Cloud Computing - Notes on the CAP Theorem

Prof. Douglas Thain, March 2016

Caution: These are high level notes that I use to organize my lectures. You main find them useful for reviewing main concepts, but they aren't a substitute for participating in class.

References

1. Eric Brewer, Towards robust distributed systems. In Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (July 16-19, Portland, Oregon): 7
2. Werner Vogels, Eventually Consistent, Communications of the ACM, Volume 52, Number 1, January 2009.
3. Eric Brewer, CAP Twelve Years Later: How the "Rules" Have Changed, IEEE Explore, Volume 45, Issue 2, 2012.
4. Chapter 7 of Tanenbaum and Steen, "Distributed Systems: Principles and Paradigms", Pearson, 2007.

The CAP Theorem

The CAP theorem[1] is an observation about the tradeoffs inherent in designing a distributed system for storing data. Simply put, the CAP theorem states that a given system design involves a tradeoff between the desirable properties of Consistency, Availability, and Partitionability. A given system cannot maximize all three of these properties simultaneously. The concept is generally attributed to Eric Brewer, who posed it as a conjecture at the PODC conference in 2000, building on ideas of consistency and performance that had been explored by systems and databases for some time.

The three properties may be loosely defined as follows:

Consistency – All clients of the system see the same data.
Availability – Clients are able to access and update data rapidly.
Partitionability – The system is able to operate even when the network fails.

You may sometimes hear this explained as "pick two of three" but that is an over-simplification. Each of these properties exists upon a continuum, and attempting to increase one of them requires decreasing another to a certain degree.

This is all very abstract, and best explained with some examples. To get the examples right, we will have to discuss them in some detail, and then come back to discuss the CAP theorem more generally.

[1] *The CAP "theorem" isn't really a proper theorem, since it uses some rather fuzzy*

Example System 1 – Direct Access

Let's start off with a model of a distributed file system. Suppose that we have a server S that contains some files named X, Y, and Z. The server is accessible via a network to clients A, B, and C. The clients occasionally wish to modify those files, so they can send messages like "read X" or "write Y" to change those files. As the server receives these requests, it sends as response message back to the client, indicating that the change is complete. The client waits for the response to come back before attempting another request.

(Sketch the system and some example interactions here.)

Now, keep in mind that these messages flow over a network. This has two effects on the users of the system:

- First, the network imposes some minimum latency (let's say 1ms) on each message. As a result, reading and writing a large amount of data from this file server is going to be very slow – much slower than accessing a local disk.

- Second, the network makes message delivery unreliable. Either a request or a response could be delayed arbitrarily, or completely dropped. In the case of a backhoe cutting through a network cable, there may be no communication at all along a particularly link. If a client doesn't receive a response to a message, then it has no choice but to wait and try the request again. It could wait a very long time!

Let's evaluate this system according to the CAP criteria:

- Consistency – HIGH – Example 1 is highly consistent because every operation is applied in a known order, and all clients have an unfiltered view of the central server.
- Availability – LOW – Every single read or write requires a network operation, making this system much slower than accessing a local file system.
- Partitionability – MEDIUM – If a single client is partitioned from the file server, it cannot perform any operations. However, all other connected clients are able to continue.

(The careful reader will note that we have not yet defined precisely what constitutes higher or lower for each of these three properties. Bear with me and let that be vague for a while, and we will formalize it more below.)

Example System 2 – Write-Through Cache

Take the system from example 1, and modify it by adding a cache of finite capacity to each client node. Let's give each client some simple logic for managing these caches:

- Read – When a client attempts to read a file, it first looks in its cache to see if that file's data is already present. If so, the read is satisfied from that data. If not, then the client issues a read request to the server, waits for the response, and replaces the least recently used (LRU) item in the cache.

- Write – When a client attempts to write a file, it first issues a write command to the server, and waits for a response. If an older version of the file exists in the cache, it is updated to the new value. If not, then the client replaces the LRU item in the cache with the newly written value.

(Sketch the system and some example operations here.)

- Consistency – MEDIUM – System 2 (Write Through Cache) is less consistent than System 1 (Direct Access) because a client may fail to see writes made by other clients, when a value is available within its own cache.

- Availability – MEDIUM – System 2 will see much better read performance than System 1, because reads can be satisfied directly from cache without consulting the central server. However, writes are no faster because they always result in a network operation.

- Partitionability – MEDIUM – A client might be able to continue operating even when the network is down, if it is only performing reads on cached data. However, any write operation must block until the network results.

Thought Experiment: Consider a modification of System 2 in which clients write data only to their local caches, and send data back to the server only when evicted from the cache. How would this affect the CAP properties?

Example System 3 – Consistent Caches

The previous system had a very simple method of dealing with cached data that (obviously) resulted in some serious inconsistencies. Let's try to solve the problem in a different way, by making the system avoid inconsistencies in cached data.

Take the design from System 2, where each client has a cache of finite capacity. Now, when each client tries to read or write a file, do this:

- Read – (Same as System 2)
- Write – When a client attempts to modify a file, it first sends a message to every other client, instructing it to invalidate its cached copy of that file. Once those caches acknowledge that they have purged their copies of the file, the client sends the write request to the server, and updates its own cache.

(Sketch the system and some example operations here.)

Ok, now you evaluate this system according to the CAP criteria:

- Consistency?
- Availability?
- Partitionability?

Thought Experiment: Now you design a system that has a combination of CAP that we haven't seen so far.

How do we quantify CAP?

Now that we have considered several model systems, it should be clear that there is a real tradeoff between Consistency, Availability, and Partitionability. Dealing with the P is central to distributed computing: when we cannot communicate, should we optimistically try to make progress, or pessimistically wait, in order to achieve consistency? There is no single answer to this problem: different applications will require different solutions.

In the analysis above, we were a bit sloppy about stating exactly what is meant by "more" or "less" of each of the three properties.

Availability is probably the easiest to quantify. For a given system, one could measure every attempt to read or write a value, and then compute a statistic like the mean, median, or 99th percentile of latency for various operations. A system that provides a lower mean is providing "more" availability.

Partition tolerance might be measured by evaluating what set of clients and operations can continue in the presence of network outages. A system that allows all clients to perform reads during a network outage is providing "more" partition tolerance than a system that allows no clients to perform reads.

Consistency is the most complex property to describe. There exist a variety of consistency models that can be implemented by adjusting just how and when caches are updated, and whether clients can continue to operate during a partition.

Strong Consistency:
    Once an update is complete, all clients will see that new value.

Causal Consistency:
    If process A tells B that it has updated X, then B will see the latest value of X.

Read-Your-Writes:
    If process A updates X, then A will never see an older value of X.

Monotonic Reads:
    If process A reads a value from X, then it will never read back an older value.

Monotonic Writes:
    All writes by process A are applied in the order they are given.

Eventual Consistency:
    All updates will become visible to everyone, if you wait long enough.

Chaos:
    No guarantees!

Storage Replication

Many distributed systems replicate data across a large number of storage devices. This is done to provide insurance against storage failures, but also to provide a high degree of availability for commonly used data.

But multiple copies results in a new kind of consistency problem. If I update 2 out of 5 copies, is the write “complete” or not? Systems answer this question in various ways, which are known as quorum protocols.

Vogels [2] gives a framework for thinking about this:

Suppose you have a system with N replicated storage units. To update an item, a client must write W of the replicas upfront. (The remainder will get updated eventually in the background.) To read an item, a client must read R of the replicas, in order to decide whether the most recent value has been read. (If the values differ, assume you can tell which one is the newest.)

For example:

> N=2, W=2, R=1 is a strongly consistent system: a writer must update both replicas, and a reader can read either one of them. (RAID 1)
>
> N=2, W=1, R=2 is also a strongly consistent system: a writer can update either replica, and a reader must read both to obtain the latest.
>
> N=2, W=1, R=1 is an eventually consistent system: the writer can update either replica, and the reader can read either replica, so you may not see consistent results.

And now some general observations:

> (W + R) > N is strongly consistent ,while (W+R) <= N is weakly consistent.
>
> W < (N+1)/2 means write conflicts can occur.
>
> Monotonic read (and write) consistency is achieved by making clients “sticky” with respect to the replicas that they use.

Strong consistency models can only be achieved by delaying either reads or writes during partitions. In a large enough distributed system, partitions are omnipresent. Ergo, very large distributed systems almost always rely on weak or eventual consistency in order to achieve acceptable availability.