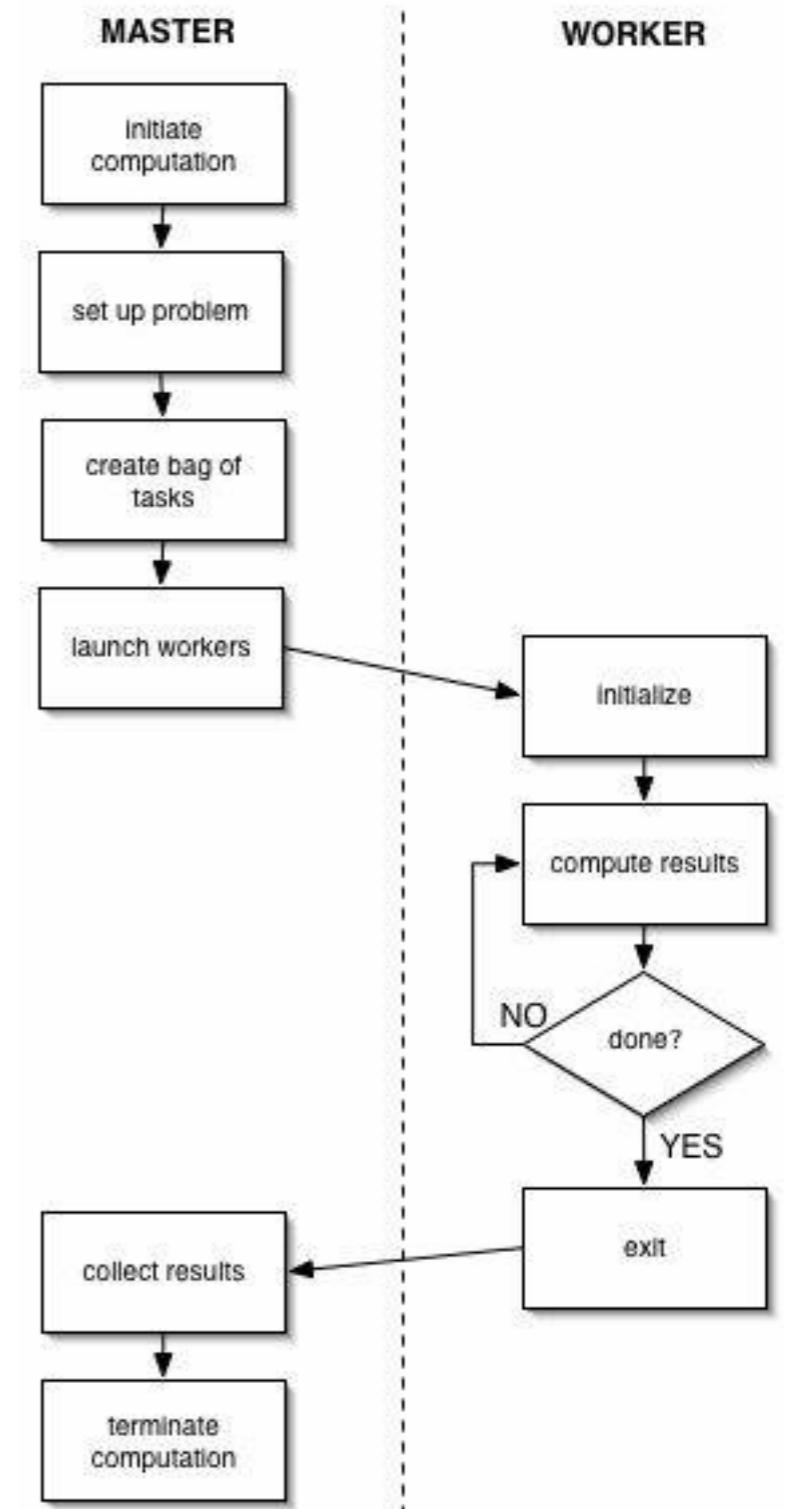
Parallel Computing

- SPMD Pattern
- Master/Worker Pattern
- Loop Parallelism Pattern
- Fork/Join Pattern
- MapReduce Pattern
- UE: Unit of Execution
 - Process
 - Thread
 - Coroutine
 - Actor

SPMD Pattern

- Single Program Multiple Data
- Very generic pattern, used in many other patterns
- Use a single program for all the UEs
- Use the UE's ID to select different pathways through the program. F.e:
 - Branching on ID
 - Use ID in loop index to split loops
- Keep interactions between UEs explicit

Master/Worker



Master/Worker

- Good scalability
- Automatic load-balancing
- How to detect termination?
 - Bag of tasks is empty
 - Poison pill
- If we bottleneck on single queue?
 - Use multiple work queues
 - Work stealing
- What about fault tolerance?
 - Use “in-progress” queue

Loop Parallelism

- Workflow
 - 1. Find the loops that are bottlenecks
 - 2. Eliminate coupling between loop iterations
 - 3. Parallelize the loop
- If too few iterations to pull its weight
 - Merge loops
 - Coalesce nested loops
- OpenMP
 - `omp parallel for`

What if task creation can't be handled by:

- parallelizing loops (Loop Parallelism)
- putting them on work queues (Master/Worker)

What if task creation can't be handled by:

- parallelizing loops (Loop Parallelism)
- putting them on work queues (Master/Worker)

Enter
Fork/Join

Fork/Join

- Use when relationship between tasks is simple
- Good for recursive data processing
- Can use work-stealing
 - 1. **Fork:** Tasks are dynamically created
 - 2. **Join:** Tasks are later terminated and data aggregated

Fork/Join

- Direct task/UE mapping
 - 1-1 mapping between Task/UE
 - Problem: Dynamic UE creation is expensive
- Indirect task/UE mapping
 - Pool the UE
 - Control (constrain) the resource allocation
 - Automatic load balancing

Fork/Join

Java 7 ParallelArray (Fork/Join DSL)

Fork/Join

Java 7 ParallelArray (Fork/Join DSL)

```
ParallelArray students =  
    new ParallelArray(fjPool, data);  
  
double bestGpa = students.withFilter(isSenior)  
    .withMapping(selectGpa)  
    .max();
```

MapReduce

- Origin from Google paper 2004
- Used internally @ Google
- Variation of Fork/Join
- Work divided upfront not dynamically
- Usually distributed
- Normally used for massive data crunching

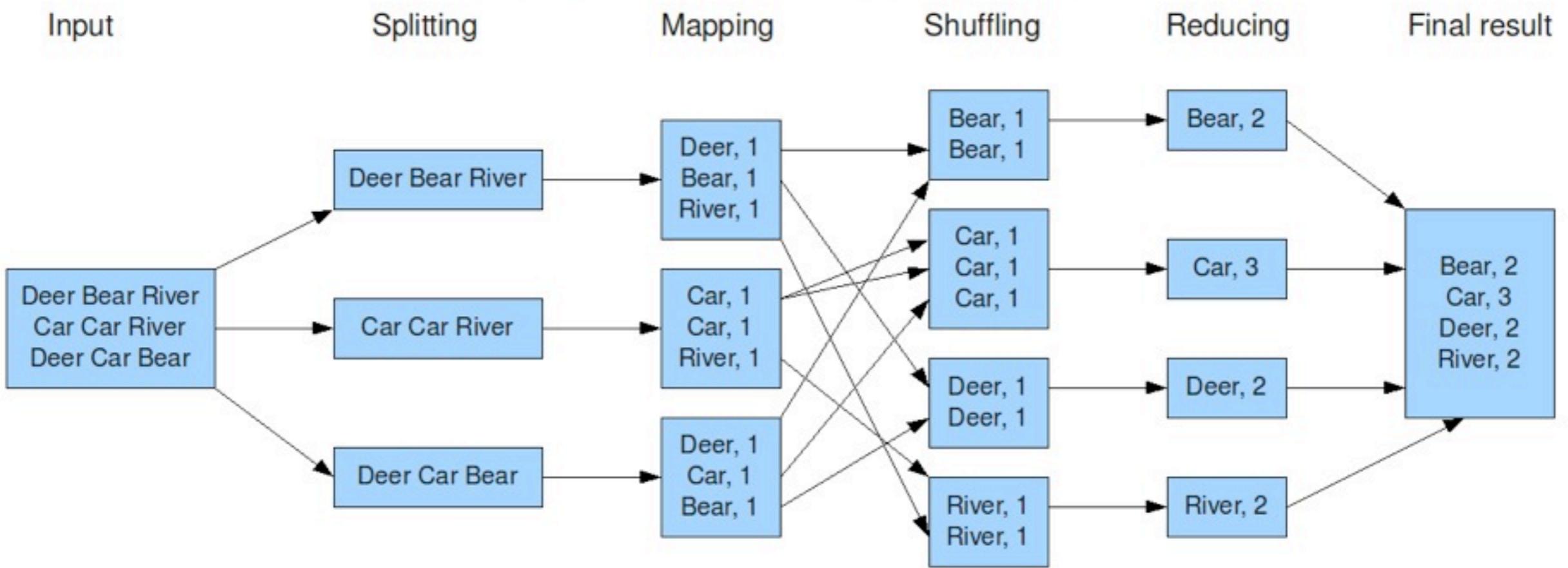
MapReduce

Products

- Hadoop (OSS), used @ Yahoo
- Amazon Elastic MapReduce
- Many NOSQL DBs utilizes it for searching/querying

MapReduce

The overall MapReduce word count process



Parallel Computing

products

- MPI
- OpenMP
- JSR166 Fork/Join
- `java.util.concurrent`
 - ExecutorService, BlockingQueue etc.
- ProActive Parallel Suite
- CommonJ WorkManager (JEE)

Stability Patterns

Stability Patterns

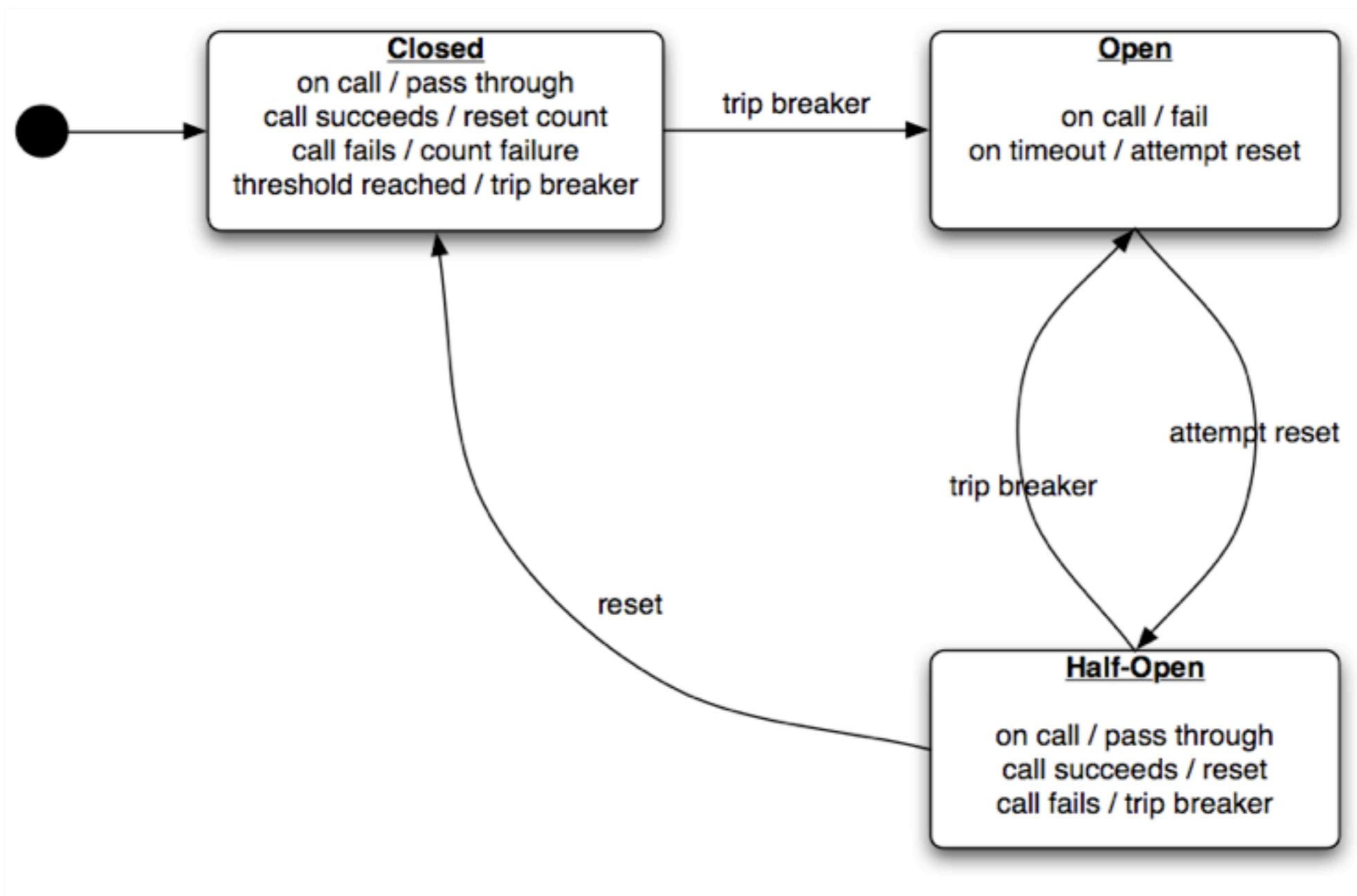
- Timeouts
- Circuit Breaker
- Let-it-crash
- Fail fast
- Bulkheads
- Steady State
- Throttling

Timeouts

Always use timeouts (if possible):

- Thread.`wait(timeout)`
- reentrantLock.`tryLock`
- blockingQueue.`poll(timeout, timeUnit)`/
`offer(..)`
- futureTask.`get(timeout, timeUnit)`
- socket.`setSoTimeOut(timeout)`
- etc.

Circuit Breaker

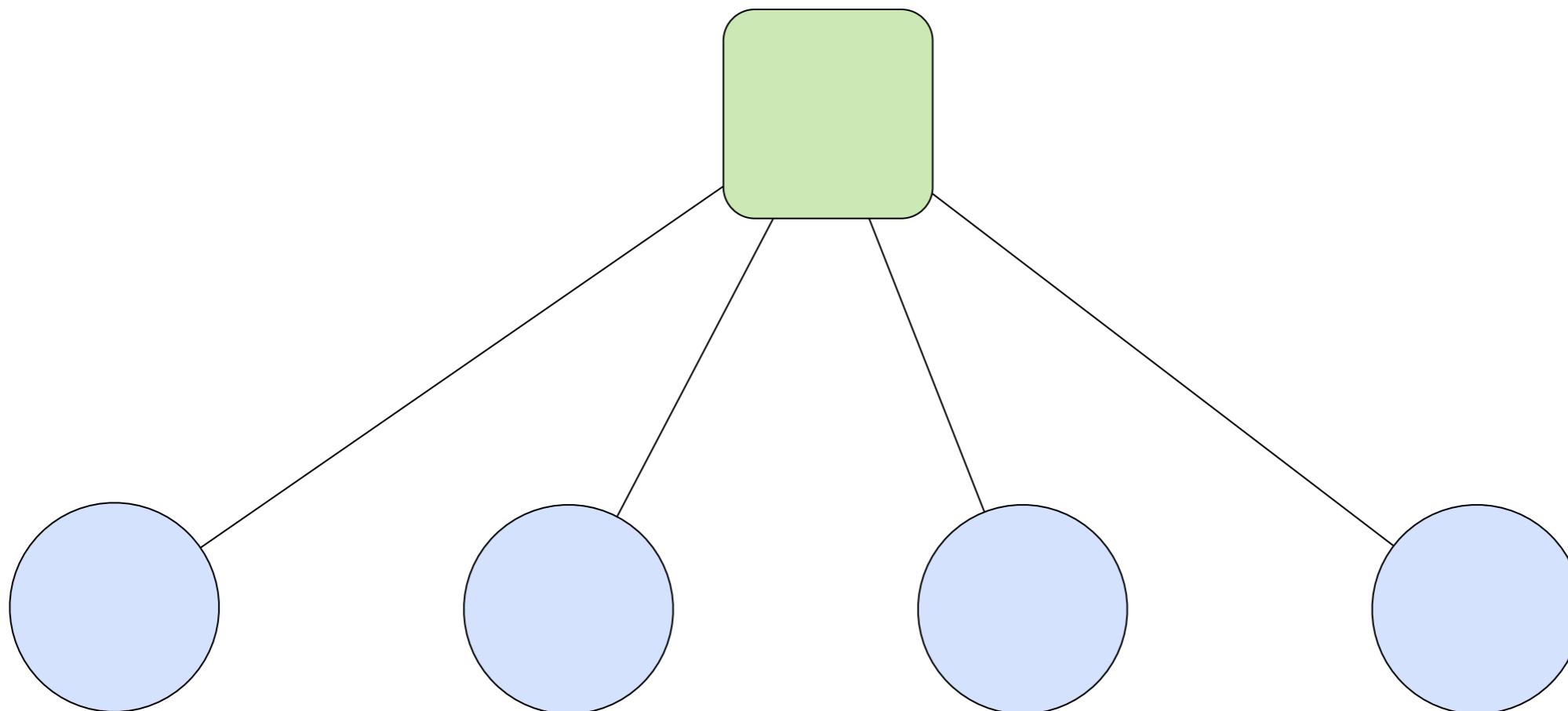


Let it crash

- Embrace failure as a natural state in the life-cycle of the application
- Instead of trying to prevent it; manage it
- Process supervision
- Supervisor hierarchies (from Erlang)

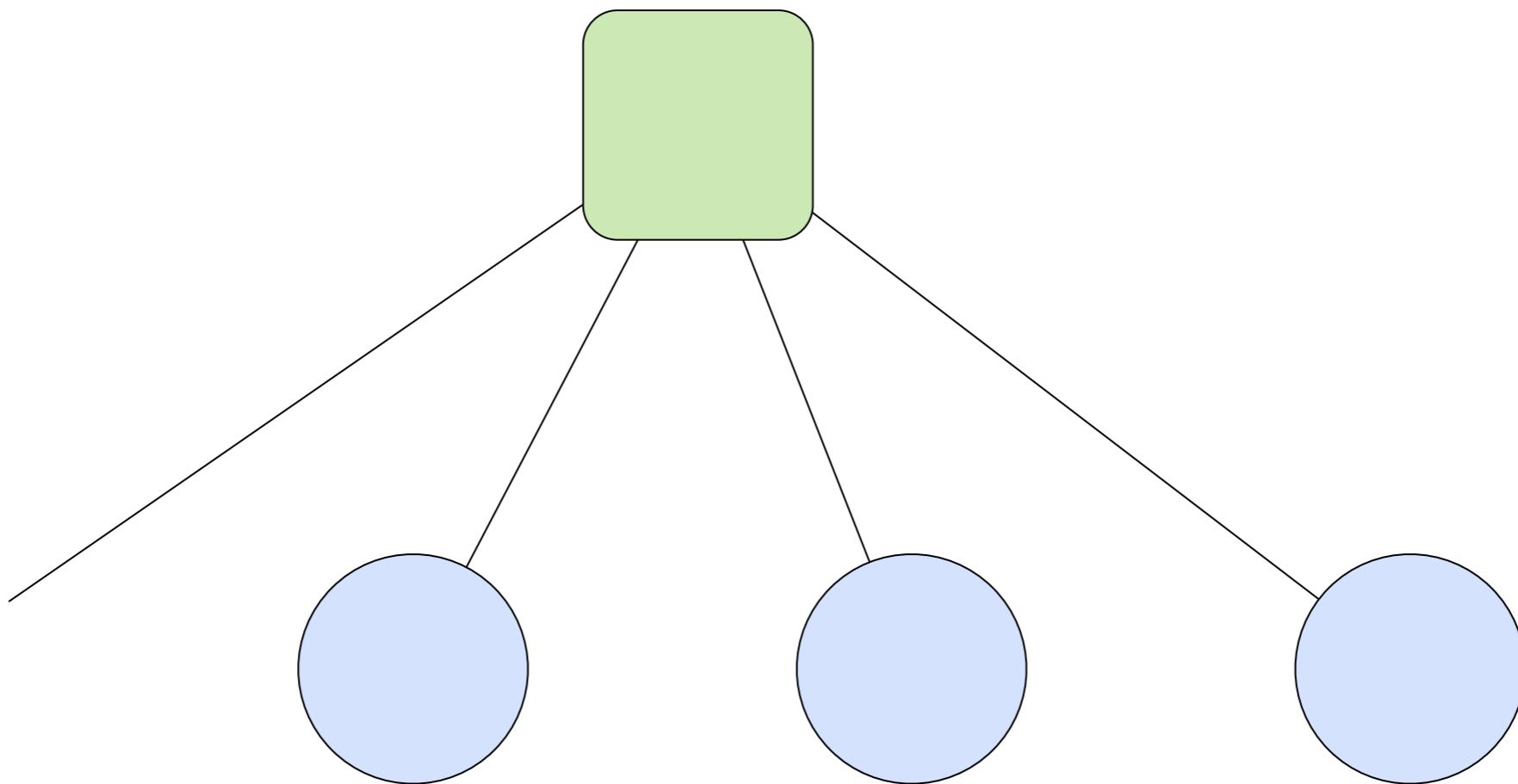
Restart Strategy

OneForOne



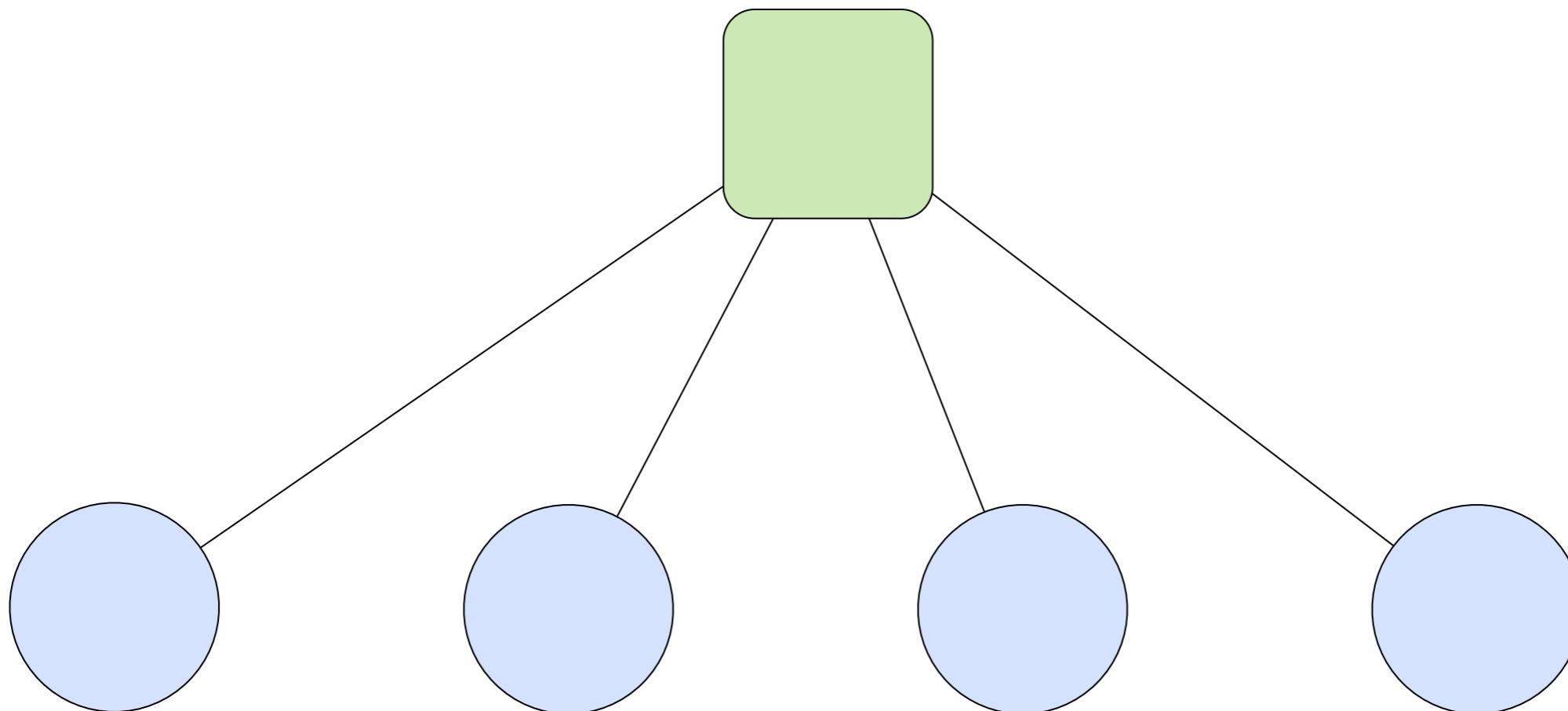
Restart Strategy

OneForOne



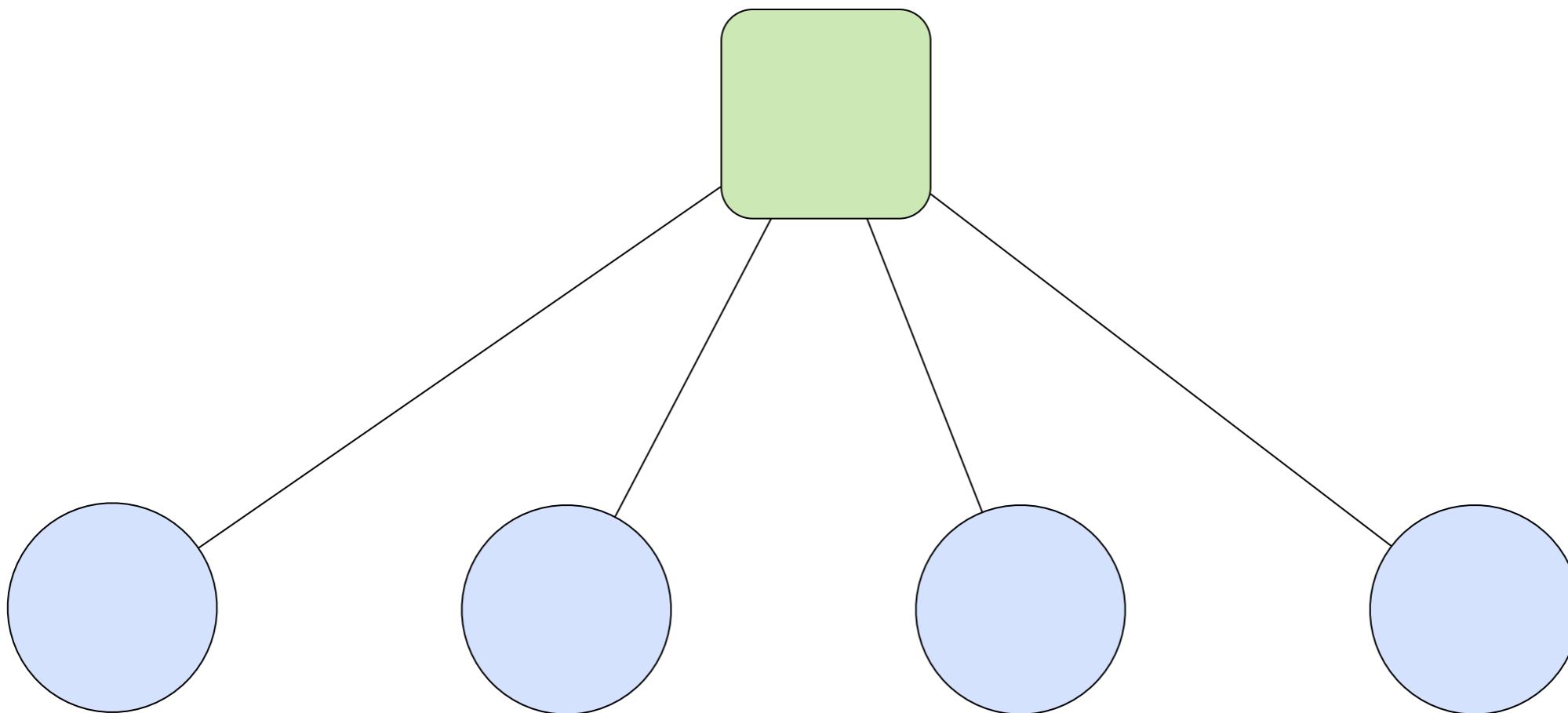
Restart Strategy

OneForOne



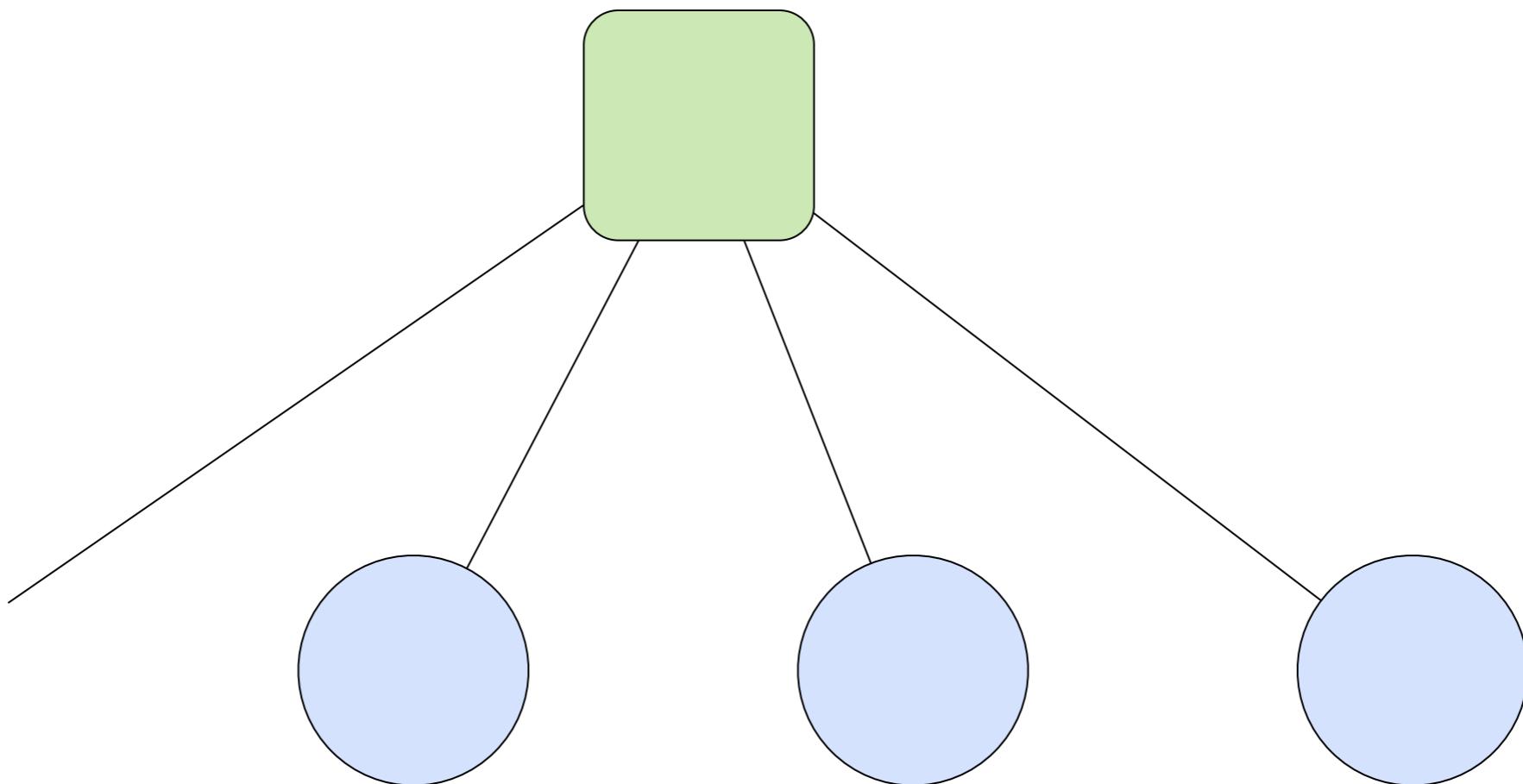
Restart Strategy

AllForOne



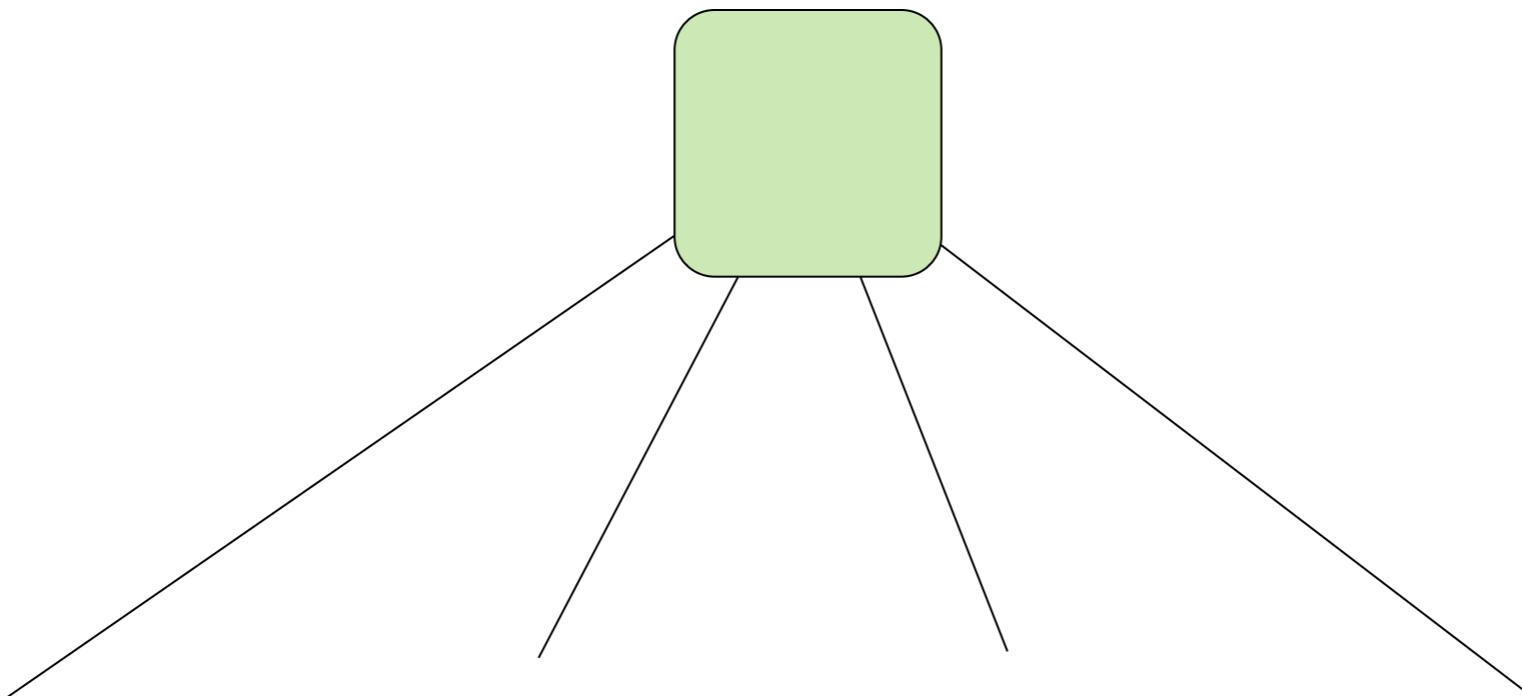
Restart Strategy

AllForOne



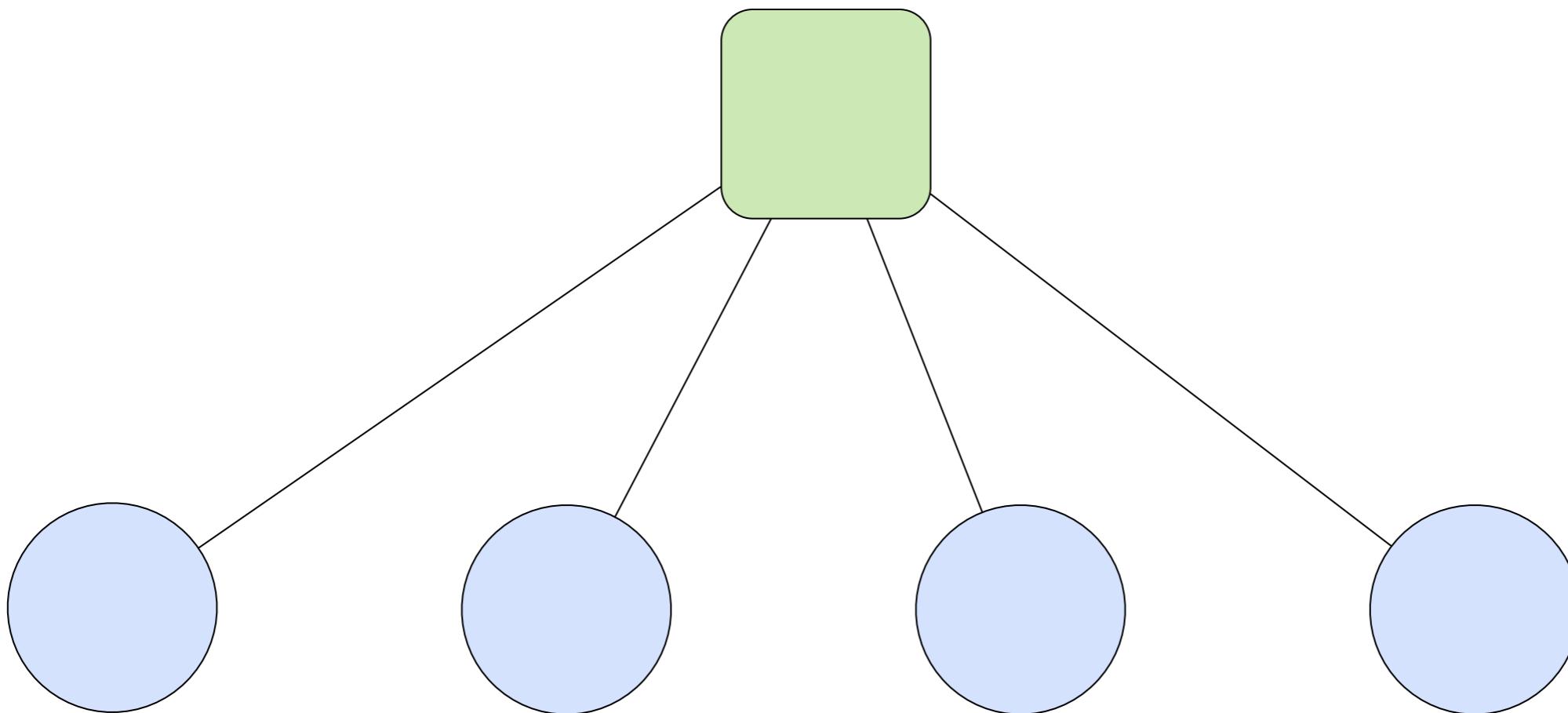
Restart Strategy

AllForOne

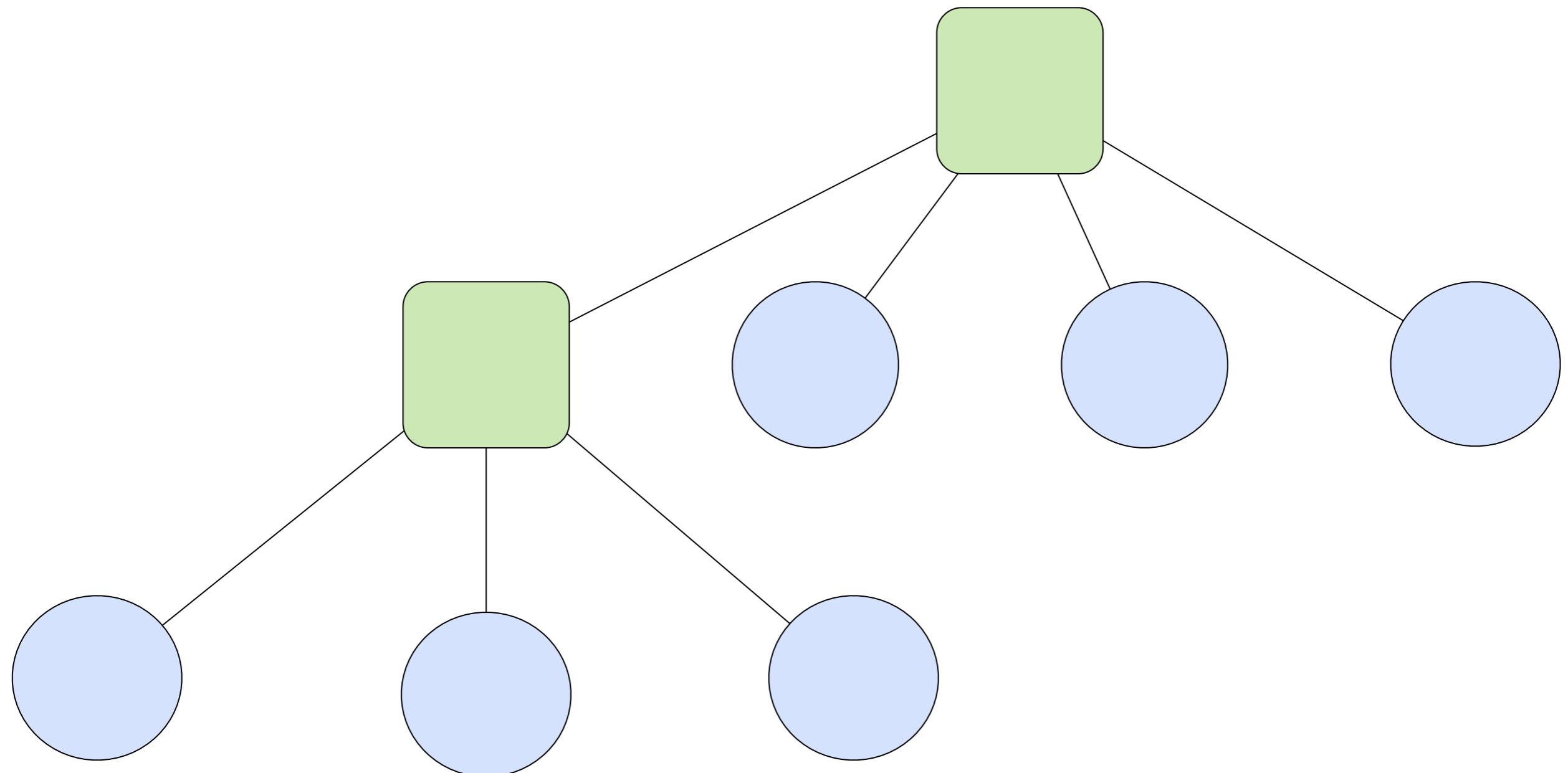


Restart Strategy

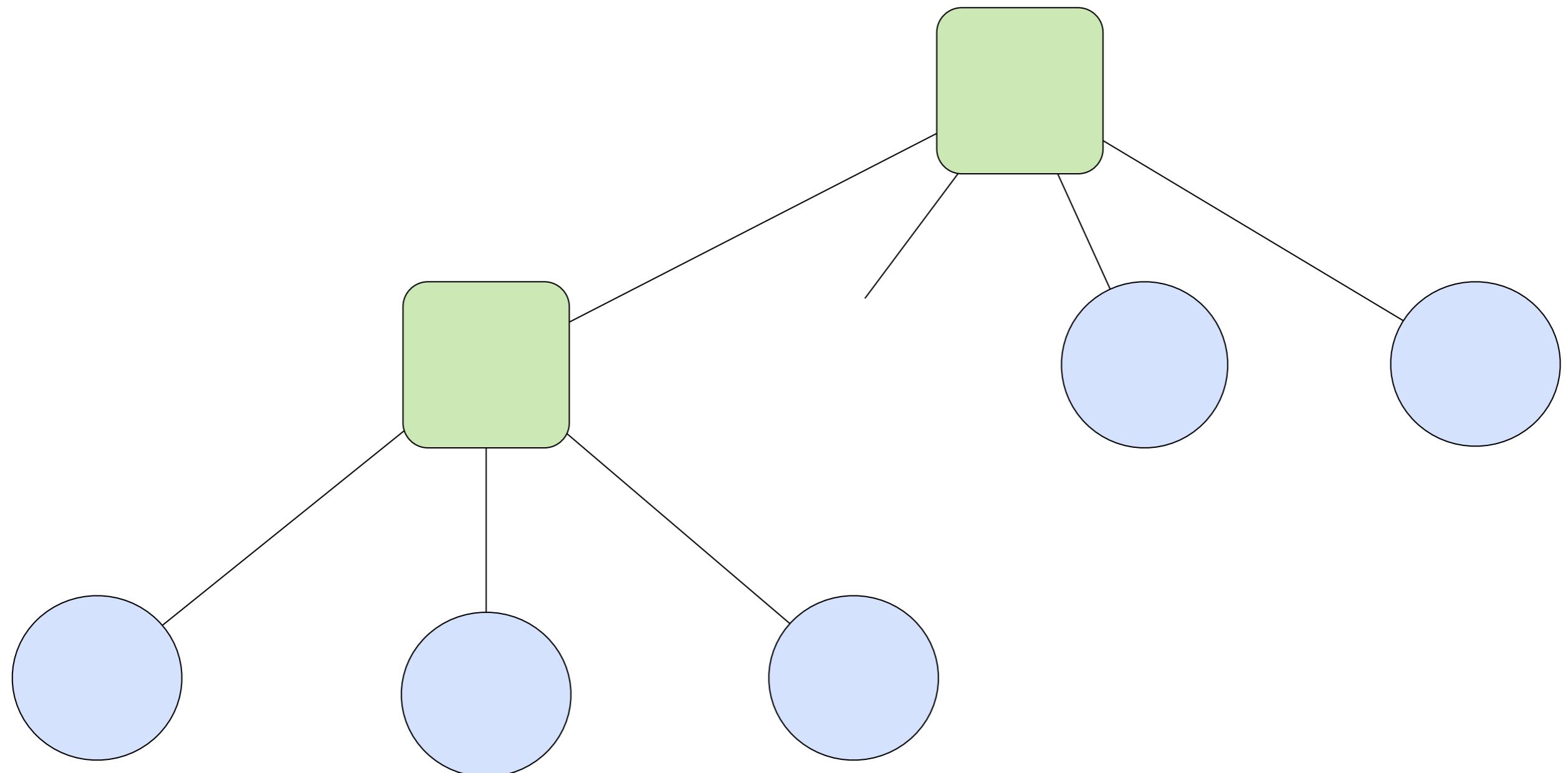
AllForOne



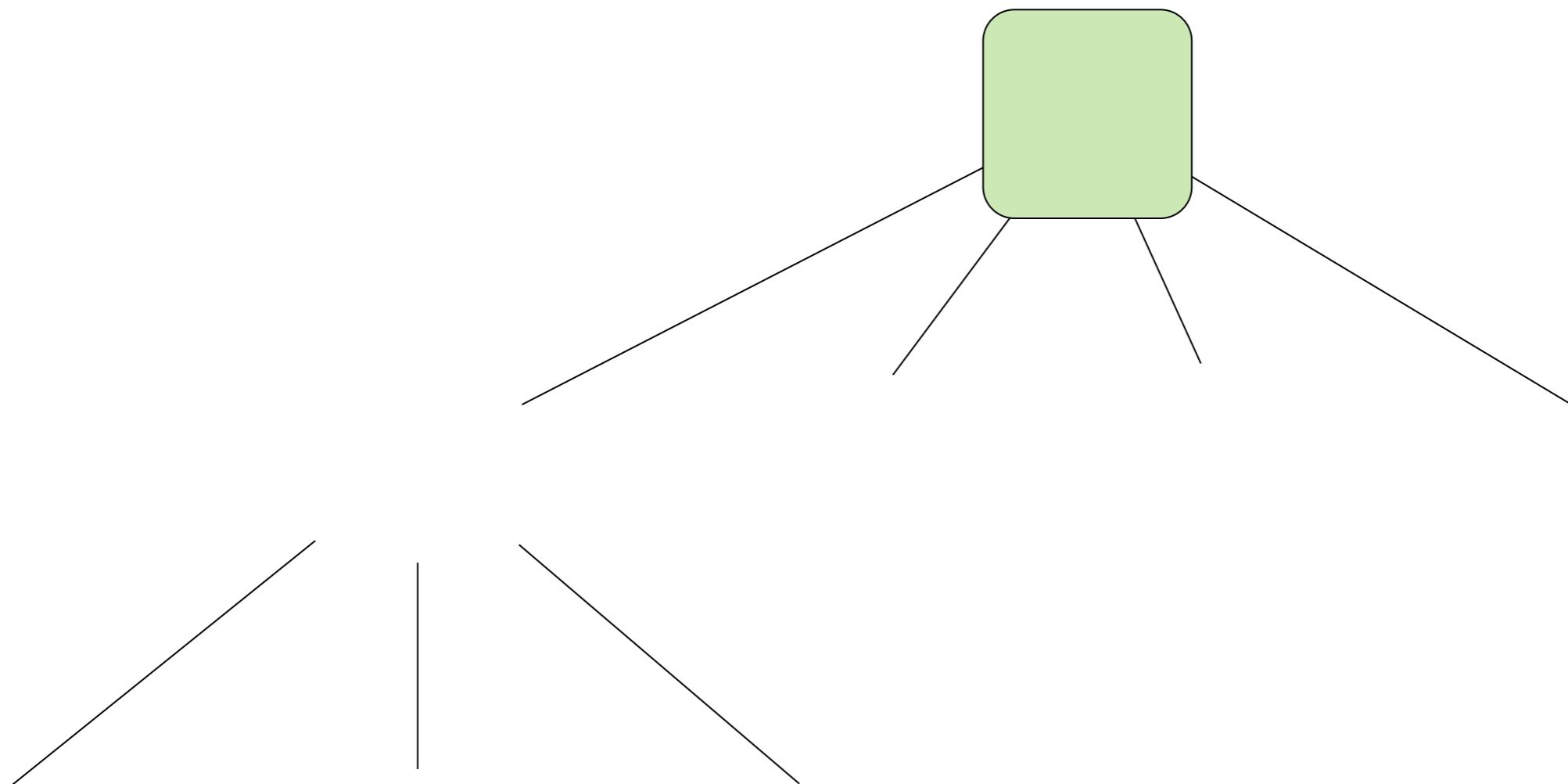
Supervisor Hierarchies



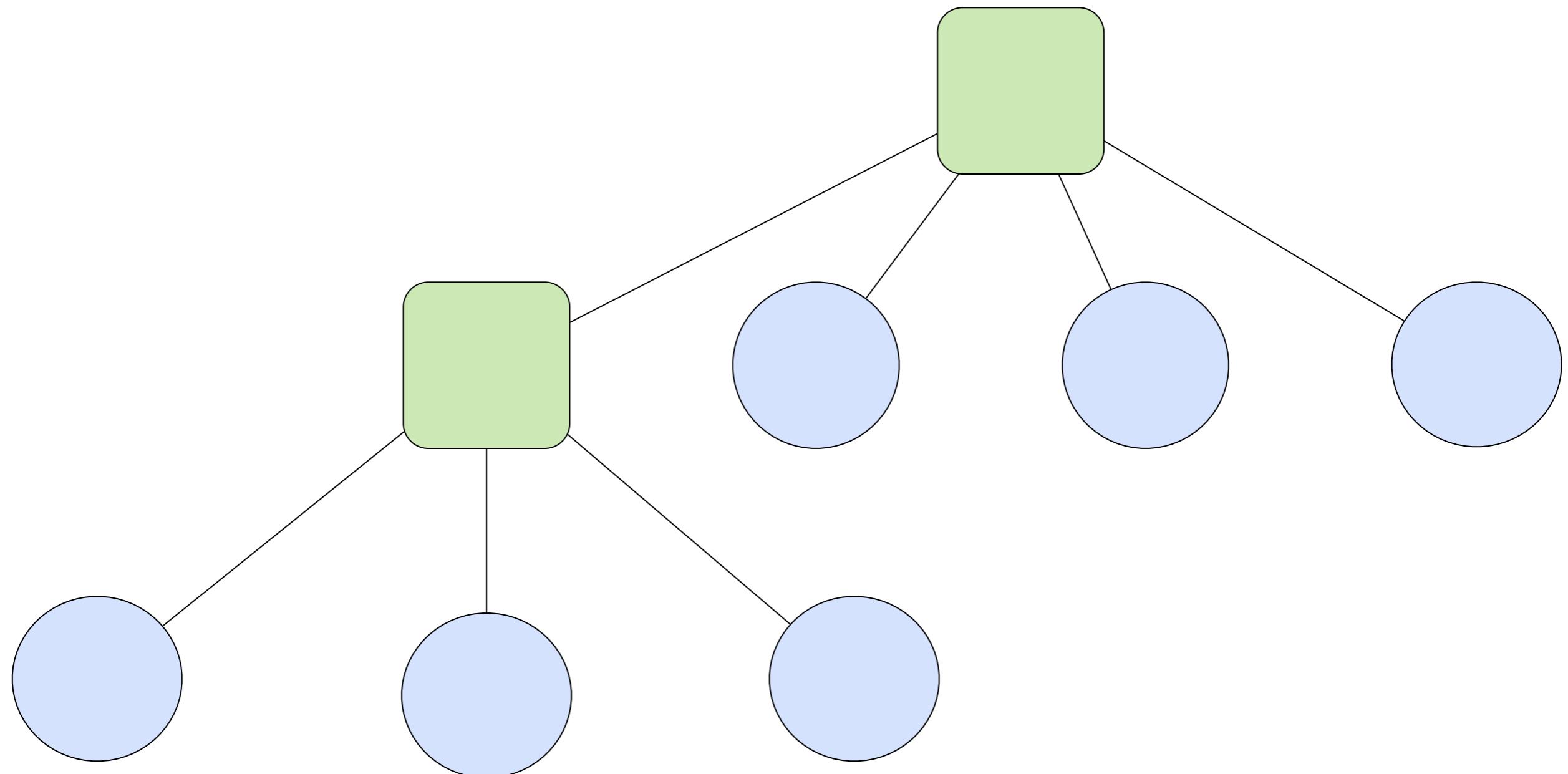
Supervisor Hierarchies



Supervisor Hierarchies



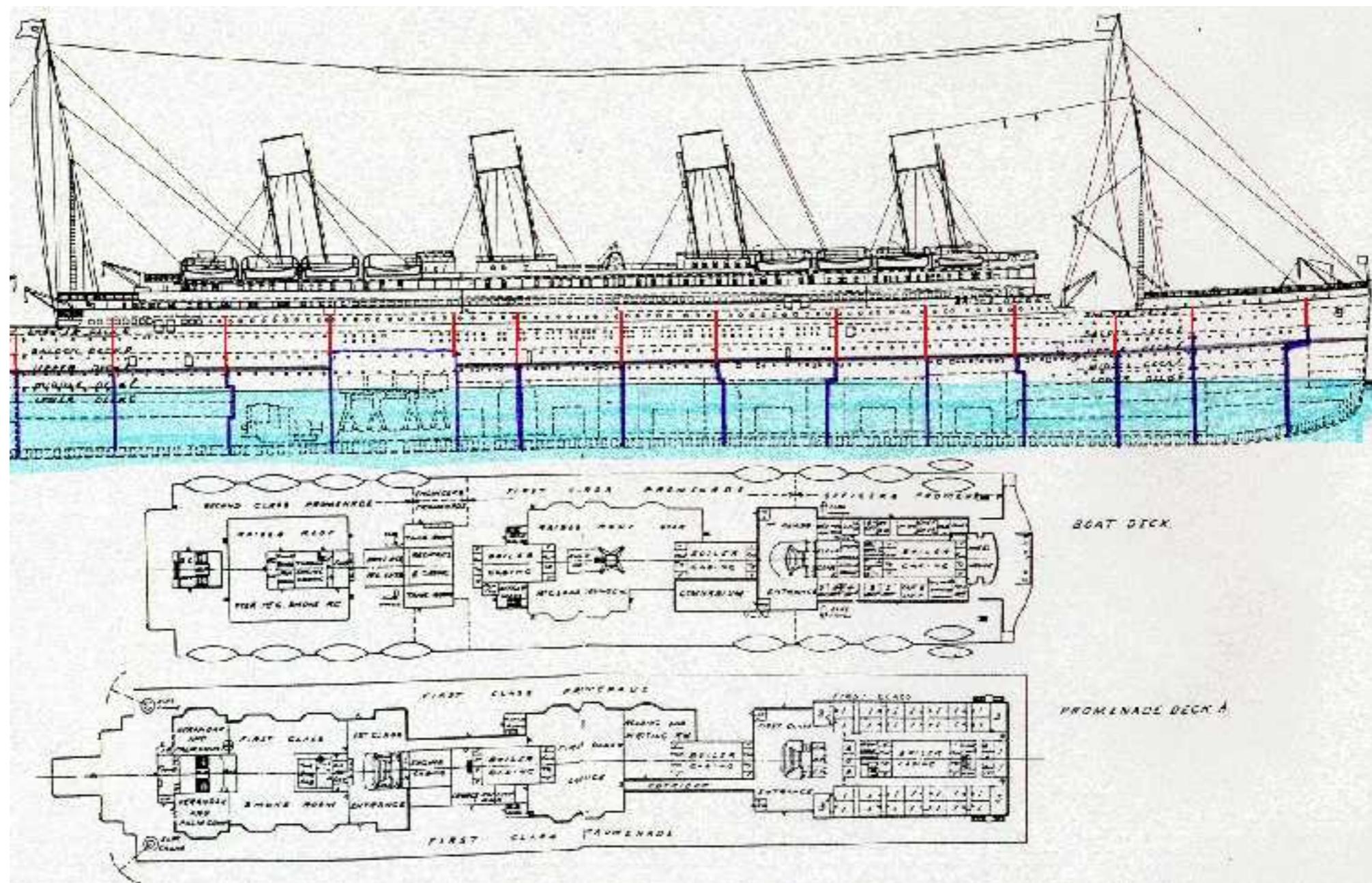
Supervisor Hierarchies



Fail fast

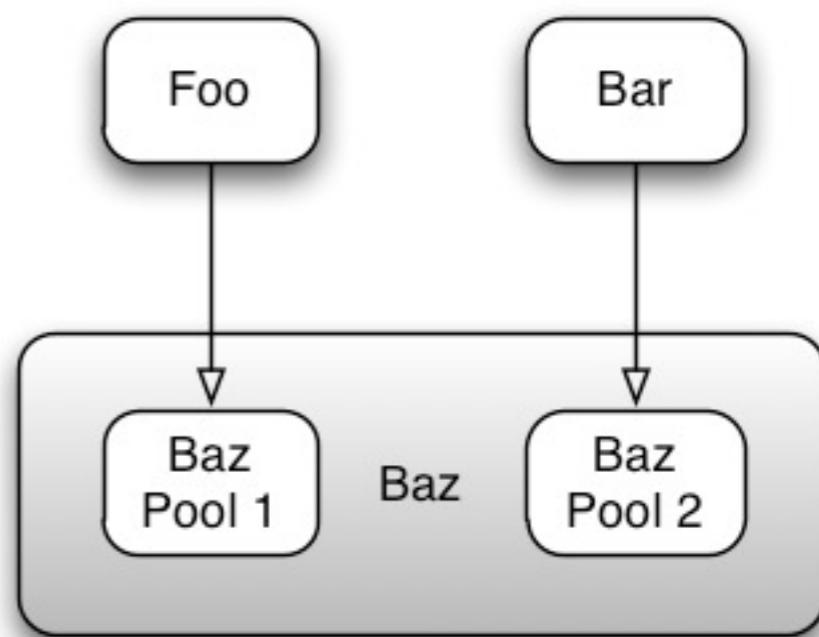
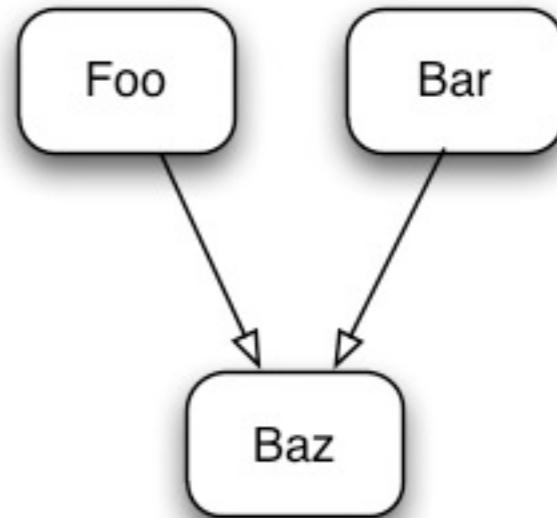
- Avoid “slow responses”
- Separate:
 - SystemError - resources not available
 - ApplicationError - bad user input etc
- Verify resource availability before starting expensive task
- Input validation immediately

Bulkheads



Bulkheads

- Partition and tolerate failure in one part
- Redundancy
- Applies to threads as well:
 - One pool for admin tasks to be able to perform tasks even though all threads are blocked

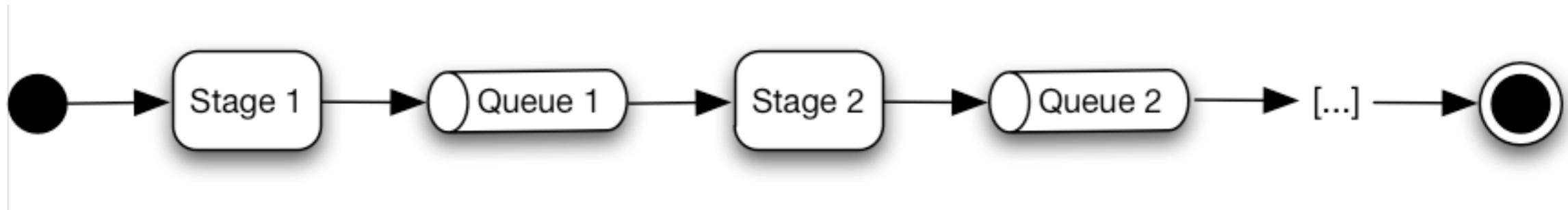


Steady State

- Clean up after you
- Logging:
 - RollingFileAppender (log4j)
 - logrotate (Unix)
 - Scribe - server for aggregating streaming log data
 - Always put logs on separate disk

Throttling

- Maintain a steady pace
- Count requests
 - If limit reached, back-off (drop, raise error)
- Queue requests
 - Used in for example *Staged Event-Driven Architecture (SEDA)*



?

A soft-focus, grayscale photograph of a mountain range. The peaks are visible through a layer of fog or mist, creating a serene and somewhat ethereal atmosphere.

thanks
for listening

Extra material

Client-side consistency

- Strong consistency
- Weak consistency
- Eventually consistent
- Never consistent

Client-side Eventual Consistency levels

- Casual consistency
- Read-your-writes consistency (important)
- Session consistency
- Monotonic read consistency (important)
- Monotonic write consistency

Server-side consistency

[**N** = the number of nodes that store replicas of the data]

[**W** = the number of replicas that need to acknowledge the receipt of the update before the update completes]

[**R** = the number of replicas that are contacted when a data object is accessed through a read operation]

Server-side consistency

- [$W + R > N$] strong consistency
- [$W + R \leq N$] eventual consistency