The CAP theorem and the design of large scale distributed systems: Part I

Silvia Bonomi
University of Rome “La Sapienza”
www.dis.uniroma1.it/~bonomi

Great Ideas in Computer Science & Engineering
A.A. 2012/2013
A little bit of History

- Mainframe-based Information Systems
- First Internet-based systems for military purpose
- Web-services based Information Systems
- Client-server architectures
- Wireless and Mobile ad-hoc networks
- Peer-to-peer Systems
- Cloud Computing Platforms
- Wireless and Mobile ad-hoc networks
- Peer-to-peer Systems
- Cloud Computing Platforms
Relational Databases History

- Relational Databases – mainstay of business
- Web-based applications caused spikes
  - Especially true for public-facing e-Commerce sites
- Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application
Scaling Up

- Issues with scaling up when the dataset is just too big
- RDBMS were not designed to be distributed
- Began to look at multi-node database solutions
- Known as ‘scaling out’ or ‘horizontal scaling’
- Different approaches include:
  - Master-slave
  - Sharding
Scaling RDBMS – Master/Slave

- **Master-Slave**
  - All writes are written to the master. All reads performed against the replicated slave databases.
  - Critical reads may be incorrect as writes may not have been propagated down.
  - Large data sets can pose problems as master needs to duplicate data to slaves.
Scaling RDBMS - Sharding

- Partition or sharding
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards
Other ways to scale RDBMS

- Multi-Master replication
- INSERT only, not UPDATES/DELETES
- No JOINs, thereby reducing query time
  - This involves de-normalizing data
- In-memory databases
Today…

Microsoft

Linkedin

Google

Amazon

Facebook

IBM

Twitter

Yahoo!
Context

- Networked Shared-data Systems
Fundamental Properties

- **Consistency**
  - (informally) “every request receives the right response”
  - E.g. If I get my shopping list on Amazon I expect it contains all the previously selected items

- **Availability**
  - (informally) “each request eventually receives a response”
  - E.g. eventually I access my shopping list

- **tolerance to network Partitions**
  - (informally) “servers can be partitioned in to multiple groups that cannot communicate with one another”
CAP Theorem

- 2000: Eric Brewer, PODC conference keynote
- 2002: Seth Gilbert and Nancy Lynch, ACM SIGACT News 33(2)

“Of three properties of shared-data systems (Consistency, Availability and tolerance to network Partitions) only two can be achieved at any given moment in time.”
Proof Intuition

Networked Shared-data system

Write (v1, A)

Read (A)
Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

**Claim**: every distributed system is on one side of the triangle.

- **CA**: available, and consistent, unless there is a partition.
- **AP**: a reachable replica provides service even in a partition, but may be inconsistent.
- **CP**: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).
The CAP Theorem

**Theorem:** You can have at most two of these invariants for any shared-data system.

**Corollary:** consistency boundary must choose A or P.
Forfeit Partitions

Examples
- Single-site databases
- Cluster databases
- LDAP
- Fiefdoms

Traits
- 2-phase commit
- cache validation protocols
- The “inside”
Observations

- CAP states that in case of failures you can have at most two of these three properties for any shared-data system
  - To scale out, you have to distribute resources.
    - P in not really an option but rather a need
    - The real selection is among consistency or availability
    - In almost all cases, you would choose availability over consistency
Forfeit Availability

- Consistency
- Availability
- Tolerance to network Partitions

Examples
- Distributed databases
- Distributed locking
- Majority protocols

Traits
- Pessimistic locking
- Make minority partitions unavailable
Forfeit Consistency

- Consistency
- Availability
- Tolerance to network partitions

Examples
- Coda
- Web caching
- DNS
- Emissaries

Traits
- Expirations/leases
- Conflict resolution
- Optimistic
- The “outside”
Consistency Boundary Summary

- We can have consistency & availability within a cluster.
  - No partitions within boundary!

- OS/Networking better at A than C

- Databases better at C than A

- Wide-area databases can’t have both

- Disconnected clients can’t have both
CAP, ACID and BASE

- **BASE** stands for Basically Available Soft State Eventually Consistent system.

- **Basically Available**: the system available most of the time and there could exist a subsystem temporarily unavailable.

- **Soft State**: data are “volatile” in the sense that their persistence is in the hand of the user that must take care of refresh them.

- **Eventually Consistent**: the system eventually converge to a consistent state.
CAP, ACID and BASE

- Relation among ACID and CAP is core complex

- **Atomicity**: every operation is executed in “all-or-nothing” fashion

- **Consistency**: every transaction preserves the consistency constraints on data

- **Integrity**: transaction does not interfere. Every transaction is executed as it is the only one in the system

- **Durability**: after a commit, the updates made are permanent regardless possible failures
CAP, ACID and BASE

**CAP**

- C here looks to single-copy consistency
- A here look to the service/data availability

**ACID**

- C here looks to constraints on data and data model
- A looks to atomicity of operation and it is always ensured
- I is deeply related to CAP. I can be ensured in at most one partition
- D is independent from CAP
Warning!

- **What CAP says:**
  - When you have a partition in the network you cannot have both C and A

- **What CAP does not say:**
  - There could not exist a time period in which you can have both C, A and P

- During Normal Periods (i.e. period with no partitions) both C and A can be achieved
2 of 3 is misleading

- Partitions are rare events
  - there are little reasons to forfeit by design C or A

- Systems evolve along time
  - Depending on the specific partition, service or data, the decision about the property to be sacrificed can change

- C, A and P are measured according to continuum
  - Several level of Consistency (e.g. ACID vs BASE)
  - Several level of Availability
  - Several degree of partition severity
2 of 3 is misleading

- In principle every system should be designed to ensure both C and A in normal situation

- When a partition occurs the decision among C and A can be taken

- When the partition is resolved the system takes corrective action coming back to work in normal situation
Consistency/Latency Trade Off

- CAP does not force designers to give up A or C but why there exists a lot of systems trading C?

- CAP does not explicitly talk about latency…
- … however latency is crucial to get the essence of CAP
Consistency/Latency Trade Off

- High Availability is a strong requirement of modern shared-data systems

- To achieve High Availability, data and services must be replicated

- Replication impose consistency maintenance

- Every form of consistency requires communication and a stronger consistency requires higher latency
PACELC

Abadi proposes to revise CAP as follows:

“PACELC (pronounced pass-elk): if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?”
Partitions Management

The figure shows a partition’s evolution. Normal operation is represented by consecutive states, starting from state $S$ and progressing through $S_1$, $S_2$, and leading to $S'$. The partition starts when an operation initiates partition recovery. Once the system enters partition mode, two strategies can be employed:

1. Limit some operations, thereby reducing availability. The second strategy is to record extra information about the operations that will be helpful during partition recovery.

Partition mode starts between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. If a partition does indeed exist, both sides enter partition mode and continue to execute operations, creating a sequence of consistent operations.

When partition recovery is required, either this side responds correctly or no communication is required; either way, operations remain consistent. The last step aims to restore consistency and compensate for any mistakes made during partitioning. The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies between operations. For an invariant that must be satisfied for mistakes the program made while the system was partitioned.

Researchers often use an offline mode, such as Google Docs, for such cases. The strategy is to record the intent and execute it after the recovery. Such transactions are typically part of a larger workflow that has an explicit order-processing aspect. When a user sees a partially updated state, and there is little downside to delaying the operation, the designer forfeits A in a way that users do not see. The users know only that they placed an order and that the system will execute it later.

For an invariant that must be maintained during a partition, designers typically decide to risk violating it with the intent of restoring it during recovery, and, assuming that they can be merged, the designer can easily restore the invariant. Keys in a table are unique, and keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged, the designer can easily restore the invariant. Keys in a table are unique, and keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged, the designer can easily restore the invariant.

Detection
Activating Partition Mode
Partition Recovery
Partition Detection

- CAP does not explicitly talk about latencies

- However...
  - To keep the system live time-outs must be set
  - When a time-out expires the system must take a decision

```
NO, continue to wait

Possible Availability Loss

Is a partition happening?

YES, go on with execution

Possible Consistency Loss
```
Partition Detection

- Partition Detection is not global
  - An interacting part may detect the partition, the other not.
  - Different processes may be in different states (partition mode vs normal mode)

- When entering Partition Mode the system may
  - Decide to block risk operations to avoid consistency violations
  - Go on limiting a subset of operations
Which Operations Should Proceed?

- Live operation selection is an hard task
  - Knowledge of the severity of invariant violation
  - Examples
    - every key in a DB must be unique
      - Managing violation of unique keys is simple
      - Merging element with the same key or keys update
    - every passenger of an airplane must have assigned a seat
      - Managing seat reservations violation is harder
      - Compensation done with human intervention
  - Log every operation for a possible future re-processing
Partition Recovery

- When a partition is repaired, partitions’ logs may be used to recover consistency

- Strategy 1: roll-back and executed again operations in the proper order (using version vectors)

- Strategy 2: disable a subset of operations (Commutative Replicated Data Type - CRDT)
Basic Techniques: Version Vector

- In the version vector we have an entry for any node updating the state
- Each node has an identifier
- Each operation is stored in the log with attached a pair `<nodeId, timeStamp>`

- Given two version vector A and B, A is newer than B if
  - For any node in both A and B, $ta(B) \leq ts(A)$ and
  - There exists at least one entry where $ta(B) < ts(A)$
Version Vectors: example

\[
\begin{array}{cc}
1 & 1 \\
0 & 1 \\
0 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
ts(B) & Ts(A) \\
1 & 0 \\
0 & 0 \\
0 & 1 \\
\end{array}
\]

\[
\text{ts}(A) < \text{ts}(B) \text{ then } A \rightarrow B
\]

\[
\begin{array}{cc}
ts (A) \neq ts (B) \text{ then } A \parallel B \\
\text{POTENTIALLY INCONSISTENT!} \\
\end{array}
\]
Basic Techniques: Version Vector

- Using version vectors it is always possible to determine if two operations are causally related or they are concurrent (and then dangerous)

- Using vector versions stored on both the partitions it is possible to re-order operations and raising conflicts that may be resolved by hand

- Recent works proved that this consistency is the best that can be obtained in systems focussed on latency
Basic Techniques: CRDT

- Commutative Replicated Data Type (CRDT) are data structures that provably converges after a partition (e.g. set).

- Characteristics:
  - All the operations during a partition are commutative (e.g. add(a) and add(b) are commutative) or
  - Values are represented on a lattice and all operations during a partition are monotonically increasing wrt the lattice (giving an order among them)
    - Approach taken by Amazon with the shopping cart.
  - Allows designers to choose A still ensuring the convergence after a partition recovery
Basic Techniques: Mistake Compensation

- Selecting A and forfailing C, mistakes may be taken
  - Invariants violation

- To fix mistakes the system can
  - Apply deterministic rule (e.g. “last write win”)
  - Operations merge
  - Human escalation

- General Idea:
  - Define specific operation managing the error
    - E.g. re-found credit card
What is NoSQL?

- Stands for Not Only SQL
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties (will talk about the CAP theorem)
Why NoSQL?

- For data storage, an RDBMS cannot be the be-all/end-all
- Just as there are different programming languages, need to have other data storage tools in the toolbox
- A NoSQL solution is more acceptable to a client now than even a year ago
How did we get here?

- Explosion of social media sites (Facebook, Twitter) with large data needs
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Just as moving to dynamically-typed languages (Ruby/Groovy), a shift to dynamically-typed data with frequent schema changes
- Open-source community
Dynamo and BigTable

- Three major papers were the seeds of the NoSQL movement
  - BigTable (Google)
  - Dynamo (Amazon)
    - Gossip protocol (discovery and error detection)
    - Distributed key-value data store
    - Eventual consistency
Thank You!

Questions?!