Cedric McDougal
3/12/2012
ENGL 3302, SEC 09
Professor Runge
Word count: 3024

Overcoming the CAP Theorem: A survey of research on distributed computing

Abstract – The vast amount of digital data in the world has led many companies to adopt distributed computing. However, this is not a perfect solution. Eric Brewer's CAP Theorem outlines a few of the limitations inherent in distributed computing. This paper explores current efforts to overcome these limitations. First, I introduce and explain the CAP Theorem. Then, I examine modern strategies for addressing the CAP Theorem limitations. Finally, I conclude the paper by summarizing my findings, explaining limitations in my research, and ruminating on the future of distributed computing.

I. Introduction

The amount of digital data in the world today is staggering. In an average 60 seconds on the Internet there are 20,000 posts on Tumblr, 600 videos added to YouTube, 1,500 blog posts created, 168 million emails sent, 695,000 Facebook status updates, and 12,000 ads posted on Craig's List [1]. This amounts to thousands of terabytes [2] that must be managed in order to keep these services running. To handle this load, many companies have given up on traditional database models in favor of distributed computing.

However, distributed computing is no silver bullet. There are many limitations introduced when scaling a database across hundreds or thousands of servers. One of the major limitations is described by Eric Brewer's CAP Theorem [3]. This theorem states that a distributed system can only guarantee two out of the following three properties: consistency (every node always has the same data), availability (the system is always accessible), and partition-tolerance (the system can still function if an arbitrary set of nodes is lost). Even though this theorem was proposed in 2000, it is still a driving factor for distributed system design today.

This paper explores how modern systems have attempted to overcome the limitations proposed in the CAP Theorem. Five major techniques are discussed: replication, partitioning, multi-version concurrency control, locking, and fault prediction. Examples of systems that use these strategies are included in each section.

II. CAP Theorem

Even though the CAP Theorem states that any of the three properties (consistency, availability, partition-tolerance) can be sacrificed, a practical system cannot afford to give up partition-tolerance. Partition-tolerance is the ability of a system to function even when nodes cannot communicate. In a distributed system, this situation is quite common. Chao Wang et al. [4] estimates that the mean time between node failures in contemporary systems is in the range of 6.5 to 40 hours. Therefore, a system that is not partition-tolerant would be unable to maintain consistency or availability for some period of time *every day*. In a world where people and businesses rely so heavily on the robustness of their technology, this type of occurrence is unacceptable. As such, partition-tolerance is almost never sacrificed, leaving system designers with a choice between consistency and availability.

Consistency means that every process in a system has the same representation of the system's state, and every change is propagated to every other process before the system becomes accessible again [5]. This property is highly desired in certain applications in which it is very important that customers are served with the most accurate data. For example, consider a banking system that offers

online account management. If a user deposits money into an account, it is critical that the change is reflected across the entire system before the user is allowed to withdraw or deposit any more money. Otherwise, it is possible for the user to see an inaccurate account balance which could lead to serious consequences for the user and the bank.

Availability means that users can always read and write data with relatively constant response times, regardless of the state of the system [6]. It is important to maintain a high state of availability because outages can cost a lot of money and affect customer trust [6]. This has driven numerous systems to design for high availability and low consistency [7], including Amazon's S3 services [5]. A good way to understand the importance of availability is to consider a situation in which a widely used service is affected. This happened in 2009 when GMail was inaccessible for almost three hours [8]. This may not seem like a lot of time, but considering that GMail now has 350 million active users [9], even a small amount of down-time can have a widespread and serious impact.

III. Replication

A very common technique for increasing the availability of a system is *data replication*. This is a technique in which a database is copied into multiple locations. These locations could be hyper-local (on the same machine), local (in the same data center), or widespread (in different data centers across the country or the globe). Storing multiple copies of the data protects against single point of failure deficiencies – if one node is compromised, the data can still be accessed from a different node [5]. However, in order to maintain consistency, every change to the data must be distributed to every replica. This can happen in one of two ways: synchronously or asynchronously [7].

A. Synchronous Replication

Synchronous replication is a method that preserves consistency at the expense of availability [7]. It works by preventing access to a piece of data until that data is consistent across every replica. This is used primarily by scalable relational systems such as MySQL Cluster and Clustrix [7]. These systems are distributed versions of traditional models, most of which have strict constraints regarding consistency [7].

Some scalable relational systems, such as VoltDB [7], have attempted to maintain moderate levels of consistency *and* availability by using synchronous replication with a master/slave model. In this model, the only node that accepts reads and writes is the master. If the master fails then one of the slaves takes over while the master attempts to recover. Restricting user access to just the master node makes it much easier to provide consistency using synchronous replication because the data doesn't have to be locked while the update propagates [6]. Furthermore, the use of backup slaves provides a higher level of availability because reads and writes are always available. However, if the master node fails before an update has been distributed, data loss may occur [6] (thus affecting consistency). Also, restricting access to a single node affects performance because requests cannot be distributed across the system [6] (thus affecting availability).

B. Asynchronous Replication

Asynchronous replication is a method that preserves availability at the expense of consistency. It works by always allowing access to data, even if that data is stale because updates haven't had a chance to propagate throughout the whole system [6]. This results in a higher probability of accessing inconsistent data [5]. The majority of distributed systems make use of asynchronous replication [6,7].

Subramaniyan et al. [10] describe one implementation of asynchronous replication called the gossip protocol. It works similar to the way gossip works in real life. Every node exchanges messages (or “rumors”) with some set group of other nodes. When a node is supplied with an update to a piece of data, that update is communicated as a message to the node's correspondents. Each correspondent then communicates that message with its own group. Eventually, that update will have made its way to every node. With this strategy, multiple versions of the same data can exist within the system at the

same time. Thus, there is a lack of consistency. However, given enough time, the system will become consistent. This is called *eventual consistency* [6].

Sakr et al. [6] describe another implementation of asynchronous replication called the publisher/subscriber (pub/sub) model. In this model, every node accepts reads, but there is some limited number of master nodes that accept writes. These masters are the “publishers” – they publish updates to a “subscriber” group of slaves. A major difference between this method and the gossip protocol is that in this method the masters are decoupled from the slaves. In other words, the masters publish without knowledge of who is listening. This allows for much easier scalability, which in turn offers much greater availability.

Sakr et al. [6] go on to explain how Yahoo's PNUTS uses an interesting twist on the pub/sub model in order to increase consistency. In this system, there is not a master for the whole database, but rather a master for each piece of data. This master is chosen automatically based on which node gets the most requests for a particular piece of data. This way, the number of people that are getting the most up-to-date version of the data is maximized. Also, this provides another advantage over the gossip protocol because in it is always known where the most recent data is located. Thus, if a user requires more consistency for a query he can request to get the most recent version with the understanding that it might not be available.

IV. Partitioning

Another common method for increasing availability in a distributed system is called *database partitioning*. Partitioning is the act of splitting a database up into different sections that can be stored in different places [5]. Similar to replication, partitioning increases availability by preventing significant data loss when a single node fails [5]. However, it comes with the added benefit of increased query speed because queries can be parallelized across multiple machines [5]. This helps the system maintain consistent performance even when some resources aren't available. Many distributed systems (Terrastore, HBase, Cassandra) support automatic partitioning of data [7]. There are two types of partitioning: horizontal and vertical.

A. Horizontal Partitioning

Horizontal partitioning is when a database is split up by row [6]. For example, a table with 1000 rows might store groups of 100 rows on 10 different nodes. An example where this might be useful is if there is a table of users and it is partitioned by country. This strategy would make it easy and efficient to search for customers in a particular country because all records for those customers are stored on the same node [7]. Horizontal partitioning is used in HyperTable, PNUTS, and Amazon's Dynamo [6,7].

B. Vertical Partitioning

Vertical partitioning is when a database is split up by column [6]. For example, a table might store similar column types together on different nodes. This is only effective when queries typically request information from the same column groups. An example would be a table of users that separates mostly-constant information (like birth dates) from frequently-updated information (like Facebook status) [7]. In this case, database writes would only have to go to one node to complete, which would be much more effective than having to write to multiple nodes [7]. This is a pretty popular method among some of the big-name systems, like Google's Bigtable, Dynamo, and HBase [6,7]. These systems allow the user to define “column families” that can be distributed among nodes. Each column family is essentially treated as a separate table and may be horizontally partitioned itself [6].

V. Multi-version concurrency control

Yet another popular method for increasing availability is called *multi-version concurrency control* (MVCC) [6,7]. In this method, data is never overwritten. Instead, every update creates a new version of the data. This increases availability because writes are *never* blocked. However, this causes a problem with consistency because now the

system is retaining many versions of the same data across different nodes. Nevertheless, multiple cases have shown MVCC to be significantly useful when dealing with complex systems that evolve over time, which has led cutting-edge systems like ASTERIX to adopt it even for non-versioned systems [11].

The main difference between implementations of MVCC is how the versions are tracked. One method is to create a timestamp associated with a piece of data whenever an update completes [11]. This makes it easy to figure out which version is the most recent. However, it adds space overhead because each piece of data needs to store this extra information with it. Another method is to use vector clocks [12]. These are similar to timestamps except they also store the causal relationships between updates. This allows one to track a series of events, which provides the unique ability for users to not only get the most recent version of the data, but also see how that data has changed over time. This is a good model for sites such as Wikipedia that track every update [12].

VI. Locking

Locking is the counterpart to MVCC: data is modified destructively, and while this is happening that data is inaccessible to reads and writes. Locking is a method that was used extensively in traditional relational databases because it maintains strict consistency [7]. In a system that uses locking, every update either fails completely or succeeds completely – the system is never in a partial state. This guarantees that any time someone reads a piece of data, it will be the most up-to-date. Locking is often used with synchronous replication [7] to ensure that every node gets the updated data before the system becomes available again.

The main difference between locking implementations is the granularity of the lock. For example, systems such as MongoDB [7] provide locking at the field-level. This creates a system with greater availability because updating one field doesn't lock the whole row. However, such control does add some overhead to queries. The next level of locking is row-level. ScaleDB [7] is an example of a system that uses row-level locking. In this model updating any field in a row will lock the entire row. This reduces availability, but increases performance. Lastly, there is a type of locking called optimistic locking (or optimistic concurrency control) [6]. This is actually a system where no locking happens, but if a conflict is detected then either the whole update is restarted or the user is given a choice to specify which update is persisted. This is useful in systems that don't have many conflicts because it prevents the overhead induced by locking but still guarantees consistency. One system that uses optimistic locking is Project Voldemort [7].

VII. Fault Prediction

Wang et al. [4] introduce a new strategy of overcoming CAP limitations that is currently being researched called fault prediction. The driving motivation behind this approach is reducing the need for sacrificing consistency and availability by preempting system errors. In other words, if a system can predict when a node is likely to fail then it can prepare back up nodes to take over immediately. This is different from current systems that engage in *reactive* fault handling – the system doesn't shift to other resources until an error happens. A major advantage of the predictive model of fault handling is that, if it's done well, a system can have near 100% availability while sacrificing very little consistency. For example, consider a situation in which the system detects that a node has a high probability of failing in the next couple hours. Given this information, the system can prep a backup node to be a master with the most recent data and switch before there is ever a failure. Since this avoids the situation where the master fails unexpectedly while an update is still propagating, it greatly increases consistency. This strategy has not been implemented in any publicly available modern systems yet.

VIII. Conclusion

Five major strategies are employed in distributed systems to overcome the limitations proposed in the CAP Theorem: replication, partitioning, multi-version concurrency control, locking, and fault prediction. The property most often sacrificed in modern systems is consistency. However, some

systems choose to offer moderate levels of both consistency and availability. Nevertheless, no system has yet been able to guarantee both high consistency and high availability. Current research efforts are being directed towards strategies for making this guarantee possible, but there is still a long way to go. Unfortunately, a lot of the cutting-edge systems remain proprietary, so there are some strategies in the works that are not yet available to the public (such as Microsoft's SQL Azure and Amazon's SimpleDB [6]). It will be important to follow these changes over time because distributed computing is still in its infancy – most of the popular systems are only four or five years old. Also, there are many technological advances on the horizon, such as quantum computing, that could create a whole new playing field for distributed systems. The creation of data in the world is only going to speed up, which means our strategies for managing that data will have to advance rapidly in order to keep up.

References

[1] Shanghai Web Designers. 60 SECONDS - THINGS THAT HAPPEN ON INTERNET EVERY SIXTY SECONDS [INFOGRAPHIC]. [Online]. *2012(02/26),* Available: http://www.go-gulf.com/blog/60-seconds.

[2] Anonymous "5 Million Terabytes of Data on the Internet & Google has only 0.004% of it!," vol. 2012, 07/30/2011, .

[3] E. Brewer, "Towards Robust Distributed Systems," 2000.

[4] C. Wang, F. Mueller, C. Engelmann and S. L. Scott, "Proactive process-level live migration and back migration in HPC environments," *Journal of Parallel and Distributed Computing,* vol. 72, pp. 254-267, FEB, 2012.

[5] D. Liu, R. Deters and W. J. Zhang, "Architectural design for resilience," *Enterprise Information Systems,* vol. 4, pp. 137-152, 2010.

[6] S. Sakr, A. Liu, D. M. Batista and M. Alomari, "A Survey of Large Scale Data Management Approaches in Cloud Environments," *IEEE Commun. Surv. Tutor.,* vol. 13, pp. 311-336, 2011.

[7] R. Cattell, "Scalable SQL and NoSQL Data Stores," *Sigmod Rec.,* vol. 39, pp. 12-27, DEC, 2010.

[8] M. Siegler. Where were you during the great gmail outage of february 2009?. *2012*2009. Available: http://venturebeat.com/2009/02/24/where-were-you-during-the-great-gmail-outage-of-february-2009/.

[9] M. Brownlow. Email and webmail statistics. *2012*2012. Available: http://www.email-marketing-reports.com/metrics/email-statistics.htm.

[10] R. Subramaniyan, P. Raman, A. D. George and M. Radlinski, "GEMS: Gossip-Enabled Monitoring Service for scalable heterogeneous distributed systems," *Cluster Comput.,* vol. 9, pp. 101-120, JAN, 2006.

[11] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou and V. J. Tsotras, "ASTERIX: towards a scalable, semistructured data platform for evolving-world models," *Distrib. Parallel Databases,* vol. 29, pp. 185-216, JUN, 2011.

[12] I. Zhuklinets and D. Khotimsky, "Logical time in distributed software systems," *Program. Comput. Softw.,* vol. 28, pp. 174-184, MAY-JUN, 2002.