The CAP Theorem Discussion

Presenters: Cuong Pham & Biplap Deka
CS525 Spring 2013
CAP Theorem

Atomic/Linearizable Consistency
Exist a total order of all Operations such that each operation looks as if it were completed at a single instant

Availabilty
Every request received by a non-failing node must result in a response

Partition-tolerance
No set of failures less than total network failure Is allowed to cause the system to response incorrectly

Brewer: Pick Two!
Historical Context

2000 — E. Brewer stated CAP trade-offs in his keynote talk at PODC

2002 — S. Gilbert & N. Lynch published a formal proof of CAP

2009 — NoSQL movement began

2012 — S. Gilbert and N. Lynch: “Perspective on the CAP theorem”
         E. Brewer: “CAP Twelve Years Later: How the “Rules” Have Changed”

Today — 150 NoSQL Databases (nosql-database.org)
CAP, Cloud Computing, and NoSQL in Google Trends
Proof Sketch
Consistency is Important

- Affect the correctness of data
- People are used to its strong notion in pre-cloud era
- The ONLY knob you can tune in CAP in many scenarios

**RedBlue Consistency**

**Causal+ Consistency**
CAP Theorem Debate

- With the wide geographical spread of the *cloud*, opportunities for partitioning in the data are not in-significant
  - An justification for NoSQL Movement
- Latency is not in the equation (e.g. PNUT)
- A and C are asymmetric
- Differences between CA and CP?
Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary

Authors: Chen Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, Rodrigo Rodrigues

Presenter: Cuong Pham
Motivation

- **Geo-replication**
  - Internet users are globally distributed
  - Applications replicate data across datacenters
    - Reduce network latencies to users
  - Dilemma:
    - Cross-site consistency latency
    - The problems are magnified with WAN latency
    - E.g.: Synchronous replication via Paxos.

- **Observation:**
  - Strong consistency is not always required: Depend on the applications

- **Goal:**
  - **RedBlue Consistency**: Mixing strong consistency (for application semantics) & eventual consistency (for fast responses) in a same system
Divide Operations into **Red** and **Blue**

**RedBlue Order:**
- Red operations must be totally ordered
- The order of Blue operations can vary from site to site
**RedBlue Consistency**

- **Causal serialization**
  - A site has a causal serialization of the RedBlue order if the ordering is a linear extension of the RedBlue order

- **State Convergence**
  - Convergent if all causal serializations of the RedBlue order reach the same state
  - All *blue* operations must be globally commutative

**RedBlue Consistency**: Each site applies operations according to the causal serialization of the RedBlue order
RedBlue Consistent Banking System

Alice in EU  
△ deposit(20)

Bob in US  
△ accrueinterest()

(a) RedBlue order $O$ of operations issued by Alice and Bob

**Problem:**
- Different execution orders lead to divergent state

**Cause:**
- $Accrueinterest()$ doesn’t commute with $deposit()$

```java
public class BankAccount {
    private float balance;

    public void deposit(float money) {
        balance = balance + money;
    }

    public void withdraw(float money) {
        if (balance - money >= 0)
            balance = balance - money;
        else
            System.out.println("failure");
    }

    public void accrueinterest() {
        float delta = balance * interest;
        balance = balance + delta;
    }
}
```
Operation Decomposition
Generator & Shadow operations

- Observation: Not all operations are commutative
- Split these operations into generator and shadow operations
- Generator Operations
  - Only executed at the primary site against a system state
  - Produces no side effects
  - Determines state transitions that would occur
  - Produces shadow operations

- Shadow Operations
  - Applies the state transitions to all the sites including the primary site
  - Must produce the same effects as the original operation given the original state for the Generator operation

- Separating operations allows for easier formation of abelian groups
  - Allows for more commutative operations (blue operations)
Revisit the Banking System

Original/Generator operation

```plaintext
deposit(float m){
    balance = balance + m;
}

accruinterest(){
    float delta=balance * interest;
    balance=balance + delta;
}

withdraw(float m){
    if(balance-m>=0)
        balance=balance - m;
    else
        print "Error"
}
```

Shadow operation

```plaintext
deposit'(float m){
    balance = balance + m;
}

accruinterest'(float delta){
    balance=balance + delta;
}

withdrawAck'(float m){
    { balance=balance - m;
}

withdrawFail'(){
}
```
Converged... but Invalid

(a) RedBlue order $O$ of banking shadow operations

(b) Invalid but convergent causal serializations of $O$
Red or Blue?

Credit: Authors’ slides
Correct RedBlue Consistent Banking

(a) RedBlue order $O$ of banking shadow operations

(b) Convergent and invariant preserving causal serializations of $O$
Summary

- RedBlue consistency combines strong and eventual consistency into a single system.
- The decomposition of generator/shadow operations expands the space of possible Blue operations.
- A simple rule for labeling is provably state convergent and invariant preserving.
Evaluations
Experimental Setup

- Deployment in Amazon EC2
  - spanning 5 sites (US-East, US-West, Ireland, Brazil, Singapore)
  - Locating users in all five sites and directing their requests to closest server

<table>
<thead>
<tr>
<th></th>
<th>UE</th>
<th>UW</th>
<th>IE</th>
<th>BR</th>
<th>SG</th>
</tr>
</thead>
<tbody>
<tr>
<td>UE</td>
<td>0.4 ms 994 Mbps</td>
<td>85 ms 164 Mbps</td>
<td>92 ms 242 Mbps</td>
<td>150 ms 53 Mbps</td>
<td>252 ms 86 Mbps</td>
</tr>
<tr>
<td>UW</td>
<td>0.3 ms 975 Mbps</td>
<td>155 ms 84 Mbps</td>
<td>207 ms 35 Mbps</td>
<td>181 ms 126 Mbps</td>
<td></td>
</tr>
<tr>
<td>IE</td>
<td>0.4 ms 996 Mbps</td>
<td></td>
<td>235 ms 54 Mbps</td>
<td>350 ms 52 Mbps</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td></td>
<td></td>
<td>0.3 ms 993 Mbps</td>
<td>380 ms 65 Mbps</td>
<td></td>
</tr>
<tr>
<td>SG</td>
<td></td>
<td></td>
<td></td>
<td>0.3 ms 993 Mbps</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Average round trip latency and bandwidth between Amazon sites.
Micro-benchmark Results

Avoid the cost of cross-site communication as much as possible

Figure 8: (a) and (b) show the average latency and standard deviation for blue and red requests issued by users in different locales as the number of sites is increased, respectively. (c) and (d) show the CDF of latencies for blue and red requests issued by users in Singapore as the number of sites is increased, respectively.
Case Studies

- Applications
  - Two e-commerce benchmarks: TPC-W, RUBiS
  - One social networking app: Quoddy
- How common Blue operations are?

<table>
<thead>
<tr>
<th>Application</th>
<th>user requests</th>
<th>Original transactions</th>
<th>LOC</th>
<th>RedBlue consistent extension</th>
<th>LOC changed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>total</td>
<td>read-only</td>
<td>update</td>
<td>shadow operations</td>
</tr>
<tr>
<td>TPC-W</td>
<td>14</td>
<td>20</td>
<td>13</td>
<td>7</td>
<td>9k</td>
</tr>
<tr>
<td>RUBiS</td>
<td>26</td>
<td>16</td>
<td>11</td>
<td>5</td>
<td>9.4k</td>
</tr>
<tr>
<td>Quoddy</td>
<td>13</td>
<td>15</td>
<td>11</td>
<td>4</td>
<td>15.5k</td>
</tr>
</tbody>
</table>

Table 2: Original applications and the changes needed to make them RedBlue consistent.
How common Blue operations are?

- Runtime Blue/Red ratio in different applications with different workloads:

<table>
<thead>
<tr>
<th>Apps</th>
<th>workload</th>
<th>Originally</th>
<th>With shadow ops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Blue (%)</td>
<td>Red (%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>96.0</td>
<td>4.0</td>
</tr>
<tr>
<td>TPC-W</td>
<td>Browsing mix</td>
<td>85.0</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>Shopping mix</td>
<td>63.0</td>
<td>37.0</td>
</tr>
<tr>
<td></td>
<td>Ordering mix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUBiS</td>
<td>Bidding mix</td>
<td>85.0</td>
<td>15.0</td>
</tr>
<tr>
<td>Quoddy</td>
<td>a mix with 15% update</td>
<td>85.0</td>
<td>15.0</td>
</tr>
</tbody>
</table>
Scalability Evaluation

Figure 10: Average latency for selected TPC-W and RUBiS user interactions. Shadow operations for doCart and StoreBid are always blue; for doBuyConfirm and StoreBuyNow they are red 98% and 99% of the time respectively.
Discussion

- Total order in one site: Red operations block Blue operations?
- Operation decomposition:
  - A manual process (how to automate?)
  - Error-prone
- Compare with Cassandra’s three consistent levels (One, Quorum, and All)
- How improvements in network technology (e.g. cross-site latency is a few ms) impact future designs of geo-rep. system?
- The idea of RedBlue operation is similar to:
  - Generalized Paxos
  - Generic Broadcast
- Gemini implementation
  - No Fault-tolerance
  - Bottleneck: serialize Red operations via token passing
Don’t Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS

Wyatt Lloyd*, Michael J. Freedman*, Michael Kaminsky†, David G. Andersen‡

*Princeton, †Intel Labs, ‡CMU

SOSP 2011

CS525 Presenter: Biplab Deka

Disclaimer: Several slides are borrowed from the presentation by the first author at SOSP 2011
Wide-Area Storage
Serves Requests Quickly
Inside the Datacenter

Web Tier

Storage Tier

A-F

G-L

M-R

S-Z

Replication

Remote DC
Desired Properties: ALPS

• **Availability**
  – All operations to the datastore complete

• **Low Latency**
  – Client operations complete quickly

• **Partition Tolerance**
  – The datastore continues to work under network partitions

• **Scalability**
  – The datastore scales out linearly
Consistency with ALPS

Linearizability > Sequential > Causal+ > Causal > FIFO > Per-Key Sequential > Eventual

Strong: Impossible [Brewer00, GilbertLynch02]

Sequential: Impossible [LiptonSandberg88, AttiyaWelch94]

Causal+ : Causal + Convergent  Conflict Handling

COPS

Eventual : Dynamo, Cassandra
Rules for Potential Causality

1. **Execution Thread.** If $a$ and $b$ are two operations in a *single thread of execution*, then $a \leadsto b$ if operation $a$ happens before operation $b$.

2. **Gets From.** If $a$ is a put operation and $b$ is a get operation that returns the value written by $a$, then $a \leadsto b$.

3. **Transitivity.** For operations $a$, $b$, and $c$, if $a \leadsto b$ and $b \leadsto c$, then $a \leadsto c$.

<table>
<thead>
<tr>
<th>Client 1</th>
<th>$\text{put}(x,1) \rightarrow \text{put}(y,2) \rightarrow \text{put}(x,3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client 2</td>
<td>$\text{get}(y)=2 \rightarrow \text{put}(x,4)$</td>
</tr>
<tr>
<td>Client 3</td>
<td>$\text{get}(x)=4 \rightarrow \text{put}(z,5)$</td>
</tr>
</tbody>
</table>

**Figure 2:** Graph showing the causal relationship between operations at a replica. An edge from $a$ to $b$ indicates that $a \leadsto b$, or $b$ depends on $a$.

- The value returned by a get operation has to be consistent with the order defined by these rules.
- It must appear that the operation that writes a value occurs after all operations that causally precede it.
Conflicts in Causal
Conflicts in Causal

Causal + Conflict Handling = Causal+
Previous Causal+ Systems

• Bayou ‘94, TACT ‘00, PRACTI ‘06
  – Limited Scalability
  – All data should fit on same machine (Bayou)
  – Data that could be accessed together should be on same machine (PRACTI)
  – Log-exchange based
COPS

- Dependency metadata explicitly captures causality
- Versions
  - Different values of a key
  - Each replica returns non decreasing versions of a key
- Dependencies
  - \( y_j \) depends on \( x_i \) if and only if \( \text{put}(x_i) \Rightarrow \text{put}(y_j) \)
  - Provide causal+ consistency by writing a version only after writing all of its dependencies
Get

Local Datacenter

Client Library

Key-Value Store

get

get

get
Put

\[
\text{put after} = \text{ordering metadata}
\]

Put

Key-Value Store

Local Datacenter

Client Library

Replication Q

put after
Dependencies

• Dependencies are explicit metadata on values
• Library tracks and attaches them to put_afters
Dependencies

- Dependencies are explicit metadata on values
- Library tracks and attaches them to `put_after`

![Diagram]

Client 1

```put(Key, Val)`

```
deps

```

`version`

```put_after(Key, Val, deps)`

```

39
Dependencies

- Dependencies are explicit metadata on values
- Library tracks and attaches them to put_afters

\[
get(K) \rightarrow \text{value, version, } \text{deps'}
\]

\[
\text{Client 2} \rightarrow \text{deps} \rightarrow K, \text{version, } L_{337}, M_{195}
\]

\[
\text{deps'} \rightarrow L_{337}, M_{195}
\]
Causal+ Replication

Key-Value Store

put_after(K,V,deps)

Replication Q

put after
Causal+ Replication

put_after(K,V,deps)

K:V,deps

deps
L_{337}
M_{195}

dep_check(L_{337})
dep_check(M_{195})

Exposing values after dep_checks return ensures causal+
Basic COPS Summary

• Serve operations locally, replicate in background
  – “Always On”

• Partition keyspace onto many nodes
  – Scalability

• Control replication with dependencies
  – Causal+ Consistency
This Isn’t Enough

- Get Transactions: Provide consistent view of multiple keys
  - `get_trans(key1, key2, key3)`
System So Far

• ALPS and Causal+, but ...

• Proliferation of dependencies reduces efficiency
  – Results in lots of metadata
  – Requires lots of verification
Many Dependencies

• Dependencies grow with client lifetime
Nearest Dependencies

- Nearest are few
- Only check nearest when replicating
Other Mechanisms

• Garbage Collection Subsystem
  – Reduce the amount of extra state in the system

• Fault Tolerance
  – Clients, nodes and datacenter failures

• Conflict Detection Mechanisms
Latency and Throughput

<table>
<thead>
<tr>
<th>System</th>
<th>Operation</th>
<th>Latency (ms)</th>
<th>Throughput (Kops/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>50%</td>
<td>99%</td>
</tr>
<tr>
<td>Thrift</td>
<td>ping</td>
<td>0.26</td>
<td>3.62</td>
</tr>
<tr>
<td>COPS</td>
<td>get_by_version</td>
<td>0.37</td>
<td>3.08</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>get_by_version</td>
<td>0.38</td>
<td>3.14</td>
</tr>
<tr>
<td>COPS</td>
<td>put_after (1)</td>
<td>0.57</td>
<td>6.91</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>put_after (1)</td>
<td>0.91</td>
<td>5.37</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>put_after (130)</td>
<td>1.03</td>
<td>7.45</td>
</tr>
</tbody>
</table>

Table 2: Latency (in ms) and throughput (in Kops/s) of various operations for 1B objects in saturated systems. put_after(x) includes metadata for x dependencies.
Sensitivity Analysis

(a)

Max Throughput (Kops/sec)

Average Inter-Op Delay (ms)

Dep Count

Average Inter-Op Delay (ms)
Sensitivity Analysis

(a)

Max Throughput (Kops/sec)

Put:Get Ratio

COPS
COPS-GT (0)
COPS-GT (1)
COPS-GT (512)

Dep Count

(a)
Figure 12: Throughput for LOG with 1 server/datacenter, and COPS and COPS-GT with 1, 2, 4, 8, and 16 servers/datacenter, for a variety of scenarios. Throughput is normalized against LOG for each scenario; raw throughput (in Kops/s) is given above each bar.
Comments

• Suggests keeping metadata to track dependencies. Isn’t this what Lamport timestamps wanted to avoid?

• Scalability benefits are not clear
  – No comparison with other systems
  – No WAN delays
  – No comparison with eventually consistent datastores

• Will it require each new application that adopts COPS to create a new client library?
Backup Slides
Figure 4: The COPS architecture. A client library exposes a put/get interface to its clients and ensures operations are properly labeled with causal dependencies. A key-value store replicates data between clusters, ensures writes are committed in their local cluster only after their dependencies have been satisfied, and in COPS-GT, stores multiple versions of each key along with dependency metadata.
COPS

Key-Value Store

Causal+ Replication

All Data

Local Datacenter

Client Library
Latency and Throughput

<table>
<thead>
<tr>
<th>System</th>
<th>Operation</th>
<th>50%</th>
<th>99%</th>
<th>99.9%</th>
<th>Throughput (Kops/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thrift</td>
<td>ping</td>
<td>0.26</td>
<td>3.62</td>
<td>12.25</td>
<td>60</td>
</tr>
<tr>
<td>COPS</td>
<td>get_by_version</td>
<td>0.37</td>
<td>3.08</td>
<td>11.29</td>
<td>52</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>get_by_version</td>
<td>0.38</td>
<td>3.14</td>
<td>9.52</td>
<td>52</td>
</tr>
<tr>
<td>COPS</td>
<td>put_after (1)</td>
<td>0.57</td>
<td>6.91</td>
<td>11.37</td>
<td>30</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>put_after (1)</td>
<td>0.91</td>
<td>5.37</td>
<td>7.37</td>
<td>24</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>put_after (130)</td>
<td>1.03</td>
<td>7.45</td>
<td>11.54</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 2: Latency (in ms) and throughput (in Kops/s) of various operations for 1B objects in saturated systems. put_after(x) includes metadata for x dependencies.
Experimental Setup

- Built on top of FAWN-KV: linearizable KV store within a local cluster
- Does not do:
  - Chain Replication
  - Conflict Detection
- Experiments in a single cluster with multiple logical datacenters

<table>
<thead>
<tr>
<th>System</th>
<th>Causal+</th>
<th>Scalable</th>
<th>Get Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>COPS</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>COPS-GT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Summary of three systems under comparison.