Cloud Design Patterns[1]

Mikael Svahnberg[1]

[1]Mikael.Svahnberg@bth.se
School of Computing
Blekinge Institute of Technology

May 7, 2015

[1]Creatively borrowed from B. Wilder, *"Cloud Architecture Patterns"*, O'Reilly, 2012.

Scalability

- (Flexible) Scalability is one of the core features of Cloud Computing
- Vertical Scaling (increase capacity per node)
- Horizontal Scaling (adding nodes)

Measures of Scalability

A combination of:

- Concurrent Users
- Response Time
- Processed items / time unit
- Complexity of processing requests

Issues that influence scalability

Scarce resources:

- Computing power
- RAM
- Storage space
- Network bandwidth

Scaling Mindsets

- Cetain mindsets help in addressing Cloud Scaling
- These do not affect the architecture *per se*, but influences your choices of solutions.
  - Eventual Consistency
  - Multitenancy
  - Inevitable Failure
  - Network Latency

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Eventual Consistency

- *"At any moment, most of an eventially consistent database is consistent, with somne small number of values still being updated"*
- CAP theorem: {Consistency, Availability, Partition Tolerance}: *Pick two!*
- This implies:
  - Data is always(?) available, although not always 100% correct
  - Your system needs to robustely deal with this.
- Compare with RDBMS' *ACID* property.
- With a distributed database (using e.g. a NoSQL database), you instead have *BASE*:
  - Basically Available
  - Soft State
  - Eventually Consistent

Multitenancy

- One company (host) operates the application for use by other companies (tenants)
- The tenants have the impression that they are alone in using the service
- Has implications on:
  - data partitioning
  - security
  - performance management
- As much a concern for the cloud provider as for the cloud application provider.

Inevitable Failure

- The cloud provider is likely to use cheap *commodity hardware*
- Therefore, hardware failure is inevitable (although not necessarily frequent)
- This implies that your application need to be *robust*
- Focus shift from MTBF to MTTR

Network Latency

- Problem: You are *here*, your data are *there*, and your users are *yonder*
- . . . And the servers running your applications are neither *here nor there*.
- Moving data between your servers and, eventually, to the users requires network bandwidth
- Strategies your application may take:
  - Data compression
  - Background processing
  - Predictive fetching
  - Moving your application closer to the users
  - Moving the data closer to the users
  - Moving data processing nodes closer together

Scalability Patterns

- Generic Scalability:
  - Horizontally Scaling Compute Pattern
  - Queue-Centric Workflow Pattern
  - Auto-Scaling Pattern
- Eventual Consistency
  - MapReduce Pattern
  - Database Sharding Pattern
- Multitenancy and Inevitable Failure
  - Busy Signal Pattern
  - Node Failure Pattern
- Network Latency
  - Colocate Pattern
  - Valet Key Pattern
  - CDN Pattern
  - Multisite Deployment Pattern

Challenges with Horizontal Scaling

- Load Balancing
- Synchronisation between nodes
- Managing Session State
  - Sticky sessions? (What happens if that node breaks?)
  - Cookies (for small amounts of data)
  - Cookies (as a key to the full db record)
- Capacity Planning (per time unit)
- Sizing the virtual machines

Queue Centric Workflow

- Problem: Some jobs take longer time. This may impact the responsiveness of the application.
- Solution:
  - Package the tasks to do in a job description and add it to a queue.
  - Worker(s) in the service tier picks work from the job queue and processes them in due order.
  - cf. Eventual Consistency
- Challenges:
  - One worker picks a job but fails halfway through.
  - Solutions: Invisibility window, idempotent processing (for repeat messages), handling of poison messages

MapReduce

- Map: execute function on each instance of the data
- Reduce: merge the results of a map into a combined and consistent data set again.
- Challenges:
  - moving big data takes time and is expensive.
  - Solution: “bring the compute to the data”

Database Sharding

- Classic database division: Vertical
- For example: db{users}, db{orders}, db{warehouse}, . . .
- Sharding divides the data horizontally.
- All db instances have the entire schema, and contains a subset of all the rows.
- One db row only exist in one db.
- Challenges:
  - Deciding how to shard your data to be most efficient
  - Cloudfronting (where should a particular row be?)
  - Defining the shards to minimise database queries over several shards, or shards far away.

Node Failure

Scenario	Warning	Impact
Sudden failure	no	local data is lost
Platform shutdown/restart	yes	local data may be available
Application shutdown/restart	yes	local data is available
Shutdown/destroy	yes	local data is lost

Advice:

- Treat all interruptions as node failures
- Maintain sufficient capacity for failure *(N+1 rule)*
- Load balancing to minimise interruption for user
- Combine with other patterns, e.g. queue-centric workflow pattern, busy signal pattern

Summary

- There are a few mindsets that influence your choices of solutions for a cloud application
  - Eventual Consistency
  - Multitenancy
  - Inevitable Failure
  - Network Latency
- Related to these mindsets, there are a number of design patterns for addressing them in a cloud setting.
- Some of these are mentioned in this lecture.
- Some of these are discussed in some further detail.
- As usual, when discussing architecture and design patterns, the details are not available until you design *your specific application*.