The CAP-Theorem & Yahoo's PNUTS

Stephan Müller

June 5, 2012

Abstract

This text is thought as an introduction to the CAP-theorem, as well as for PNUTS, a particular distributed databased. It subsumes a short presentation on the same topics. A more illustrated presentation may be found in the slides, [Mü].

The CAP-theorem relates consistency, availability and partition tolerance of distributed systems. After a short introduction, these concepts are explained and examples for possible combinations are given. The CAP-theorem is explicitly stated, a proof and the necessary tools are presented.

In the second part a distributed database from Yahoo is presented: PNUTS. It is build for geo-replicating data to provide worldwide low latency access. The text explains its local and global architecture. Also the consistency model and its realization is explained.

Contents

1 The CAP-Theorem

CAP is an acronym for *Consistency, Availability, Partition tolerance*. Any web service is desired to be *available*, that is its *raison d'être*. For better scalability and lower latency distributed databases are used. These advantages have to be paid by answering the question of *partition tolerance*. How is the *consistency* of data guaranteed if the intercommunication of different parts of the underlying database is interrupted? The CAP-Theorem establishes in some sense an upper bound for distributed systems. In a very sketchy formulation it states:

It is impossible to have a distributed web service which is at the same time consistent, available *and* partition tolerant.

1.1 History

In an intuitive manner, the CAP-Theorem is obvious. So it is surprising that its first official presentation is only dated to July, 2000. On a ICM symposium *Principles of Distributed Computing*, Eric. A. Brewer formulated the CAP-Theorem in a keynote presentation, [Bre]. His formulation was:

> For a shared data-system, you can have at most two of the following properties: Consistency, Availability, Tolerance to network Partitions.

Since Brewer left it as an informal statement, also the name *Brewers conjecture* established. The problem is that, as soon as one tries to formalize its statement, there is nothing trivial about CAP-theorem.

In 2002, Nancy Lynch and Seth Gilbert [GL02] published a precise formulation and gave a proof. It is interesting to note that the proof itself is not difficult. The hard part consists in developing a framework for a useful formulation.

1.2 A General Overview

1.2.1 Availability

The requirement of availability is that a web service should work fully or not at all. Any request should succeed or at least give a meaningful response. For example, a shopping cart application answering with a browser notification for a time out clearly violates availability. An important aspect is, as pointed out in J.Browne's blog [Bro], that a non-used web service is of no benefit and used ones are often unavailable just because of heavy usage. This is reflected in the common experience that a website is unavailable just when it is most needed.

1.2.2 Consistency

Consistency is a broad topic. Many different understandings depending on the point of view are possible. From a database perspective, we could say that in a consistent system all operations respect integrity of the data and that all users see the same data at all times. This is meant in the sense that the same requests at the same time originated from different users result in the same results. That is the ACID perspective.

1.2.3 Partition Tolerance

A distributed system consists of a collection of communicating nodes. Partition tolerance requires that as long as some node is reachable the system still should behave well. In other words, the system should be as available as the underlying network. On an intuitive level this is clear, whereas in the formal framework below this is very subtle.

1.2.4 AC - Available and Consistent Systems

There are many examples for such systems. From a database perspective, this is the world of ACID-design with no replication. Such databases run on a single machine (that is an unique and isolated node in a network) and implement ACID-constraints which subsume availability and consistency in the sense of above. Of course, there are many situations where ACID-databases are suitable. But in need for scalability and fault-tolerance distributed solutions emerge.

Another system of this type are (non-distributed) file-systems. In this case it is clear that availability and consistency are primary design goals. Also it should be noted that latency is an important factor.

1.2.5 PC - Consistent and Partition Tolerant Systems

A good example form everyday life is the cash machine service of any bank. It is expected to be distributed and consistent. Technically, many distributed databases belong to this category. They implement for example some type of majority protocol.

1.2.6 PA - Available and Partition Tolerant Systems

Systems of this category often share the property that availability is absolutely necessary. A primary example are nameservers in DNS. It is acceptable to deliver outdated data, but unavailability would heavily decrease usability of the internet. This generalizes to many web services (for example PNUTS, see part two below), where stale data is not a serious threat or could be handled on the application layer. As already pointed out, a non-connected website is of no use at all.

The idea boils down to: if one has to drop availability, consistency or partition tolerance, often consistency is chosen. The other two choices do have serious drawbacks concerning availability or again latency.

1.2.7 Misunderstanding the CAP-Theorem

Because of the CAP-theorem, architects of distributed systems often decided to sacrifice (strong) consistency to guarantee high availability. However in D.J. Abadi's article [Aba12] this is asserted to be unnecessary. For example, it is possible to implement a

distributed database satisfying complete ACID as long as no network partitions arise. Abadi criticizes the default setup for unnecessary lowered consistency in many systems.

1.3 Beyond the CAP-Theorem: PACELC

1.3.1 Consistency Models

As outlined in the previous paragraph, the requirement of (strong) consistency is replaced by weaker versions. One famous model is *eventual consistency*. It states that an update propagates in certain amount of time (if no other updates occur), so that in the end the database is consistent. Slightly different requirements are possible. By adding constraints, eventual consistency can be split in various different types. For example timeline-consistency as implemented by PNUTS. In general this leads to the BASE-paradigm opposed to ACID, where BASE stands for *Basically Available, Soft-state and Eventual consistent*. In this text, this will not be developed more explicitly. However, it is important to note that it is not a binary choice between ACID and BASE. It has to be understood as a spectrum with BASE and ACID as opposed ends, [Bre].

1.3.2 Considering Latency

As already mentioned several times, latency is always important. Slow web services are not accepted. According to Browne's blog, increased response times of Amazon by 10 ms will cost 1% of sales. This motivates the following refinement of the CAP-theorem:

> If the system is **P**artitioned, how is **A**vailability traded against **C**onsistency and **E**lsewise how are **L**atency and **C**onsistency balanced?

This is proposed in [Aba12] and is called PACELC (pass-elk).

1.4 Precise Formulation and Proof of the CAP-Theorem

In [GL02], the CAP-theorem is formulated as follows:

> It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties: availability and atomic consistency.

In order to be able to give a proof or even sketch it, it is necessary to introduce a technical framework which clarifies the formulation. This is done in [Lyn96]. We briefly sketch the most important concepts. The focus is to present the ideas rather than to give the correct definitions.

1.4.1 I/O-Automata

Definition: [Lyn96, Ch.8.1, p.201] An *I/O-Automaton* may be thought of as a deterministic state machine, but with infinitely many states. Such an automaton A comes with a *signature* which is a disjoint union of *input-*, *internal-* and *output-actions*. The input actions correspond to A's input and the output actions to A's output. We assume all automata to be *input enabled*, that is in every state and for every input action there must be a transition labeled with this action, so that the automata knows how to react

on input occurring at arbitrary times. But we do not impose any conditions like a global clock. So such an automata is timeless, or *asynchronous*.

There is more to say about this definition, for example some initial state has to be specified and there is something like threads, but the shortened version should be sufficient. However, an important concept is *execution*:

This is a (possibly infinite) sequence $(s_0, \pi_0, s_1, \pi_1, s_2, \ldots)$ where the s_i are states and the π_i are actions (of any type), such that $s_i \to s_{i+1}$ via π_i is an allowed transition in A. The subsequence containing only external actions (i.e. no states and no internal actions) is called the *trace* of the execution. Any sequence of external actions is called *admissible* if it is the trace of an execution.

I/O-atomata are the basic building blocks for the whole setup. It is therefore important to note that it is possible to plug together several of such automata obtaining a new one. In concatenating automata the output of one are connected to the input of another. The unconnected actions are the new inputs resp. outputs of the composed automaton.

1.4.2 Read/Write-Dataobjects

[Lyn96, Ex. 9.4.1, p.245 & Ch.13.1, p.399] A *read/write-dataobject R* is a special I/O-automaton modeling shared memory between several processes, say P_1 and P_2. For example, a simple register. Each process could be allowed to read, write or both. For each process R offers a read and/or write interface. A *read-interface for process* P_i has $read_i$ input and v_i as output. Similarly, a *write-interface* provides $write_i$ as input and ack_i as output. The idea is that a process P_i reads the value of R by sending a $read_i$-request and R returns its current value via the v_i output. Writing a new value Z_0 works similarly: P_i sends $write_i(z_0)$ to R's input. After a certain time R answers with ack_i. This acknowledgment approves the successful write, from then on R answers to every *read* with z_0. If no *ack* is send, a read request can return the new value z_0 or still an outdated old value.

1.4.3 Atomic Consistency

Atomicity is a subtle concept. We restrict ourselves to *atomic read/write-objects*. The reference is [Lyn96, Ch.13.1, p.398]. The definition is that any execution could be *linearized* in the following sense: Given an execution α and consider its trace $tr(\alpha)$. If one can insert (ordered) *serialization points* between every pair of $write/ack$ and $read/v$ actions, such that the sequence obtained by writing $write, ack$ resp. $read, v$ for every serialization point is an admissible trace, then the execution is *atomic*.

The idea is that one can always explain the (observable) behavior of an atomic read/write object by a set of consecutive interactions. Even in the case the observed behavior emerges from highly concurrent interaction. Note that it is not required to reproduce the trace of any execution by a linear execution.

An important example is that

$$write_1(z_1), ack_1, read_2, v_2(z_0)$$

is not atomic. Assuming that $z_0 \neq z_1$, then it is impossible to read a value z_0 just after z_1 was written. In the proof below we will construct an execution containing the above. This will violate the assumed atomicity.

1.4.4 Proof of the Theorem

The proof is taken from [GL02]. In their paper this is theorem 1. Assume we have an asynchronous network A which is a disjoint union of two non-empty components G_1 and G_2 and there is an atomic read/write-object R with initial value z_0. The proof is indirect, under the hypothesis of availability we construct an execution which contradicts the atomicity of R.

Let α_1 be the beginning of an execution of A in which a single $write(z_1)$ (for some $z_1 \neq z_0$) occurs. α_1 ends with the corresponding *ack*. This acknowledgment from R is sent because of the availability hypothesis. Moreover, we assume that no other interaction with R takes place, neither in G_1 nor in G_2 and no messages between G_1 and G_2 are delivered.

Now we take another prefix of an execution α_2. In α_2, a single *read* request for R occurs from G_1. Again, by availability, this request has an answer with which α_2 ends. And similarly as for α_1 no other interaction with R takes place and all messages between the network components are lost. Its clear that R returns z_0.

Now consider the composed execution $\alpha = \alpha_1\alpha_2$. That is α_1 followed by α_2. Because all messages between the components are lost, to G_2 the execution α is indistinguishable (in the sense of [Lyn96, Ch.8.7, p. 229]) from α_2. Therefore R has to answer the *read* request with the initial value z_0 in both executions α and α_2. But the read starts after the acknowledgment of writing z_1 to R, this contradicts the atomicity of R.

2 PNUTS

PNUTS is an acronym for *Platform for Nimble Universal Table Storage*. It is a hosted data service from Yahoo providing a geographically distributed database with low latency and high availability guarantees. It is primary designed to provide a distributed database infrastructure for web services storing frequently accessed user data, e.g. social networks or customized websites. There are basically two published descriptions of PNUTS, these are [SCL+12] and the more explicit but much older [CRS+08].

2.1 Local Architecture

As an outline, PNUTS consists of several data centers, called *regions*. These, in 2007 there were 18 of them, are scattered around the globe. Each region consists of several *storing units* holding all the data and *routers* providing the clients interface and so called *tablet controllers* organizing the storage units. Since all regions hold the same data, users can interact with closest, i.e. fastest region. Updates or replication propagate asynchronously using the so called *Yahoo Message Broker* (YMB).

2.1.1 Data Model and Storage Units

This is a description within a single region concerning a single customer. As usual the data is organized into tables. Each table is *horizontally* partitioned into subtables called *tablets*. An entry is called *record* and consists of a key, some metadata and a JSON-object. This allows a flexible scheme for Yahoo's customers, transparent for PNUTS. A table may be split over several storage units, but a tablet always lives on a unique (within a region) storage unit. The association of tablets and storage units is managed by the tablet controller. The tables are stored using a classical DBMS such as MySQL

or InnoDB. Both *hashed* and *ordered* tables are possible, depending on the customers preferences. Also for storage units commodity hardware can be used and since the interval mapping is not static, it is possible to add or remove units or split tablets to avoid read or write localities (with respect to storage units).

2.1.2 Tablet Controller

This section is still local and describes a single region. The tablet controller is an active/standby server pair managing an *interval mapping*. This map splits the key space into intervals and assigns to each such interval a tablet and a storing unit holding this tablet. This mapping is critical, so the tablet controller has a backup machine. It is important to separate it from the data path. Therefore routers are introduced.

2.1.3 Routers

Routers are in the data path and handle communication with users. The API is relatively simple. So their primary task is to forward the client's request to the corresponding storage unit. To do that, they hold a copy of the interval mapping and perform a binary search. This is efficient since the interval mapping has a size of few hundred megabytes and fits into the router's RAM. The interval mapping is periodically polled from the tablet controller. If a client request cannot be resolved, for example because of an outdated interval mapping or storage unit failure, then an error is returned and the router reinitializes. In this sense, they are *soft state*.

2.2 Consistency Model

Before we continue to describe PNUTS's architecture from a global point of view, we discuss the consistency model. It is enough to know that somehow updates propagate asynchronously using the Yahoo message broker system. The *unit of consistency* is not the database itself as in classical ACID-like DBMSs. Instead, consistency on record level is provided.

2.2.1 Time-line Consistency

PNUTS implements *(per-record) time-line consistency*. This is a form of eventual consistency or weak consistency in general. It is defined as follows:

> All replicas of a given record apply all updates in the same order.

The slogan is: All replicas move only forward in time. To achieve this, each record has a privileged replica, the so called *master*. In each record's metadata it is stored in which region the master copy is hold. If a storage unit, which does not hold the master copy is assigned to update a record, it forwards the request to the corresponding master replica. This is done using the message broker system, which is explained in the next section. From the master replica the update is asynchronously propagated (back) to all other replicas. This protocol fulfills the desired properties. As only the master replica accepts updates, an ordering of updates is achieved. And as all other replicas follow the master, they apply these updates in the same order. Note that there are times when the data is not consistent.

2.2.2 Versioning

To allow the users to handle possibly stale data, each record has a version number in its metadata. This number consists of a *generation* and a *version* counting the (re-)insertions and updates. The API provides different read and write calls. As an example a read request carries a version parameter (`any`, `latest` or an actual version). `read-any` just returns the data from the region in which the request arose. In contrast, `read-latest` is forwarded to the master, which by definition has the latest version. Similarly, a `read-critical(version)` call always returns data with the same or newer version than specified.

Of course, a `read-any`-request has a much lower latency than one of the others which may involve inter-regional communication. And often, for example for user status information, stale data is not a problem. Since practically requests have a high locality (with respect to regions), it is desirable to reflect this in choosing the master. PNUTS maintains a list (in the record metadata) of the origins of the last three updates and changes mastership if this seems useful.

2.2.3 Unavailability of master replicas

An important questions is what happens if a storage unit containing a master replica fails. In this case availability is affected. All writes are locked and the data has to be replicated from another region. Again the YMB is used. If another region does not contain the latest version, the message broker system knows how to do the necessary updates. In practice, this is very rare since replications have latency essentially equal to network latency. Backup regions with low latency connection are used. In [CRS+08], a version branching system is proposed with the possibility of (automatic) merge of branches at later times. Nothing is said about, if this is implemented. However, recent versions support unmastered writes in which conflicts are resolved using some kind of timestamps similar as in Amazon's Dynamo.

2.3 Global Architecture

2.3.1 Yahoo Message Broker

The message broker system (*YMB*) is used for interregional communication. It is a *topic based publish/subscribe-system* developed by Yahoo. It is part of Yahoo's so called *Sherpa* project [Neg]. We use YMB as a black box and outline only the provided functionality. Each tablet is a *topic* and each region holding a copy subscribes to that topic. An update is considered to be committed if it is submitted to YMB. This is not a problem since the message broker has strong guarantees. For example, no updates are lost and updates are not deleted until it is verified that all replicas applied it and of course the order of updates is strictly respected. This works even in the case of broker machine failures. YMB is also used as a redo-log.

The details of the involved protocols are far beyond the scope of this text. Also there seem to be no explicit descriptions available. A starting point may be reading about *HedWig* which is an open source project providing similar functionality, see [Apa].

References

[Aba12] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[Apa] ApacheWiki. Apache zookeeper - hedwig. `https://cwiki.apache.org/ZOOKEEPER/hedwig.html`. last visited on 06/01/2012.

[Bre] Eric. A. Brewer. Towards robust distributed systems. `http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`. last visited on 05/29/2012.

[Bro] Julian Browne. Brewer's cap theorem - the kool aid amazon and ebay have been drinking. `http://www.julianbrowne.com/article/viewer/brewers-cap-theorem`. last visited on 05/29/2012.

[CRS+08] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[GL02] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2): 51–59, 2002.

[Lyn96] Nancy Ann Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[Mü] Stephan Müller. *CAP & PNUTS*. `http://www.math.hu-berlin.de/~muellste/CAP-PNUTS.pdf`. last visited on 06/02/2012.

[Neg] Toby Negrin. Moving to the cloud. `http://developer.yahoo.com/blogs/ydn/posts/2009/06/sherpa/`. last visited on 06/01/2012.

[SCL+12] A. Silberstein, J. Chen, D. Lomax, B. McMillan, M. Mortazavi, P. Narayan, R. Ramakrishnan, and R. Sears. Pnuts in flight: Web-scale data serving at yahoo. *Internet Computing, IEEE*, (99):1, 2012.