RX-M Cloud Native Consulting

Microservices

An in depth look at the microservices architecture pattern

Objectives

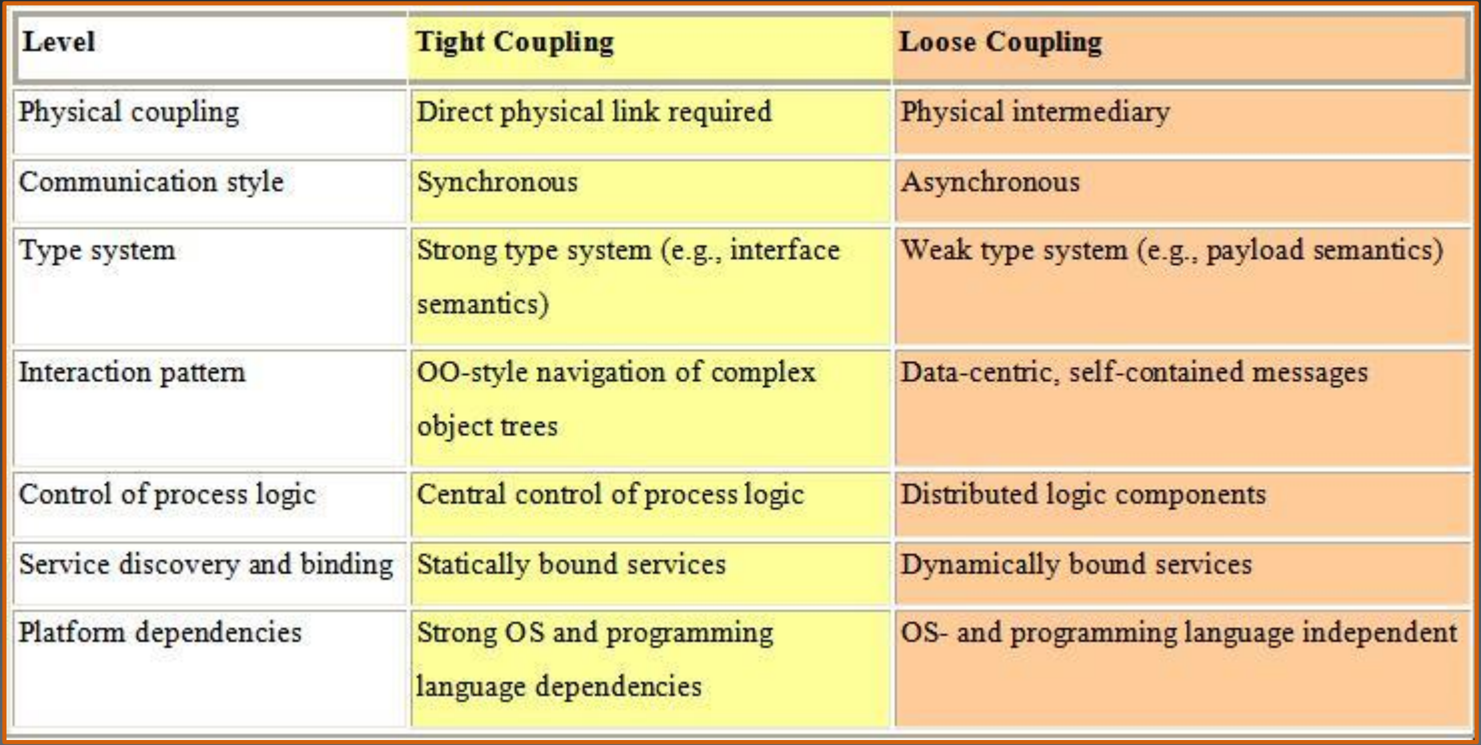
- Explain the benefits of loosely coupled systems
- Describe the difference between asynchronous and synchronous communications
- Map out the behavior of event based systems
- List some of the more important messaging platforms

Loosely Coupled Systems

- A loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components
- One of the most critical enablers of loosely coupled systems is messaging

Loose Coupling

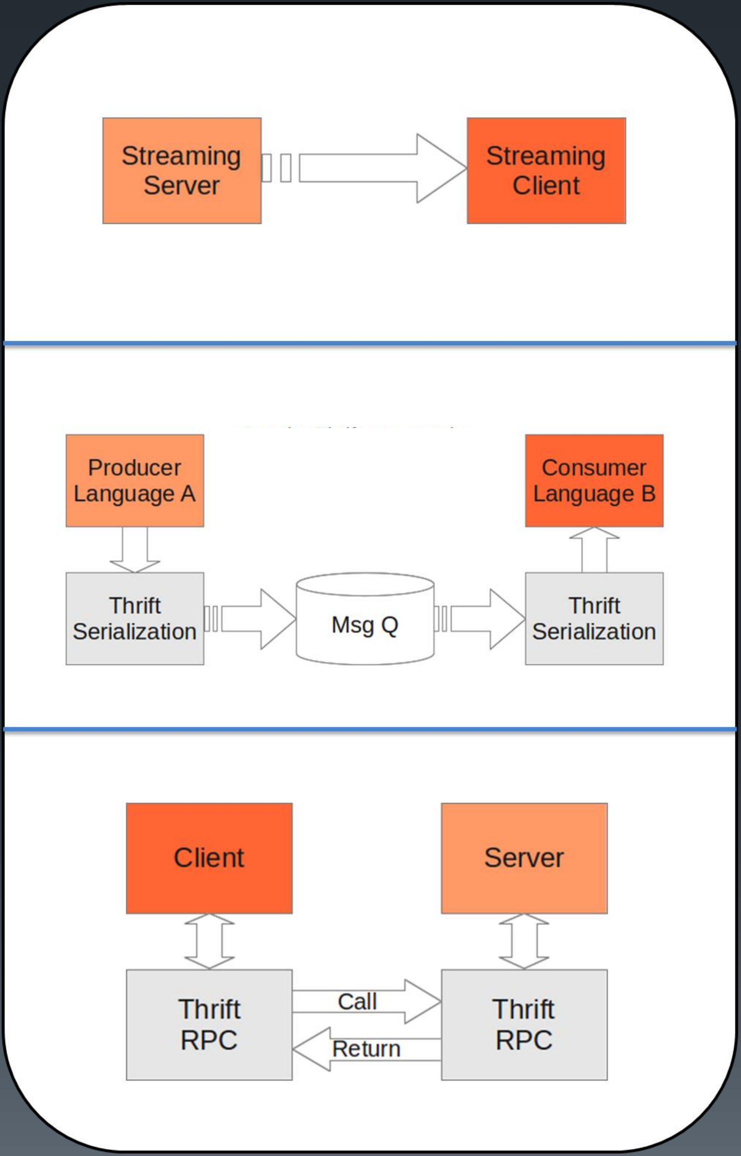
- Systems that are loosely coupled have independent lifespans
  - If service A crashes or is taken down, service B should not crash
  - Rather service B should either
    - Not know (as would be the case in many messaging scenarios)
    - Rediscover and reconnect (trying repeatedly over time as necessary)
  - Services directly depending upon other service (e.g. client/server relationships) should degrade favorably until the dependency is resolved
- Loose coupling is critical to overall application safety
  - Isolating failure modes rather than propagating them



Level	**Tight Coupling**	**Loose Coupling**
Physical coupling	Direct physical link required	Physical intermediary
Communication style	Synchronous	Asynchronous
Type system	Strong type system (e.g., interface semantics)	Weak type system (e.g., payload semantics)
Interaction pattern	OO-style navigation of complex object trees	Data-centric, self-contained messages
Control of process logic	Central control of process logic	Distributed logic components
Service discovery and binding	Statically bound services	Dynamically bound services
Platform dependencies	Strong OS and programming language dependencies	OS- and programming language independent

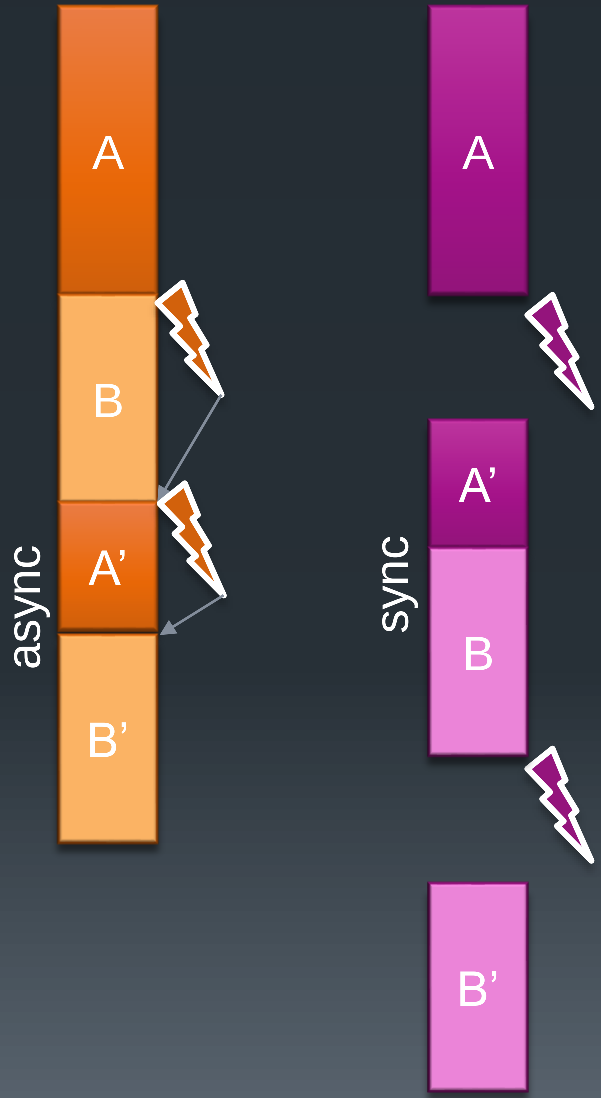
Communications Schemes

- **Streaming** – Communications characterized by an ongoing flow of bytes from a server to one or more clients.
  - Example: An internet radio broadcast where the client receives bytes over time transmitted by the server in an ongoing sequence of small packets.
- **Messaging** – Message passing involves one way asynchronous, often queued, communications, producing loosely coupled systems.
  - Example: Sending an email message where you may get a response or you may not, and if you do get a response you don't know exactly when you will get it.
- **RPC** – Remote Procedure Call systems allow function calls to be made between processes on different computers.
  - Example: An iPhone app calling a service on the Internet which returns the weather forecast.



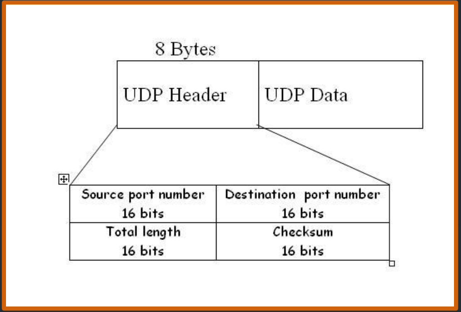
Asynchrony

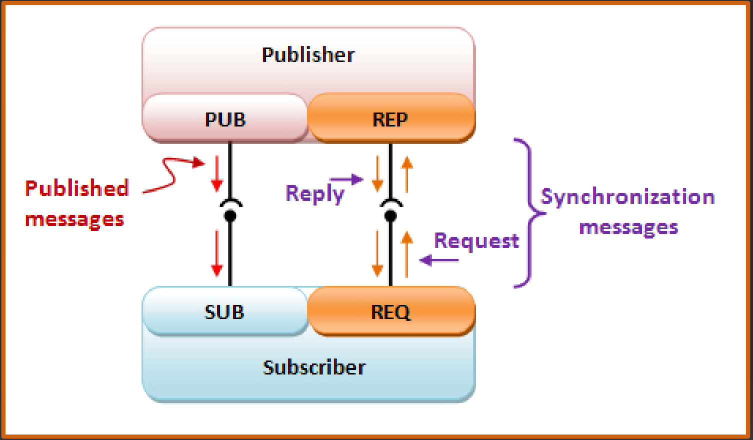
- Synchronous communication
  - Call block until the operation completes
  - Easy to reason about
  - One knows when things complete and what the status is
- Asynchronous communication
  - The caller doesn't wait for the operation to complete
  - May not even care whether or not the operation completes
  - Useful for long-running jobs
  - Good for low latency operations
- Streaming can be sync but is usually async
- Messaging is almost always async
- RPC can be either sync or async
- REST is sync by definition when over HTTP
  - Schemes like chunked responses and long polling attempt to alleviate

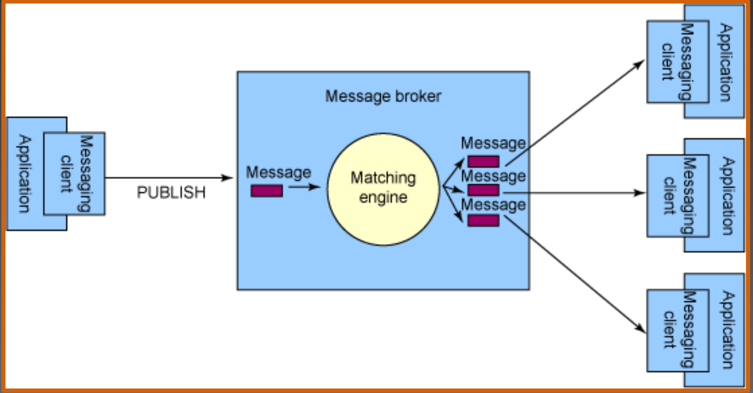


Types of Messaging

- Raw Network Messaging
  - UDP
  - No store, no forward
  - True Multicast, True Broadcast
  - Fast as it gets
- Library Based
  - Can offer some delivery assurances (retry)
  - Deploys in process
  - No server or middleware required
  - Examples
    - Nanomsg
    - ZeroMQ
- Message Oriented Middleware (MOM)
  - Store and forward
  - Pub/sub
  - Perhaps various Transaction Levels
  - Routing
  - More complicated deployment
  - Not the fastest messaging solution
  - Examples
    - NATS
    - Kafka
    - RabbitMQ
    - ActiveMQ
    - MSMQ
    - Redis







AMQP

- Advanced Message Queuing Protocol (AMQP)
- An OASIS open standard
- Application layer protocol for message-oriented middleware
  - message orientation
  - Queuing
  - routing (including point-to-point and publish-and-subscribe)
  - reliability
  - security
- AMQP mandates the behavior of the messaging provider and client to the extent that implementations from different vendors are truly interoperable, in the same way as SMTP, HTTP, FTP, etc. have created interoperable systems
- Previous attempts to standardize middleware have happened at the API level (e.g. JMS) and thus did not ensure interoperability
- Unlike JMS, which merely defines an API, AMQP is a wire-level protocol
- Any tool that can create and interpret messages that conform to this data format can interoperate with any other compliant tool irrespective of implementation language
- Broker Implementations
  - SwiftMQ, a commercial JMS, AMQP 1.0 and AMQP 0.9.1 broker and a free AMQP 1.0 client
  - Windows Azure Service Bus, Microsoft's cloud based messaging service
  - Apache Qpid, an open-source project at the Apache Foundation
  - Apache ActiveMQ, an open-source project at the Apache Foundation
  - Apache Apollo, open-source modified version of the ActiveMQ project at the Apache Foundation (threading functionality replaced and non-blocking techniques implemented more widely)
  - RabbitMQ an open-source project sponsored by Pivotal, supports AMQP 1.0 and other protocols via plugins

Java JMS 2.0

- Java Message Service (JMS) originally released in 2001
- Java based Message Oriented Middleware (MOM) API
- Used to send messages between two or more clients
- JMS is a part of the Java Platform, Enterprise Edition
  - JSR 914
- Allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous
- JMS 2.0 is the first update in over 10 years
  - Simplified API
  - Async send
  - Delayed delivery
  - Scaling through shared topics
- Not the best choice for microservices as it is Java specific (like RMI)

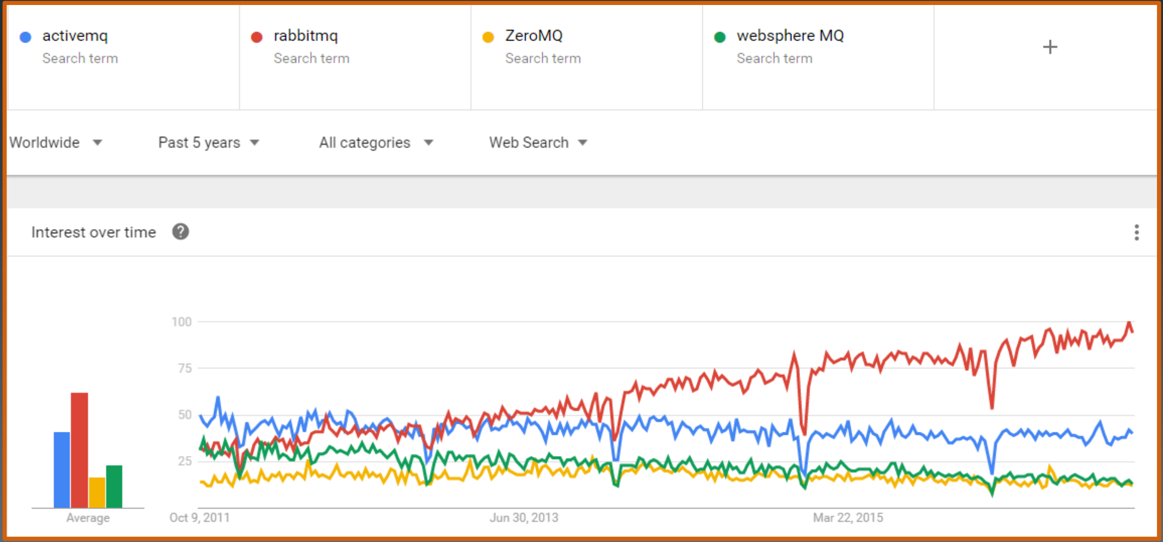
- Many Provider implementations
  - Apache ActiveMQ
  - Apache Qpid, using AMQP[5]
  - Oracle Weblogic and AQ from Oracle
  - EMS from TIBCO
  - FFMQ, GNU LGPL licensed
  - JBoss Messaging and HornetQ
  - JORAM, from the OW2 Consortium
  - Open Message Queue, from Oracle
  - OpenJMS, from The OpenJMS Group
  - Solace JMS from Solace Systems
  - RabbitMQ by Pivotal
  - SAP Process Integration ESB
  - SonicMQ from Progress Software
  - SwiftMQ
  - Tervela
  - Ultra Messaging from Informatica
  - webMethods from Software AG
  - WebSphere MQ FioranoMQ

MSMQ

- Microsoft Message Queuing [MSMQ]
- A message queue implementation developed by Microsoft and deployed in its Windows Server operating systems since Windows NT 4 and Windows 95
- The latest Windows 8 also includes this component
- In addition to its mainstream server platform support, MSMQ has been incorporated into Microsoft Embedded platforms since 1999 and the release of Windows CE 3.0
- Support for three transmission modes
  - Express – not written to disk, no ack
  - Reliable – written to disk, acked
  - Transactional – ACID, DTC eligible
- In .Net Systems, implemented through WCF Reliable Messaging
- Not the best choice for microservices as it is Windows specific

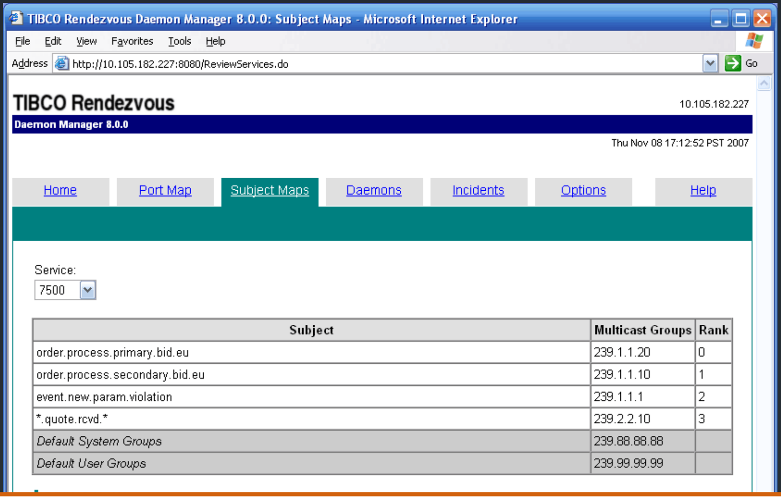
High Profile Messaging Platforms

- Websphere MQ – The first MOM system released in 1992
  - Originally known as MQ Series
  - A revelation
  - Still important but loosing the spotlight to open source and others
- ActiveMQ – Most popular community open source MQ system
  - Supports everything (JMS, JDBC, STOMP, XMPP, MQTT, REST, OpenWrite, …)
  - **Apache Apollo** – rewrite of ActiveMQ in Scala, much simpler and faster
- Apache QPID – AMQP driven messaging (100% AMQP support)
  - RedHat MRG enterprise messaging product is based on this
- RabbitMQ – Most popular commercially backed open source MQ system
  - Erlang based, elegant, fast, reliable
- ZeroMQ – Library based messaging, thus very fast, but brokerless, requiring rendezvous



High Performance Messaging Systems

- Tibco Rendezvous [RV] – frequently used in financial applications, fast, expensive
- Informatica Ultra Messaging – Formerly 29West, fast, expensive (a little cheaper than Tibco)
- ZeroMQ – High performance library only messaging, open source
- NATS - open source messaging broker designed for microservices and cloud native applications
- Apache Kafka - a distributed platform supporting pub/sub, processing and storage for streams of data in a distributed, replicated cluster, originally created at LinkedIn
- Apache Pulsar – a distributed platform supporting queuing and pub-sub messaging, originally created at Yahoo


TIBCO Rendezvous Daemon Manager 8.0.0: Subject Maps - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://10.105.182.227:8080/ReviewServices.do
Go
Home
Port Map
Subject Maps
Daemons
Incidents
Options
Help

TIBCO Rendezvous

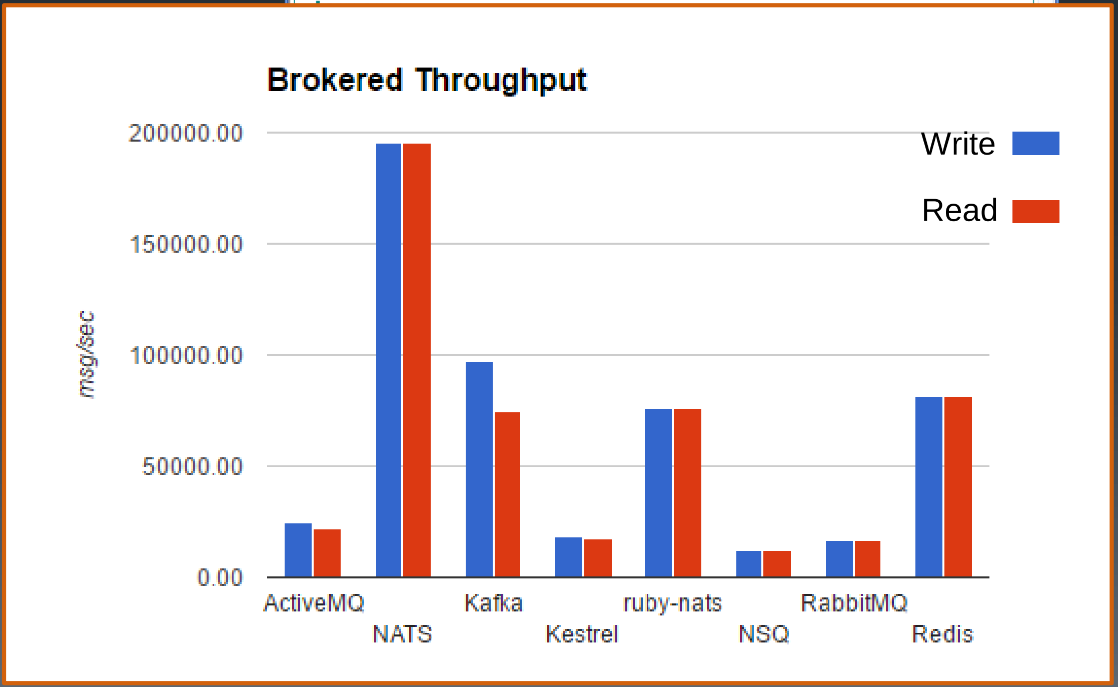
Daemon Manager 8.0.0

10.105.182.227

Thu Nov 08 17:12:52 PST 2007

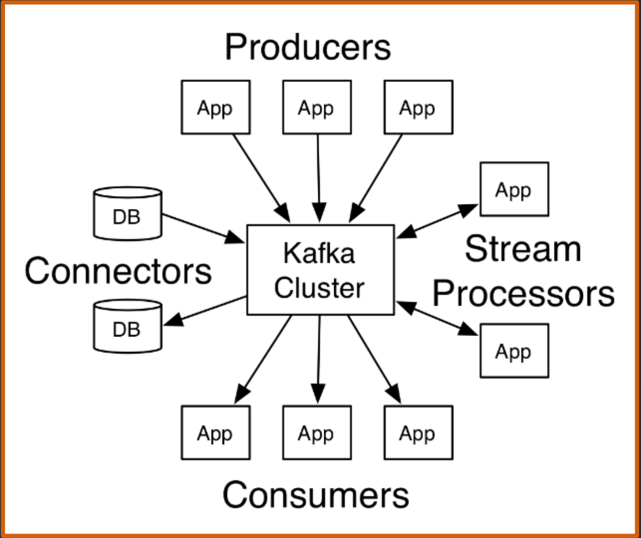
Service: 7500

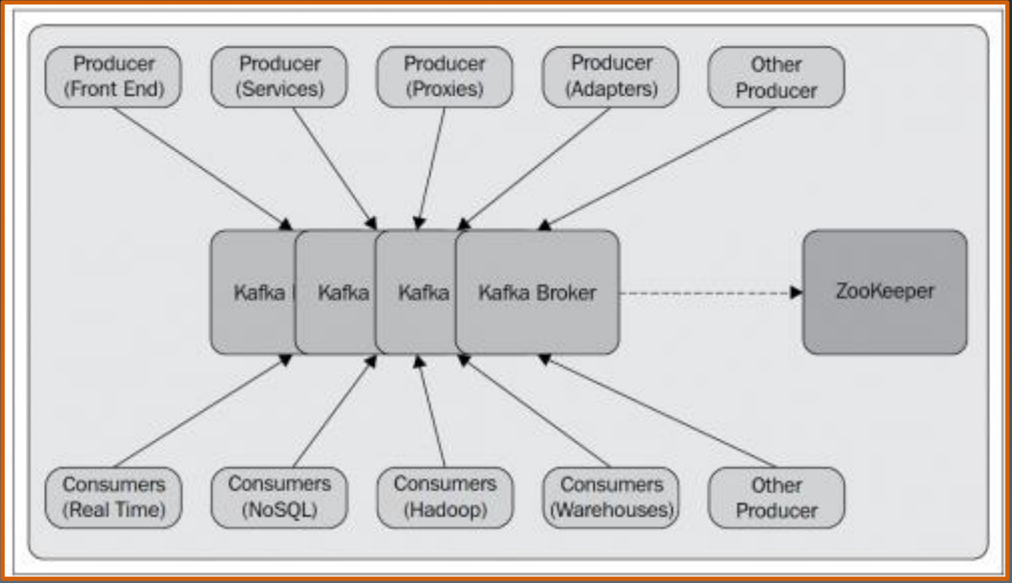
Subject	Multicast Groups	Rank
order.process.primary.bid.eu	239.1.1.20	0
order.process.secondary.bid.eu	239.1.1.10	1
event.new.param.violation	239.1.1.1	2
.quote.rcvd.	239.2.2.10	3
Default System Groups	239.88.88.88	
Default User Groups	239.99.99.99	



Distributed Messaging

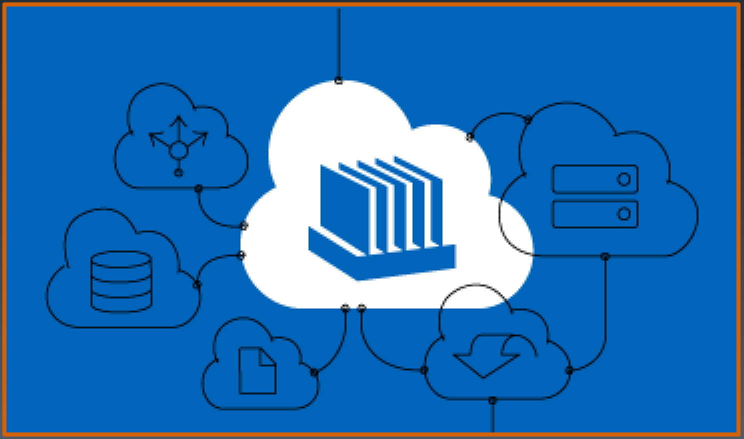
- Apache Kafka
  - Open source, publish-subscribe message broker
  - Amazon Kinesis Streams, managed Kafka
- Implemented as a distributed commit log
- Written in Scala
- Unified, high-throughput, low-latency platform for handling real-time data feeds
- Developed at LinkedIn, open sourced in 2011
- Designed to allow a single cluster to serve as the central data backbone for an organization
- Can be elastically and transparently expanded without downtime
- A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients
- Data streams are partitioned and replicated over a cluster of machines
  - Allows data streams larger than the capability of any single machine and to allow clusters of coordinated consumers





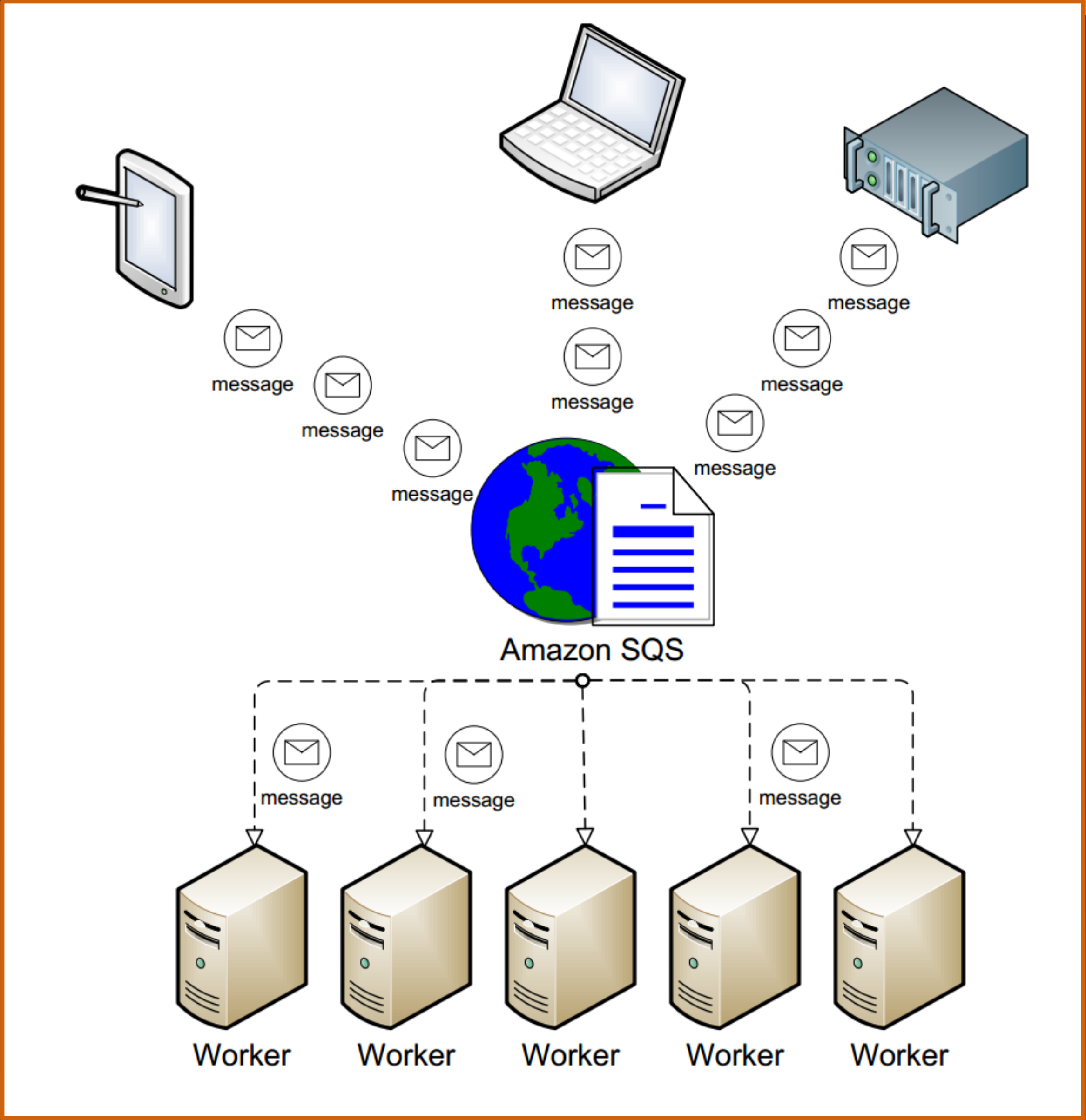
Cloud Based Messaging

- Amazon Simple Queue Service (SQS) is a fast, reliable, scalable, fully managed message queuing service
  - SQS is simple and can transmit large volumes of data, at any level of throughput, without losing messages or requiring other services to be available
- Azure Queue
- Google Cloud Pub/Sub
- Rackspace offers CloudQueues
- Heroku offers RabbitMQ
- EngineYard offers IronMQ
- Etc.



Amazon SQS

- Amazon Simple Queue Service (SQS) is a reliable distributed messaging system
  - A reasonable choice for fault-tolerant applications
  - At least once or at most once delivery
- Messages are stored in user defined queues
  - Each queue is accessed by URL
  - Available to the Internet
  - An Access Control List (ACL) determines access to the queue
- Any messages that you send to a queue are retained for up to four days (or until they are read and deleted by an application)
  - Messages read are hidden for the hide window time
  - Message delivery can have a preconfigured delay
- Amazon Kinesis, a fully managed real-time streaming data service
  - Hosted Apache Kafka
  - Designed for streaming big data
  - Multiple applications can receive the same data
  - Data is delivered in order and cached for up to 24 hours
    - i.e. A client can replay data from the prior 24 hours



IoT Messaging

- MQTT (formerly MQ Telemetry Transport)
  - ISO standard PRF 20922
  - Publish-subscribe-based "lightweight" messaging protocol for use on top of TCP/IP
  - Designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited
- Powers Facebook Messanger
- The publish-subscribe messaging pattern requires a message broker
  - The broker is responsible for distributing messages to interested clients based on the topic of a message
  - MQTT can also work without a broker
- 1999: Cirrus Link Solutions Stanford-Clark/Nipper develop MQTT
- 2013: IBM submitted MQTT v3.1 to OASIS
  - The "MQ" in "MQTT" came from IBM's MQ Series message queuing product
- MQTT-SN is a variation of the main protocol aimed at embedded devices on non-TCP/IP networks, such as ZigBee
- Alternative protocols include:
  - Advanced Message Queuing Protocol
  - IETF Constrained Application Protocol
  - XMPP
  - Web Application Messaging Protocol (WAMP)

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios. It is also ideal for mobile applications because of its small size, low power usage, minimised data packets, and efficient distribution of information to one or many receivers (more...)

News

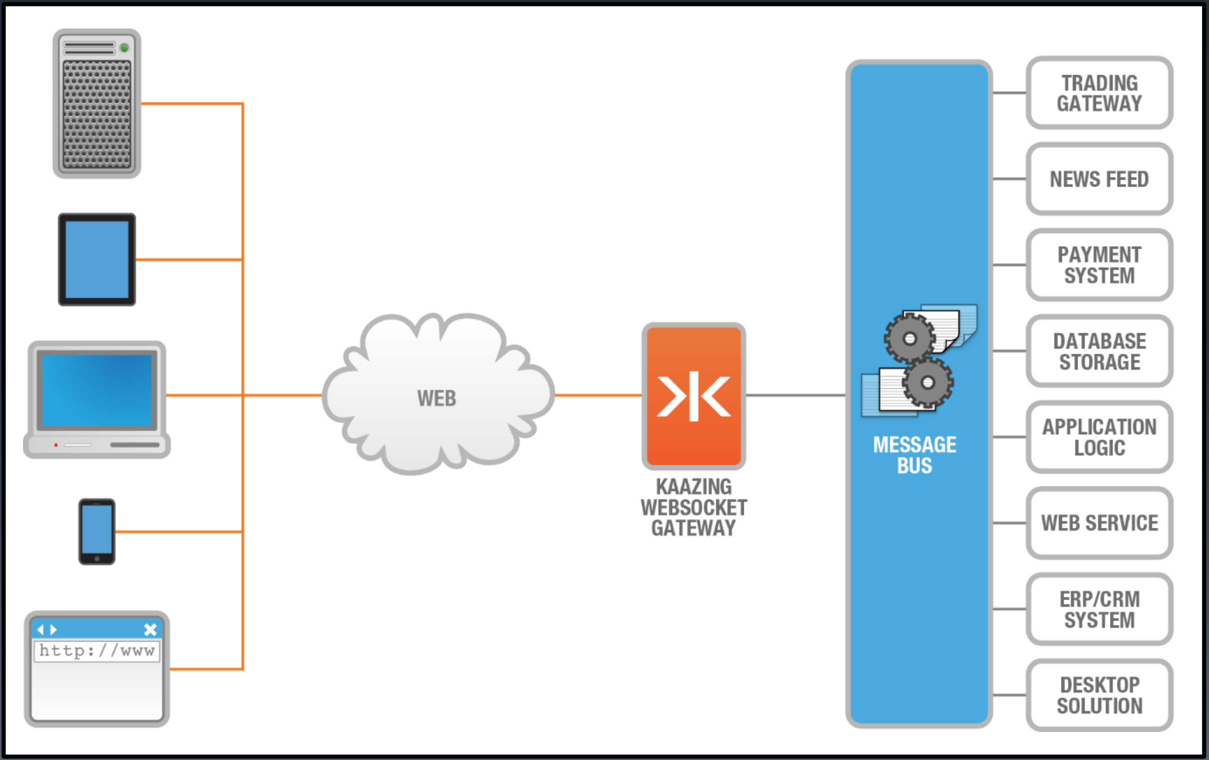
MQTT v3.1.1 now an OASIS Standard

November 7th, 2014 - 5 Comments

Good news everyone! MQTT v3.1.1 has now become an OASIS Standard.

Web Socket

- Web Socket brings high performance messaging to the Web
  - Suitable for Messaging and Streaming
- Products like Kaazing and empowering richer JavaScript applications in the browser
  - Kaazing provides a messaging protocol which operates over Web Sockets
- The PaaS EngineYard offers Pusher, a messaging solution over Web Sockets



Summary

- Loosely coupled systems promote service independence
- Asynchronous systems do not wait for operations to complete before returning
- Event based systems allow subscribers to come and go without disturbing the sender
- A wide range of messaging platforms with several key target markets exist, many of which are useful in microservice based applications

Lab 4

- Building an RPC service

5: Cloud Native Transactions and Event Sourcing

Objectives

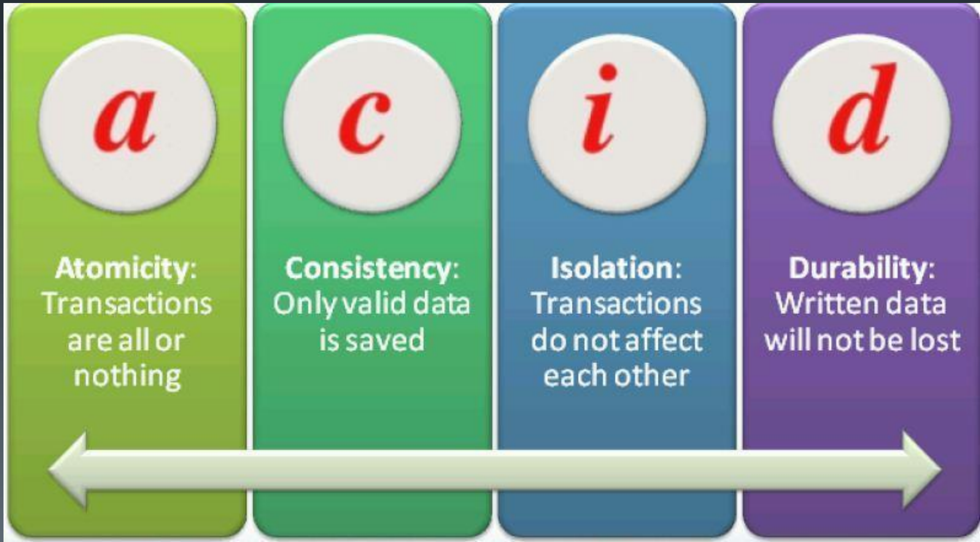
- Define transaction
- Discuss the challenges facing distributed transactions
- Contrast ACID transactions with eventual consistency
- Explain the CAP theorem
- Describe the value of Etags, Event Sourcing and CQRS

State

- State def:
  - All the stored information, at a given instant in time, to which the program has access
- The output of a computer program at any time is completely determined by its current inputs and its state
  - If your state goes bad, your application breaks
- State management presents the single most challenging facet of distributed application design

ACID

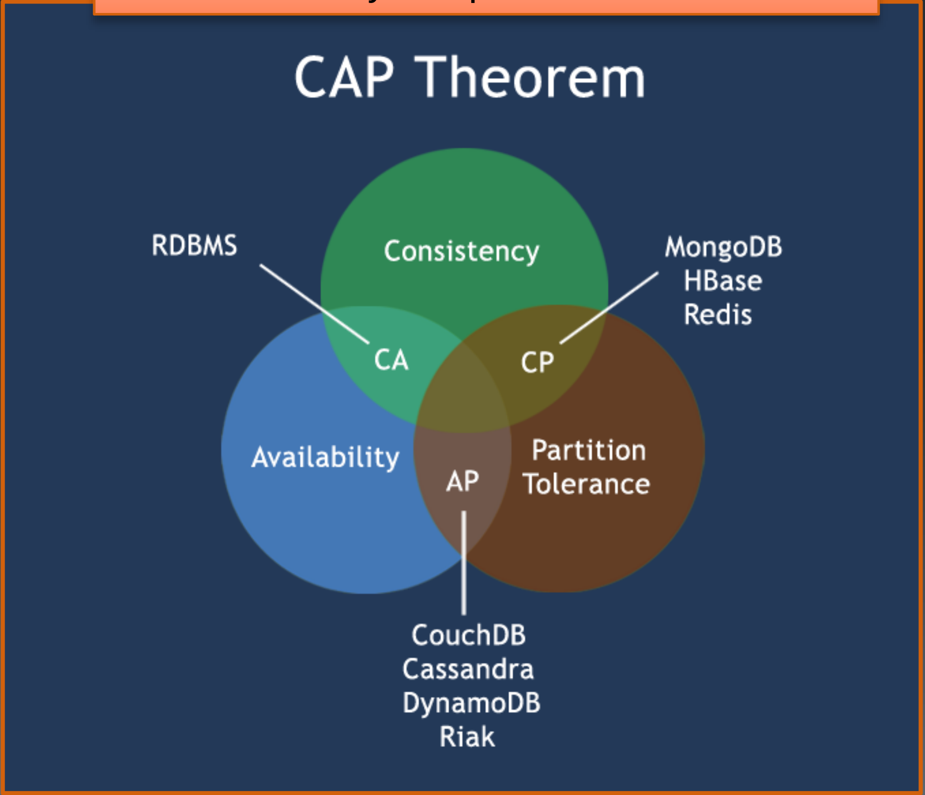
- ACID is a set of properties that guarantee that database transactions are processed reliably
  - Atomicity - requires that each transaction is "all or nothing"
  - Consistency - ensures that any transaction will bring the database from one valid state to another
  - Isolation - ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
  - Durability - means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors
- Transactions - In the context of databases, a single logical operation on the data is called a transaction
- Jim Gray defined the properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically
- In 1983, Andreas Reuter and Theo Härder coined the acronym ACID to describe them


a
Atomicity: Transactions are all or nothing
c
Consistency: Only valid data is saved
i
Isolation: Transactions do not affect each other
d
Durability: Written data will not be lost

The CAP Theorem

- The CAP theorem states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:
  - Consistency - all nodes see the same data at the same time
  - Availability - guarantees every request receives a response whether successful or failed
  - Partition tolerance - the system operates despite message loss or failure of part of the system
- CAP consistency means that any data item has a value reached by applying all the prior updates in some agreed-upon order
  - A consistent service must never forget an update once it has been accepted and the client has been sent a reply
- Availability is a mixture of performance and fault-tolerance
  - A service should keep running and offer rapid responses even if a few replicas have crashed or are unresponsive, and even if some of the data sources it needs are inaccessible
  - No client is ever left waiting, even if we cannot get the needed data
- Partition tolerance means that a system should be able to keep running even if the network itself fails, cutting off some nodes from the others
  - Partitioning is not an issue within modern data centers only across data centers

http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf



The CAP impact on the Cloud

- The CAP Principle was developed by Berkeley Professor Eric Brewer (Brewer 2000)
  - A CAP Theorem was thereafter proved by MIT researchers Seth Gilbert and Nancy Lynch (Gilbert and Lynch 2002)
  - The theorem addresses a strict ACID view of consistency
  - Brewer has since made three key points
    1. Because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned
    2. The choice between C and A can occur many times within the same system at very fine granularity
       - Not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved
    3. All three properties are more continuous than binary
- Brewer argues that the value of quick responses is so high that even a response based on stale data is often preferable to leaving the external client waiting

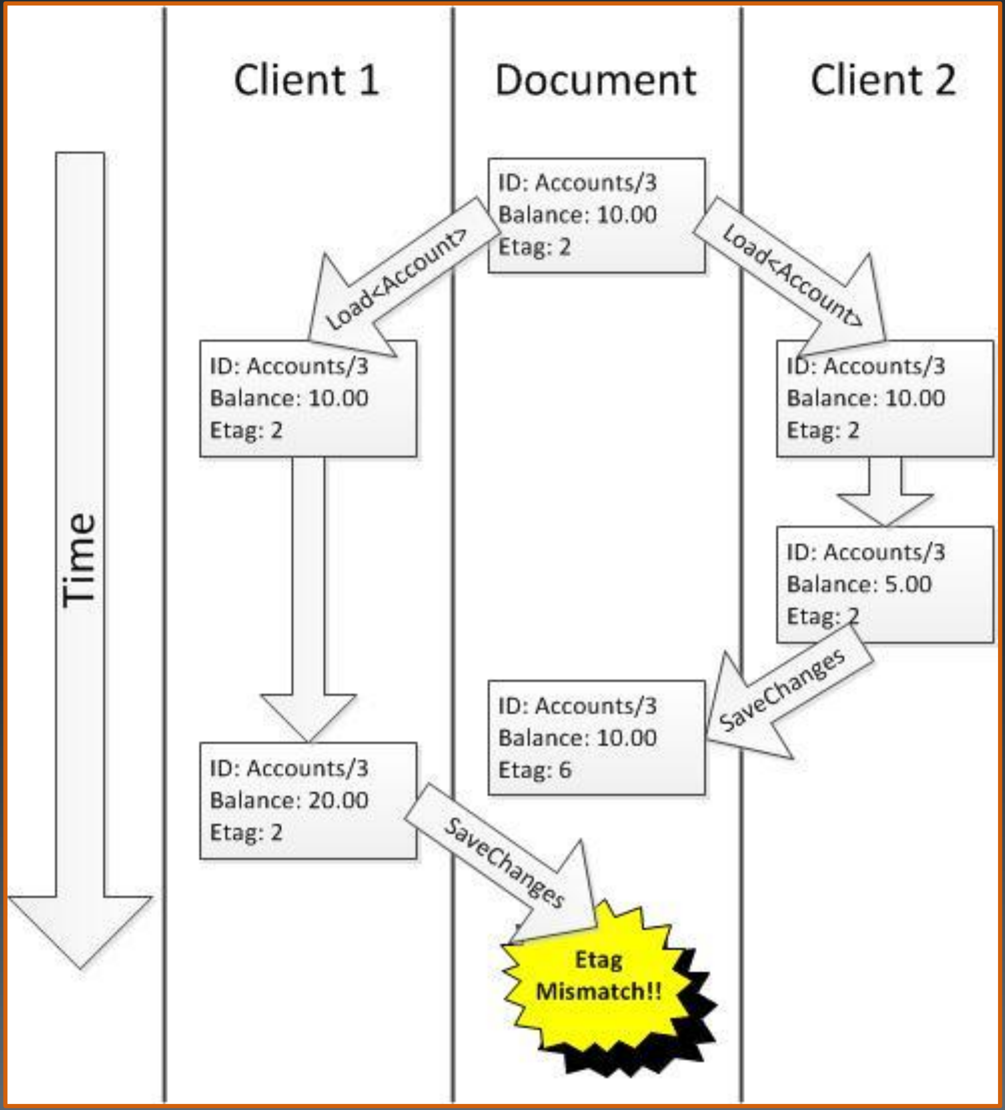
When a web page renders some content with a “broken” icon to designate missing or unavailable content, we are seeing CAP in action

How CAP plays out in application design

- Brewer argues, that it is better to initially run in an optimistic mode when designing large scale applications
  - E.g. Booking the sale without checking the remaining inventory stock
  - A question of priorities
    - scale and performance versus absolute accuracy
- The CAP theorem suggests that sometimes a system should run acceptable risks if by doing so it can offer better responsiveness and scalability
- Consistency in CAP is really a short-hand for two properties
  - An order-based consistency property
    - Updates to data replicated in the system will be applied in the same order at all replicas
  - A durability property
    - Once the system has committed to do an update, it will not be lost or rolled back
- This conflation of consistency with durability is important
  - Durability is expensive and turns out to be of limited value in the first tier of the cloud where services do not keep permanent state

Distributed State

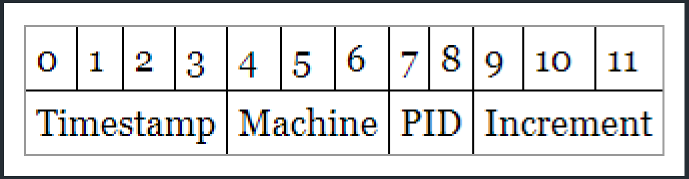
- The web works so well because it is based on:
  - Representations of resources
    - You never have the real resource state, only a representation
    - Your representation may be old the moment you receive it
  - Caching
    - Many web servers would be crushed if they had to directly support all of the clients using pages that they are responsible for
    - The fabric of caches on the web is massive and offloads many servers by one or more orders of magnitude
- The idea that the data you receive might be stale as soon as you receive it is tied up with eventual consistency
- Etags can be used as a state versioning mechanism to ensure that state changes are only applied if you are changing the most recent state



Key Generation

- Keys are a critical feature of distributed data systems
- Keys allow objects to be uniquely identified
  - This is usually a prerequisite for Idempotence
- For this reason keys are best generated at the time of datum/entity creation or ingestion
- Potential Key Algorithm Input Factors:
  - Explicitly assigned prefix
  - Machine ID
    - Process ID
      - Thread ID
  - Time
    - Perhaps highly granular (microsecond, nanosecond, etc.)
  - Monotonically increasing counters
  - Foreign/External Keys
- Unique keys require a set of factors sufficient to eliminate all orthogonal vectors of replication
- Critical to many Idempotent interface implementations

Considerations:
- Key compare time/rate
- Key generation time/rate
- Key storage (data & index)
- Key clustering
- Composite keys and implicit foreign keys

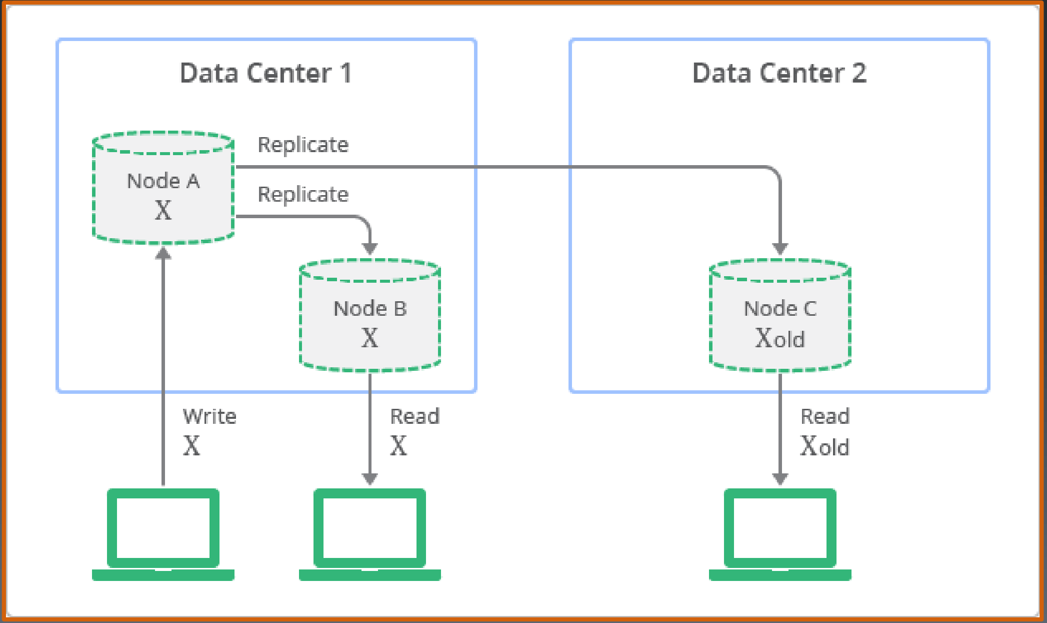


0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				Machine			PID		Increment		

- MongoDB ObjectID (OID) example
- 12 bytes
- 0-3: timestamp in seconds since epoch
  - Provides uniqueness at the granularity of a second
  - The timestamp comes first so OIDs will sort in roughly insertion order, making OIDs efficient to index
  - Implicit creation timestamp
- 4-6: unique machine identifier (usually a hash of the machine's hostname)
  - Statistically ensures that OIDs on different machines will not collide
- 7-8: Process ID
  - Ensures concurrent processes generate unique IDs on the same machine
- 9-11: Increment responsible for uniqueness within a second in a single process
  - Allows up to 256^3 (16,777,216) unique OIDs to be generated per process in a single second
  - Must be synchronized in a multithreaded process

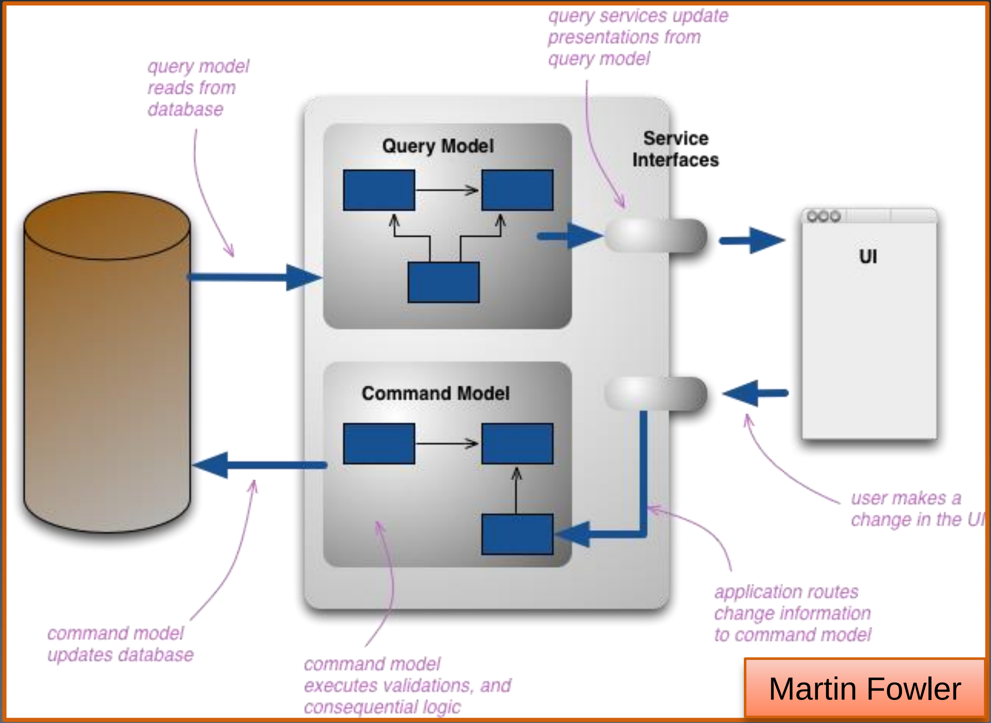
Eventual consistency

- Eventual Consistency
  - A consistency model used in distributed computing
  - Achieves high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value
- Quorum based systems can ensure strong consistency
  - Consul, etcd, zookeeper, etc.
  - Do not scale well due to the high communications overhead between nodes
  - Good for special purposes
    - Leader election
    - Small key/value bits of cluster state
  - Bad for large data storage
- Many times you can trade things you do not need for things you do
  - Read your own writes
    - Trading global consistency for local consistency
    - Others may not see your writes immediately but you will



CQRS

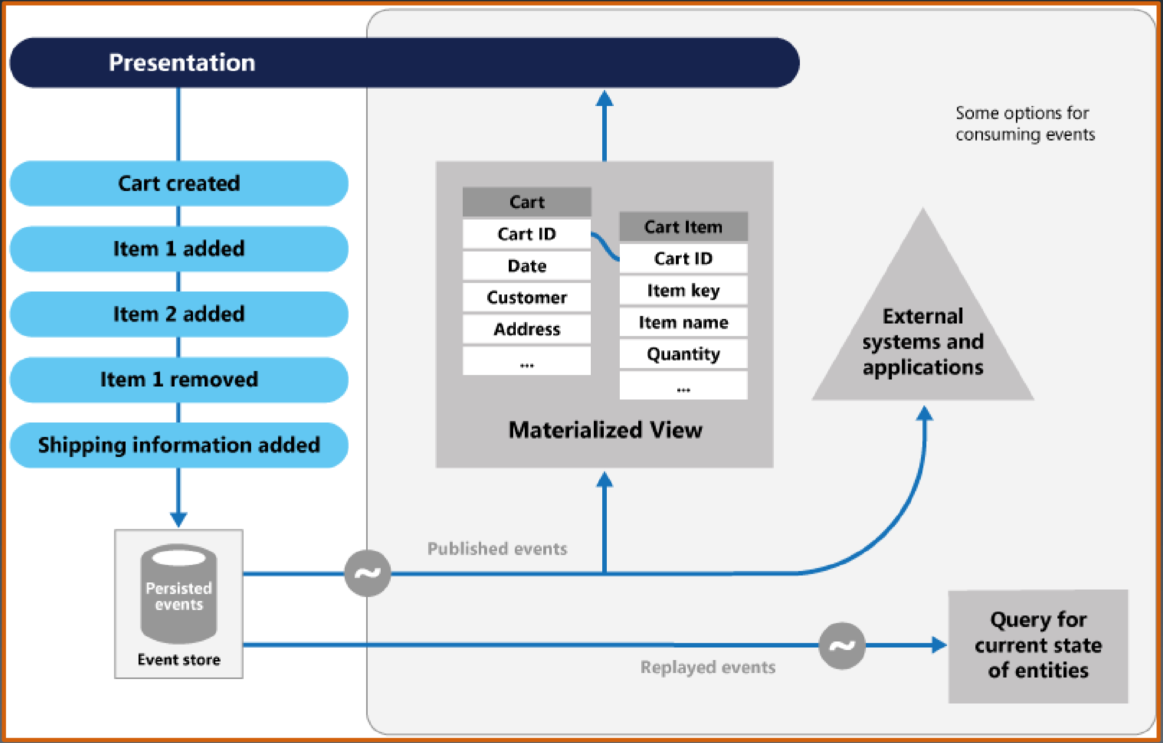
- Command Query Responsibility Segregation (CQRS)
  - A pattern which uses one model to modify state and another to retrieve it
- CQRS can solve intractable problems or make things intractable
  - Should only be used when appropriate (the tax on incorrect usage is much high than with most patterns)
- CQRS differs from CRUD, which uses a single (typically synchronous) model for all operations
  - CRUD = create/read/update/delete operations on (typically) records in a database
- CQRS Commands
  - Modify state
  - Queued and asynchronous
  - Are more expensive
    - Often need to be atomic and/or serialized, perhaps involving some locking
- CQRS Queries
  - Retrieve data
  - Typically synchronous in CQRS
  - Cheap and fast
    - Can often be performed lock less
  - Must tolerate eventual consistency
    - Read your own writes becomes difficult


query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

Martin Fowler

Event sourcing

- Event Sourcing is the process of capturing all inputs as events
- Often combined with eager read derivation
  - Reads don't touch the main database
  - Reporting databases service most reads and are structured to accept all state change events and derive the typical read data model in advance of any user requests for data
  - This makes Queries even faster
- The state of an entity is established by looking at its event history
  - Each event is immutable
  - Balances and other "summary" type states can be computed and optionally cached if fast access is required
    - However the event history is always canonical

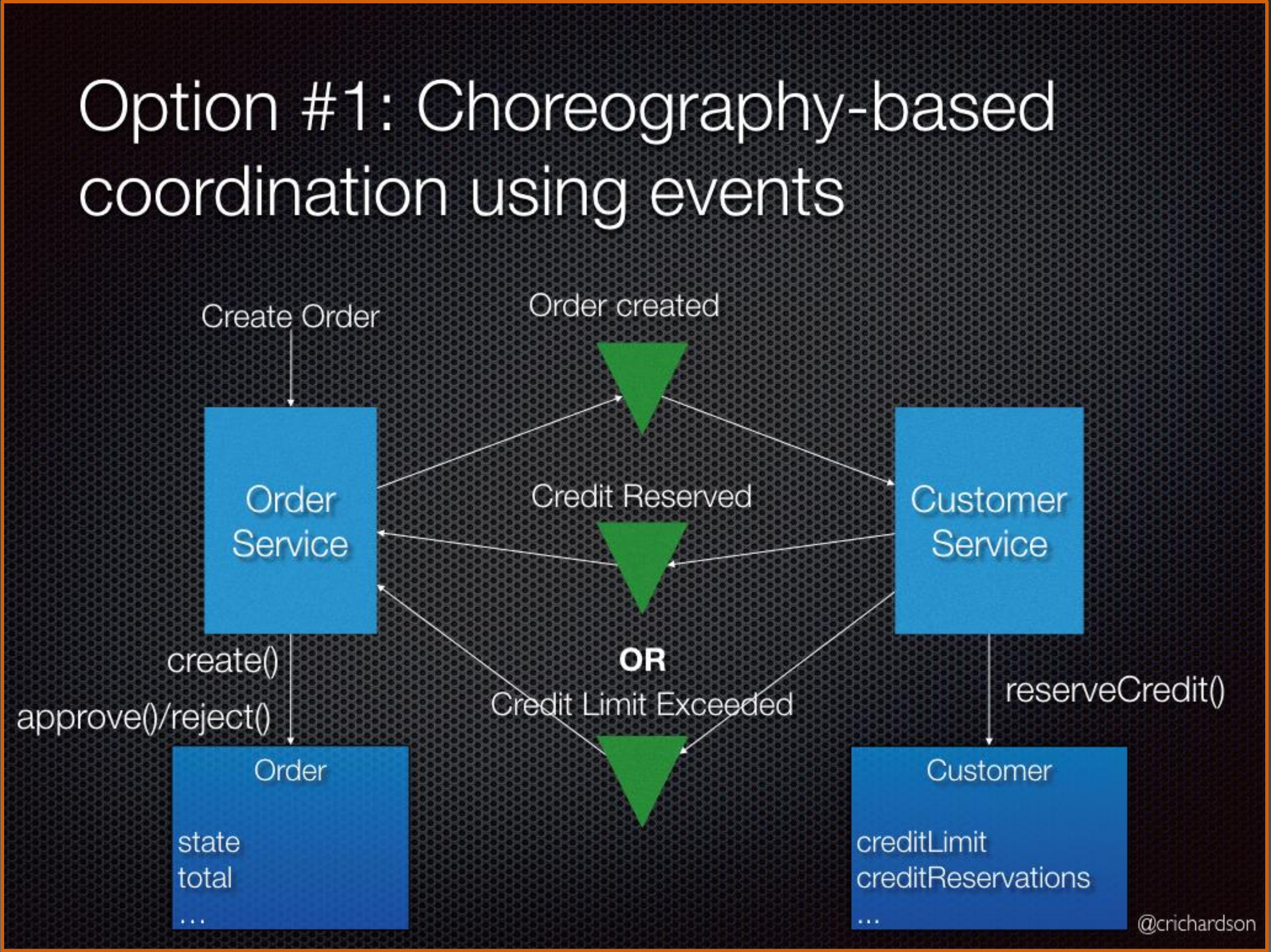


The Saga Pattern

- Microservices are typically designed such that each service has its own database if it is stateful
- Some business transactions span multiple service
- A mechanism may be needed to ensure data consistency across services
  - E.g. an e-commerce store where customers have a credit limit, the application must ensure that a new order will not exceed the customer's credit limit, requiring coordination between the Order and Customer services
- A saga is a sequence of local transactions
  - Each local transaction updates the local state and publishes a message or event to trigger the next local transaction in the saga
  - If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions
- There are two common ways to coordinate sagas:
  - Choreography - each local transaction publishes domain events that trigger local transactions in other services
  - Orchestration - an orchestrator (object) tells the participants what local transactions to execute

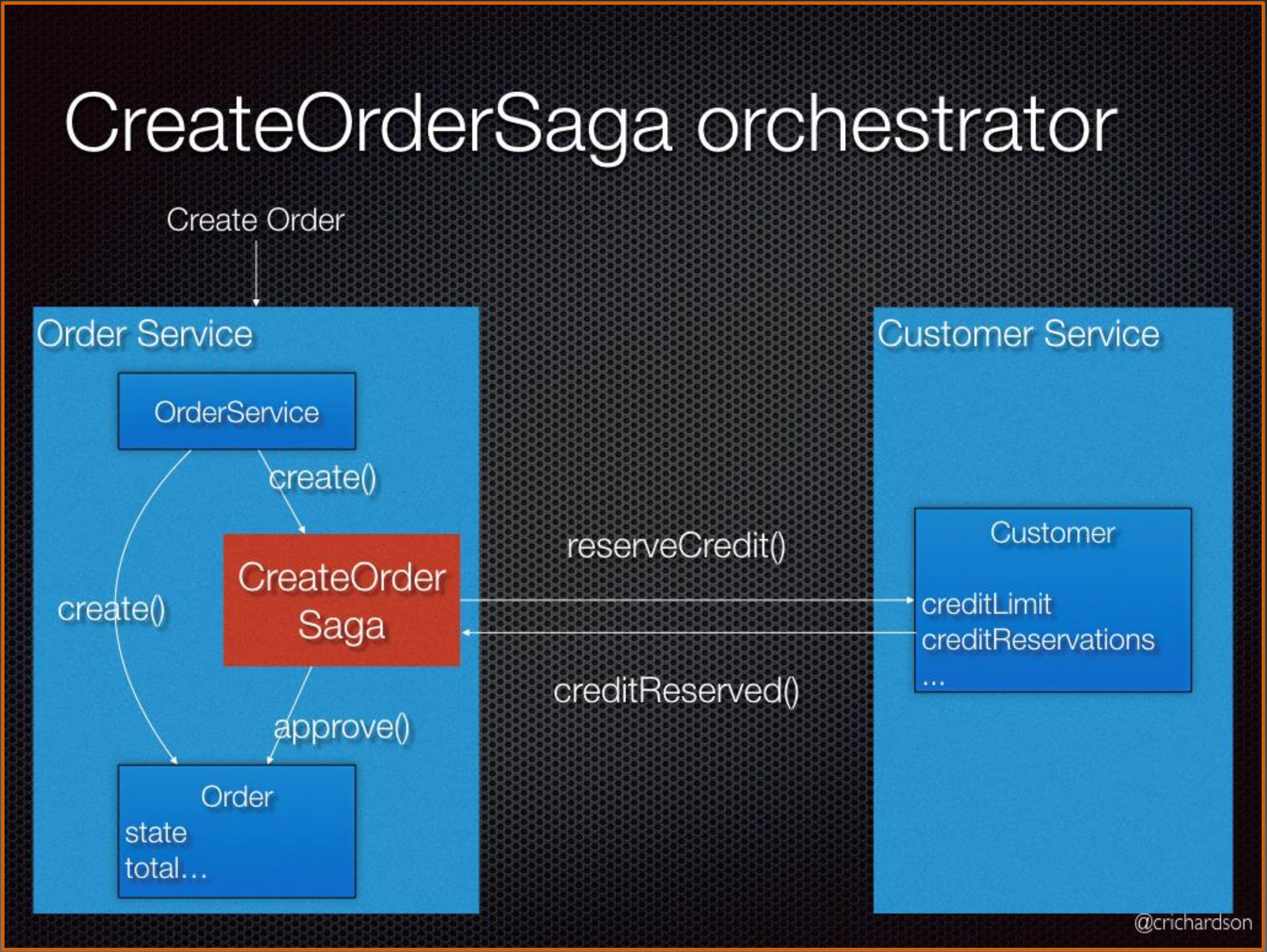
Choreography

1. The Order Service creates an Order in a pending state and publishes an OrderCreated event
2. The Customer Service receives the event attempts to reserve credit for that Order. It publishes either a Credit Reserved event or a CreditLimitExceeded event.
3. The Order Service receives the event and changes the state of the order to either approved or cancelled


Option #1: Choreography-based coordination using events

Orchestration

1. The Order Service creates an Order in a pending state and creates a CreateOrderSaga
2. The CreateOrderSaga sends a ReserveCredit command to the Customer Service
3. The Customer Service attempts to reserve credit for that Order and sends back a reply
4. The CreateOrderSaga receives the reply and sends either an ApproveOrder or RejectOrder command to the Order Service
5. The Order Service changes the state of the order to either approved or cancelled



Summary

- A transaction is a sequence of operations performed as a single logical unit of work
- Per the CAP theorem, distributed systems must balance consistency with availability when partitioned
- Eventual consistency implies that the state of an object at any given moment may be viewed differently in different parts of a distributed system but will eventually converge
- Etags are a web centric means of ensuring that one only updates a remote object if the object is in the expected pre-state
- Event Sourcing is a means to decouple state managers by generating broadcast events associated with all state changes
- CQRS – Command Query Responsibility Separation is a process where expensive state change commands are queued and fast/cheap state queries are synchronous

Lab 5

- Create a message driven loosely coupled service

6: Stateless Services and Polyglot Persistence

Objectives

- Describe the range of state managers are available for use in microservice based systems
- Explain the differences between various database types
  - Relational
  - Graph
  - Key/Value
  - Document
  - Column
- Explore the importance of Schemas

Relational Databases

- Relational databases still manage most of the world's critical data
- ACID Transactions
- Perfect for shared database integration [Hohpe and Woolf]
  - One database hosting data for a range of applications
- Single Source of Truth
- Years of refinement and evolution
  - No data platform type is more widely supported
- One system to manage
  - Concurrency is notoriously difficult to get right
  - Errors can trap even the most careful programmers
  - Relational systems scale vertically, sidestepping many concurrency concerns and managing most of the rest for you

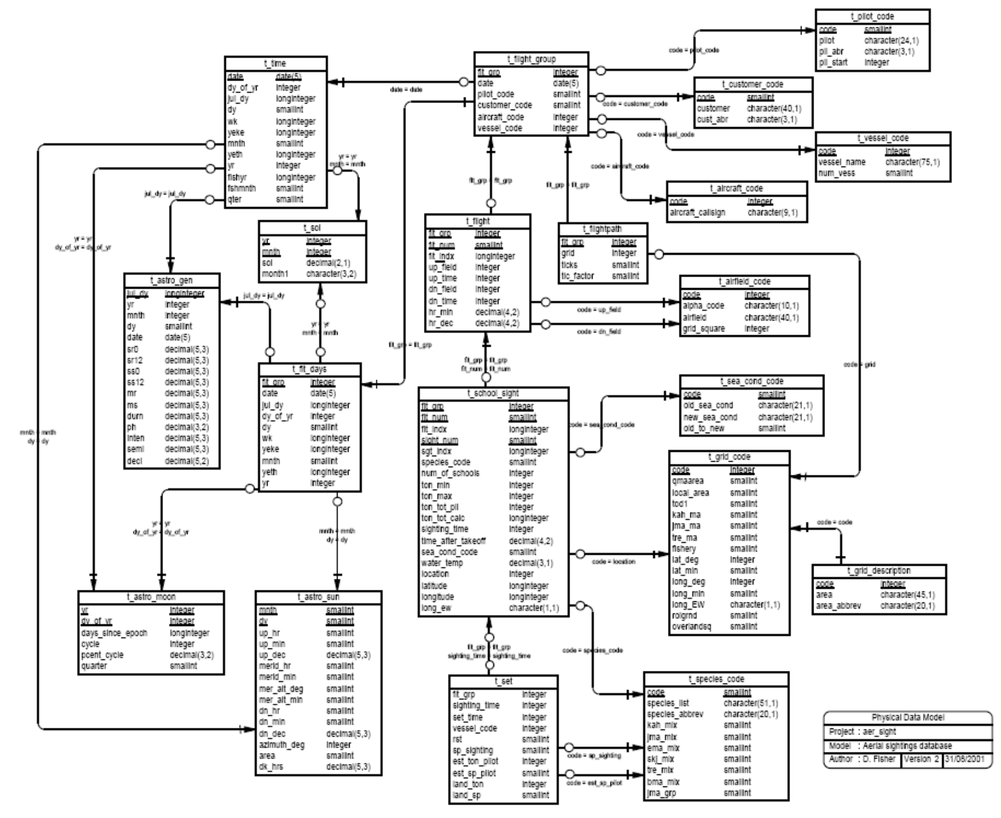
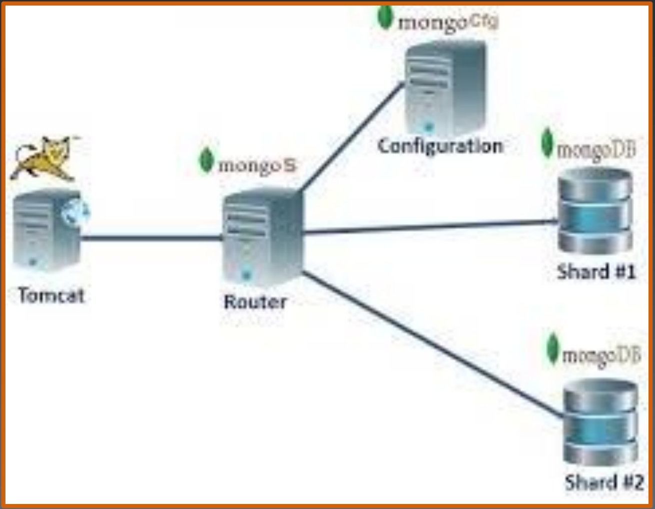


Figure 1: Entity Relationship Diagram (ERD) for the aer_sight database

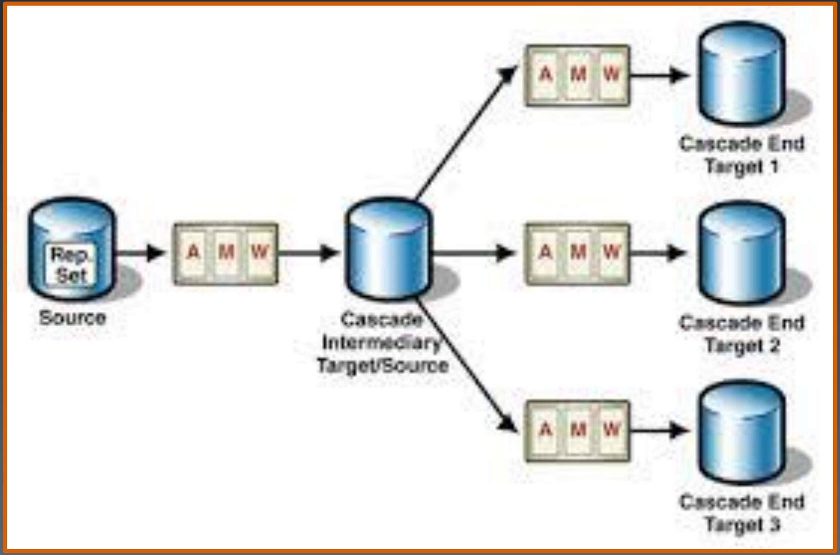
Database Scale Out

- There are two basic means used to distributed data
  - Replication
    - Copies the same data to multiple nodes
  - Partitioning
    - Vertical – sets of columns are placed on different nodes
    - Horizontal (aka Sharding) – sets of rows are placed on different nodes
- Replication and Partitioning can be combined to create an array of data distribution patterns
- Single server databases are the easiest to manage
  - Not everyone is Google
  - If a quality server and a fast database can handle your data you may be much better off keeping things simple
    - Single server solutions run on reliable hardware (not commodity)
    - Redundant power supplies and network interfaces, RAID disks, monitoring hardware, robust manufacturing QA, etc.
  - In memory cache and hot standby solutions allow single server models to run fast and reliably
  - Distributed systems create unwanted overhead, don't do it if you don't have to!
- If you plan to shard do so early and during a slow time (when you have plenty of head room)
  - Enabling partitioning on many systems will cause heavy data rebalancing and replication traffic



Replication Benefits

- Read Scale: replication enables scale out architectures to be designed involving a master server and many replicas sharing application read load
- Availability: system redundancy imparted by replication allows for highly available solutions
- Geographic Distribution: Replicas can be placed in geographically diverse locations, close to their user
- Task Offloading: Replicas present excellent platforms for backups, analytics, application development and testing environments



Why NoSQL

- Impedance mismatch
  - the difference between the relational model and the in-memory data structures is often substantial
  - This requires data translation
  - Relational data is organized into tables and rows (relations and tuples)
    - A tuple is a set of name -value pairs and a relation is a set of tuples
  - SOA services use richer data structures with nested records and lists
    - Represented as documents in XML, JSON, etc.
  - Reducing the number of server round trips makes a rich structure desirable
- Relational systems (due to joins and ACID) are difficult to scale horizontally
  - Join performance and highly structured data also makes them ill suited for today's "big data" environments
- A cluster of small machines can use commodity hardware
  - Scaling out
  - Cheaper
  - More resilient
- NoSQL systems process aggregates not tuples and are typically designed for clustering and unstructured data
  - Good support for nonuniform data
- SQL is a powerful and well understood language
  - Requires significant complexity in the underlying platform
  - Not present in NoSQL platforms
  - SQL like solutions are present on many NoSQL solutions
    - Hive
    - Cassandra CQL

NoSQL Data Models

NoSQL Attributes:
- Not using the relational model
- Running well on clusters
- Open-source
- Built for the 21st century web estates
- Schemaless

KeyValue
- Redis
- Memcached
- Riak
- BerkeleyDB
- Hazelcast
- LevelDB

Document
- AWS DynamoDB
- MongoDB
- CouchDB
- Couchbase
- MarkLogic
- OrientDB
- ArangoDB
- RavenDB

Column
- Cassandra
- AWS SimpleDB
- HBase
- Accumulo
- Hypertable

Graph
- Neo4J
- FlockDB
- Virtuoso
- Giraph
- Neptune
- Titan (can use DynamoDB as backend)
- OrientDB
- ArangoDB

AWS Solutions in orange

NoSQL Considerations

- NoSQL databases have no explicit schema
  - Schemaless databases have implicit schemas
  - Changing the structure of data during the life of an application has an impact on the application
- Many NoSQL databases operate on aggregates
  - An aggregate may be a document, column set, block, etc.
- NoSQL databases work best as application databases
  - Application databases serve a single application or application component
  - Application databases are often wrapped in SOA services
  - A strength of NoSQL solutions is their ability to map directly to application constructs
  - NoSQL databases do not always work well in multitenant (application) scenarios where more general solutions are more broadly applicable
  - Polyglot Persistence using different data stores in different circumstances
- Choosing the best NoSQL solution involves:
  - Selecting a programming model and choosing a well aligned data storage model
  - Selecting the simplest storage solution that provides the scale, performance and growth potential required

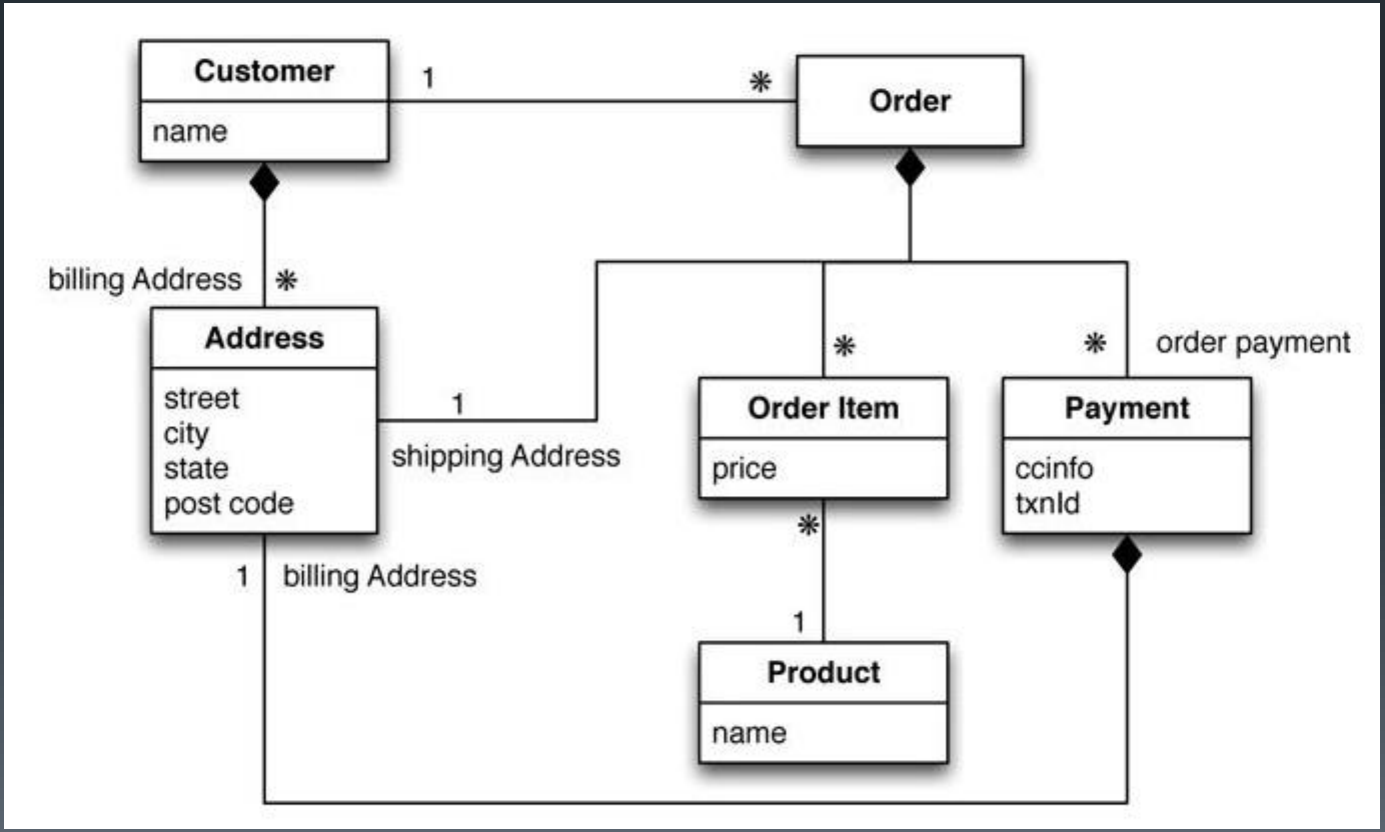
There are two primary reasons for considering NoSQL

- To handle data access with sizes and performance that demand a cluster
- To improve the productivity of application development by using a more convenient data interaction style

--Fowler 2012

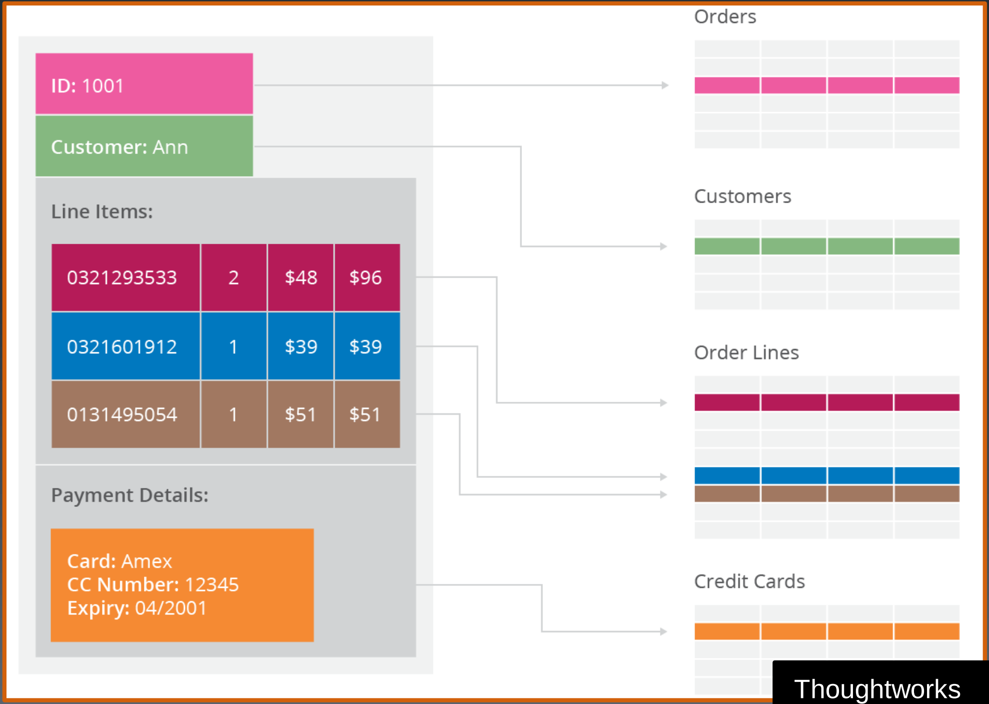
Aggregates

- In Domain-Driven Design [Evans], an aggregate is a collection of related objects that we wish to treat as a unit
- Choosing aggregates is an important part of the design process in distributed applications
- Where are the aggregate boundaries?
- Which datum will be repeated if any?
- How can aggregates be structured to minimize server round trips?



NoSQL Transactions

- ACID transactions offer the pinnacle of consistency
- Most NoSQL database offer no transaction support
  - Graph databases are the exception
- However, transaction-less NoSQL systems offer atomic aggregate updates
- If aggregates equate to joined rows in SQL similar transaction semantics can be achieved



Thoughtworks

Consistency

- Highly distributed clusters introduce new consistency concerns as compared to Relational ACID type systems
  - **write-write conflict**: two people updating the same data item at the same time
    - Can produce **Lost Updates**
    - We both increment 6 and instead of going to 8 it goes to 7
  - **sequential consistency:** ensuring that all nodes apply operations in the same order
- Systems can take a pessimistic or optimistic approach to consistency
  - pessimistic prevents conflicts from occurring
  - optimistic lets conflicts occur, but detects them and takes action to sort them out (hopefully?)
- Conditional updates test the value just before updating it to see if it's changed since his last read
- Some systems save all write-write conflict updates allowing users (or application code) to resolve the conflict
- Any update that affects multiple aggregates leaves an inconsistency window
  - Amazon SimpleDB reports an inconsistency windows usually less than 1 second
- replication consistency is the process of ensuring that the same data item has the same value when read from different replicas
  - Quorums
  - Eventual Consistency
- read-your-writes consistency means that, once you've made an update, you're guaranteed to continue seeing that update
- session consistency a user's session ensures read-your-writes consistency

Sticky Session

- A session tied to one node (aka. session affinity)
- ensures that as long as you keep read-your-writes consistency on a node, you'll get it for sessions too
- sticky sessions reduce the ability of the load balancer to do its job

Quorums

- write quorum
  - $W > N/2$
  - The number of nodes participating in the write (W) must be more than the half the number of nodes involved in replication (N)
  - The number of replicas is often called the replication factor
- read quorum
  - How many nodes you need to contact to be sure you have the most up-to-date change
  - You can have a strongly consistent read if $R + W > N$
- These inequalities are written with a peer-to-peer distribution model in mind
- Most suggest a replication factor of 3
  - This allows a single node to fail while still maintaining quora for reads and writes
  - With automatic rebalancing the cluster will create a third replica quickly
- The number of nodes participating in an operation vary with the operation
  - When writing a quorum may be required for some types of updates but not others, depending on the value of consistency and availability

Key Value Model

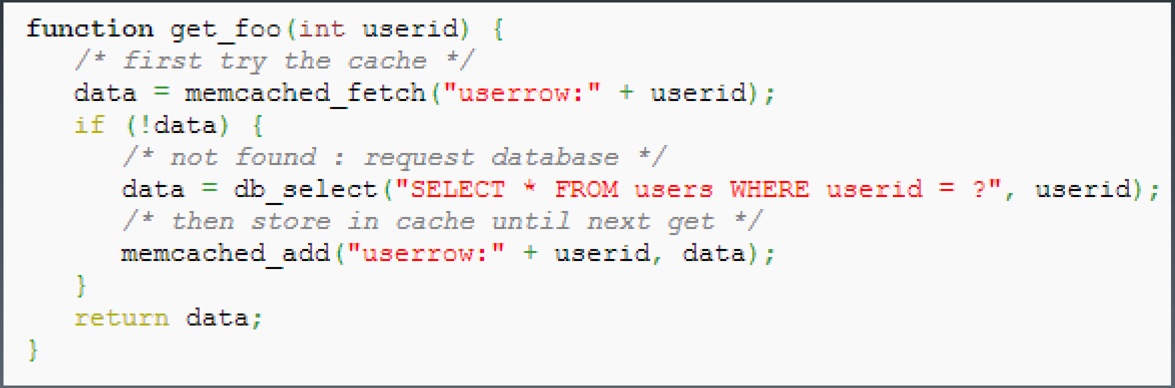
- Each aggregate has a key or ID that's used to get the data
- The aggregate is opaque to the database
  - Usually seen as a blob
  - Aggregates can be anything
  - Usually size limited
- Some dbs support meta data to define relationships, expiration times, etc.
- These solutions are very simple, and thus tend to be very fast
- Sharding is based on key hash

Use:
- Storing Session Info
- User profiles/preferences
- Shopping carts

Don't Use:
- Data Relationships
- Multioperation Transactions
- Query by Data
- Operations on sets

Memcached

- A general-purpose distributed memory K/V caching system
- Used to speed up dynamic database/API driven websites
- Runs on Unix, Linux, Windows and Mac OS X
- Keys are up to 250 bytes long and values can be at most 1 megabyte
- The client library knows all servers
  - Servers do not communicate with each other
  - Clients read/set values by computing a hash of the key to determine the server to use
- When the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order
- Applications using Memcached typically fall back to a database request on cache miss
- Originally developed by Danga Interactive for LiveJournal, now used by YouTube, Reddit, Zynga, Facebook, Twitter, Tumblr, Wikipedia, etc.
- Engine Yard and Jelastic are using Memcached as the part of their platform as a service technology stack
- Heroku offers a managed Memcached service built on Couchbase Server as part of their platform as a service
- Google App Engine, AppScale, Windows Azure and Amazon Web Services also offer Memcached



```
function get_foo(int userid) {
   /* first try the cache */
   data = memcached_fetch("userrow:" + userid);
   if (!data) {
      /* not found : request database */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);
      /* then store in cache until next get */
      memcached_add("userrow:" + userid, data);
   }
   return data;
}
```

Document Model

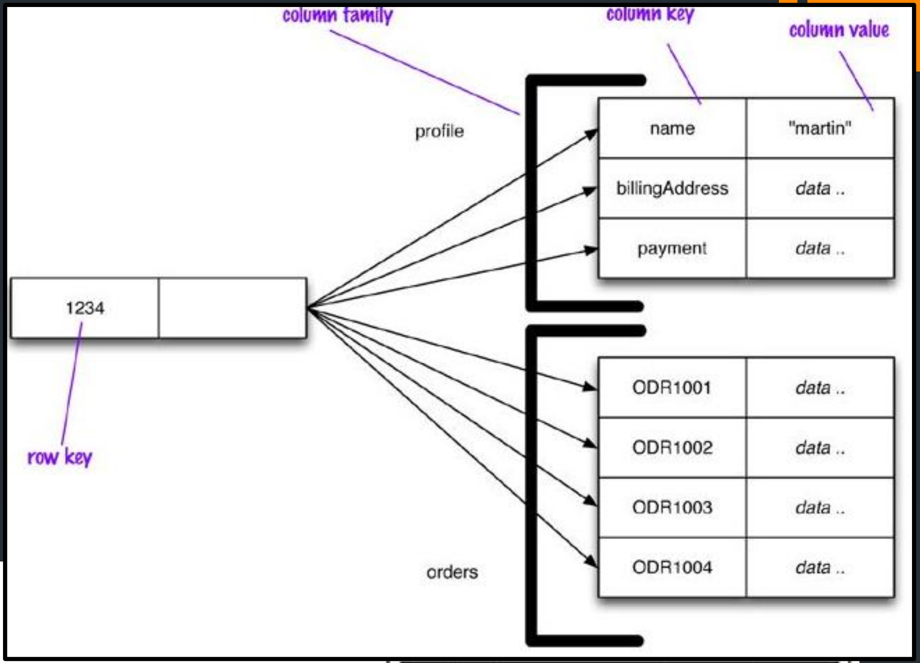
- Each aggregate has a key or ID that's used to get the data
- The aggregate is visible to the database
  - Usually a JSON or XML document
  - Aggregates can have any structure support by the document type
- Document dbs support limited searches on document data
- These solutions are simple, fast but supply useful aggregate search features
  - You can look up data by something other than aggregate key
- Sharding is based on key hash
- In practice the distinction between KeyValue and Document dbs blurs heavily in some implementations

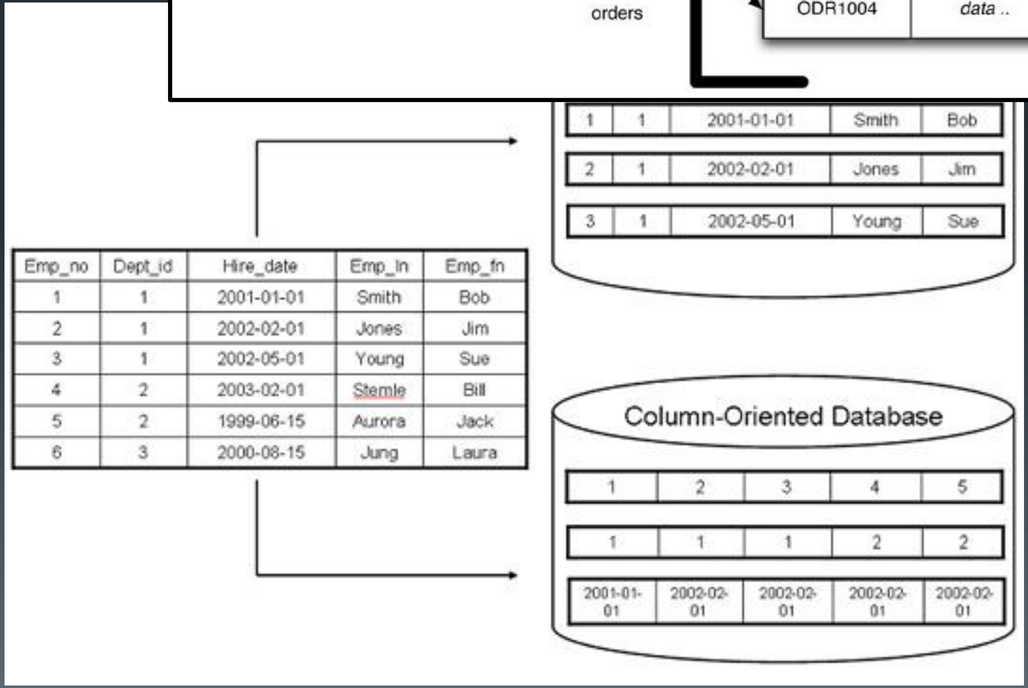
Use:
- Event Logging
- CMS/Blogging
- Web Analytics
- E-Commerce

Don't Use:
- Complex transactions
- Queries against varying aggregate structure

Column

- Google documented BigTable in a famous white paper [Chang etc.]
  - Their core data store at the time (previously used for search)
  - Ues sparse columns and no schema
  - A two-level map
  - Influenced HBase and Cassandra
- Column Families
  - Stores groups of columns together
  - Each column has to be part of a single column family
  - Columns are the unit for access
  - Column stores make summing columns very fast due to column layout on disc versus row layout
  - Column families can be treated like tables
- Columns can be added freely
- Skinny Rows
  - Few columns
  - Same columns used across the many rows
  - The column family defines a record type
  - Each row is a record, and each column is a field
- Wide Rows
  - Have many columns (perhaps thousands), with rows having very different columns
  - A wide column family models a list, with each column being one element in that list
  - A consequence of wide column families is that a column family may define a sort order for its columns





1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue

Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura

Column-Oriented Database

1	2	3	4	5
1	1	1	2	2
2001-01-01	2002-02-01	2002-02-01	2002-02-01	2002-02-01

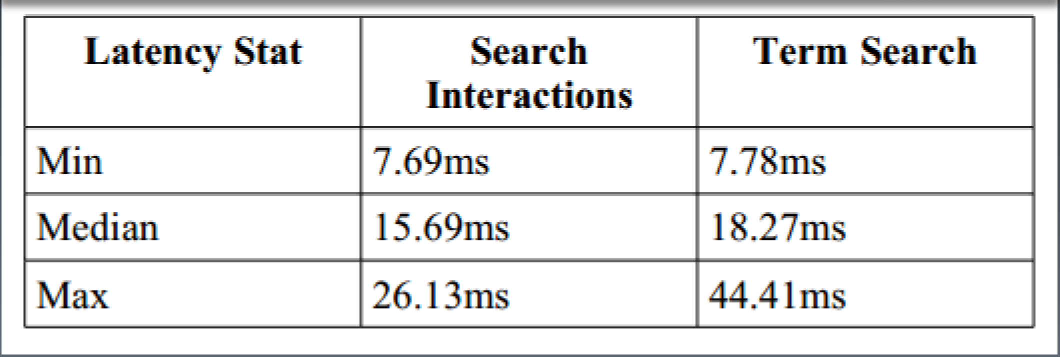
Use:
- Event Logging
- CMS/Blogging
- Counters
- Expiring usage (columns)

Don’t Use:
- Situations where the column families change frequently

Cassandra Case Study

- Invented at Facebook by the author of Amazon's DynamoDB, Apache Cassandra is an open source distributed database management system designed to handle large amounts of data across many commodity servers
  - Cassandra's distributed architecture is specifically tailored for multiple-data center deployment, for redundancy, for failover and disaster recovery
- Cassandra is implemented as a DHT producing high availability with no single point of failure
- Cassandra implements asynchronous masterless replication
- University of Toronto researchers studying NoSQL systems report that Cassandra achieved the highest throughput for the maximum number of nodes in all of their experiments
- Based on the Chord DHT and uses an MD5 128 bit hash key
- Cassandra's data model is a partitioned row store with tunable consistency
  - Partition keys can be hashed randomly or in order for systems benefitting from partition key locality
  - Rows are organized into tables
  - The first component of a table's primary key is the partition key
  - Within a partition, rows are clustered by the remaining columns of the key
  - Other columns may be indexed separately from the primary key
  - Tables may be created, dropped, and altered at runtime without blocking updates and queries
- Cassandra does not support joins or subqueries, except for batch analysis via Hadoop
- Cassandra emphasizes denormalization through features like collections
- Memcached competitive performance without a two level architecture

Facebook statistics for a >50TB, 150 node cluster



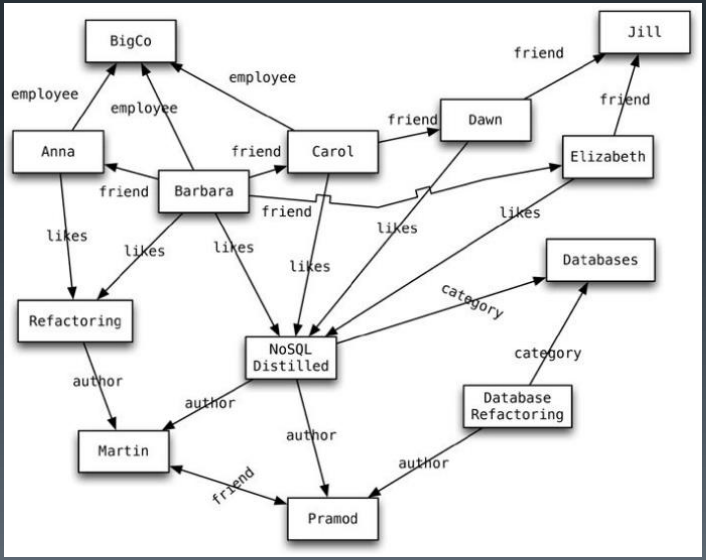
Latency Stat	Search Interactions	Term Search
Min	7.69ms	7.78ms
Median	15.69ms	18.27ms
Max	26.13ms	44.41ms

Graph

- Graph databases are motivated by a different frustration with relational databases and thus have an opposite model
  - small records with complex interconnections
- Nodes & Edges (aka. Arcs)
- Queries like:
  - Select books in the Databases category written by someone whom a friend of mine likes
- Edge traversal is cheap
  - Graph databases shift most of the work of navigating relationships from query time to insert time
  - Good only if querying performance is more important than insert speed
- Usually single server based
- Examples
  - FlockDB - nodes and edges with no mechanism for additional attributes
  - Neo4J - attach Java objects as properties to nodes and edges
  - Infinite Graph - stores Java objects, which are subclasses of its built-in types, as nodes and edges

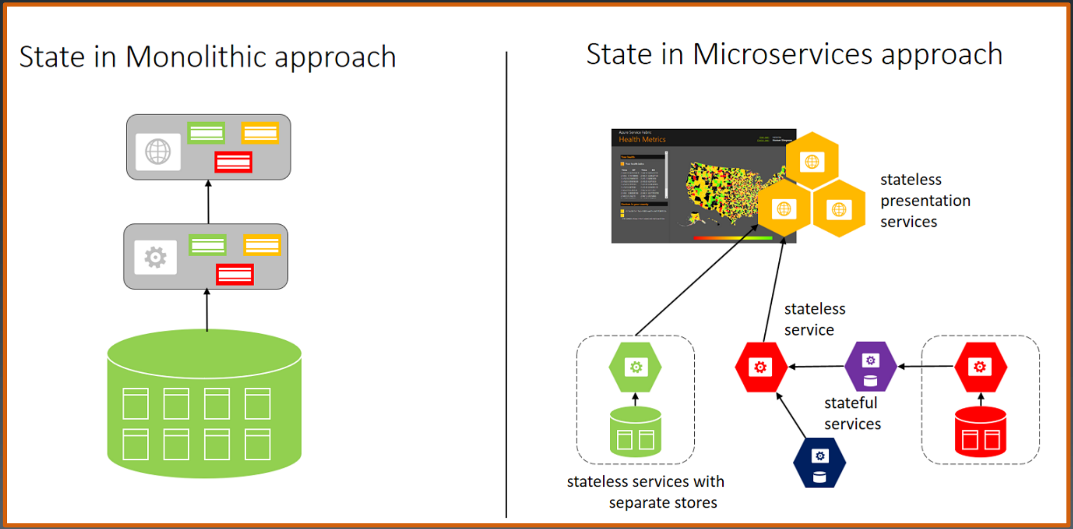
Use:
- Connected Data (social graphs)
- Routing/Dispatch (location based systems)
- Recommendation Engines

Don't Use:
- Situations where multiple nodes must be updated



Cloud native state management, patterns and practices

- Cloud native applications divided microservices into 2 camps
  - Stateless
  - Stateful
- Stateful services use volumes to store durable data
- State managers are typically cloud native cluster aware systems
  - Redis, MongoDB, Cassandra, etc.
- State managers are owned by a single service and live inside the bounded context of that service


State in Monolithic approach
State in Microservices approach
Health Metrics
stateless presentation services
stateless service
stateful services
stateless services with separate stores

Summary

- A range of state managers are available for use in microservice based systems
  - Relational
  - Graph
  - Key/Value
  - Document
  - Column
- Schemas are critical components of most data stores and should be documented
- Aggregates describe the elements of storage in NoSQL systems
- A range of consistency tuning features are available with different storage solutions

Lab 6

- Building a stateful service