Consistency in Distributed Databases

Therin Irwin
tcirwin@calpoly.edu

ABSTRACT

Consistency has always been a pillar of the database fundamentals. Consistency has always trumped partition tolerance and availability when it comes to database systems, but not any longer. The highly distributed and always-available structure of the Internet makes it difficult to keep consistency when users want data as soon as possible. This paper will discuss the history and development of attaining reasonable consistency in data when dealing with distributed networks of computers. We present two differing styles of consistency, one that guarantees consistency and one that focuses on high availability while still retaining some consistency.

1. INTRODUCTION

This survey will study how consistency is achieved while maintaining the desired levels of availability and partition tolerance for highly-scalable database applications. I will start by explaining the differences between traditional database consistency (the C in ACID) and distributed database consistency (the C in CAP), because they are not exactly the same concept and the difference is significant. Then I will discuss weighted voting, which serves as the underpinnings of all quorum-based distributed database architectures today. This scheme involves a quorum that must be met before any node can read or write data. Next, I will briefly outline Paxos, which builds on the quorum model to provide a stable, scalable system in the presence of network and node failures. Then, I will discuss how applications can be trusted to handle consistency that users typically rely on the database to perform, and what kinds of applications can do this, because there are significant limitations to this model. Finally, I will discuss how consistency is achieved in current database architectures and how partition tolerance and availability are managed while maintaining a consistent state.

Obviously, different applications might require varying degrees of each design consideration and there are a vast number of such applications, so I have selected two different techniques to survey. The first technique I will study guarantees consistency by using a quorum; an example is Google's Chubby lock service which uses a Paxos consensus protocol [2]. The second technique forgoes some consistency for greater availability and fault tolerance. An example of this technique is the Dynamo database system, which is designed for the highest availability possible [3]. This technique cannot guarantee consistency, so the application must resolve conflicts itself.

2. BACKGROUND

2.1 The CAP Theorem

The CAP theorem, coined by Eric Brewer at UC Berkeley, is the end result of many years of distributed database research. The increasingly distributed nature of the World Wide Web made it clear that ACID would not be tolerable for highly available internet applications. CAP, which stands for Consistency, Availability, and Partition-tolerant, was the new term coined to describe this class of databases. Brewer theorized that only two of these are possible at one time - a theorem later to be proved by another group of researchers [6].

2.2 Difference between ACID consistency and CAP consistency

CAP consistency is a strict subset of ACID consistency; specifically, CAP consistency is single-copy ACID consistency [4]. This implies that CAP does not require that a key be unique at any time, but only when the database is in a single state: when the database system is not partitioned. Applications that require unique keys and other forms of traditional consistency will not be satisfied in this type of environment without intervention by reconciliation logic, which is highly application-specific [3]. This would still involve possibly duplicate keys during partitions, and the application would have to resolve any duplicates appropriately when the system recovers [1]. Generally, however, scalable applications that must be made available to millions of users simultaneously do not require these types of consistency.

As the new century began, certain high-capacity database systems began to experiment with this theorem. In particular, it became clear that although it was not possible to completely guarantee all three attributes of CAP, it was possible to do so under certain circumstances. This was referred to in literature as the Weak CAP Principle [4]. In particular, the proof of the CAP theorem assumes that each node has no way to determine how much time has elapsed [6]. Using this, we can say it is impossible to know whether a server is down or whether it just hasn't gotten back to us yet. Therefore, to be completely consistent we must potentially wait forever for a response, sacrificing availability [1]. In reality, however, this does not happen. Certain assumptions can be made, and acted upon, to prioritize one of the

three features over the other two, without completely sacrificing any. In particular, waiting a finite amount of time before determining that a server is indeed dead is a way to sacrifice some availability while keeping partition tolerance.

3. VOTING SYSTEMS FOR DISTRIBUTED DATABASES

The earliest applications of distributed databases, in the late 70s, handled consistency concerns using voting systems. In this type of system, each node on a distributed cluster was allocated some number of "write" votes and some number of "read" votes [5]. This guarantees that every sequence of operations will have at least one node in common, so every write or read will get the most recent data from at least one node. Then, a minimum number of votes would be required to read or write a value. For any of the distributed nodes to read a value, they would first have to contact R nodes, where R is the read quorum, to ensure that the value returned is the most recent. Complete consistency in this model can be achieved by having the read quorum plus the write quorum be greater than the number of machines in the cluster: $R + W \geq N$.

3.1 The Basic Algorithm

The essential algorithm is simple. To perform a read operation, a system using weighted voting would simply collect a read quorum, place a read lock on the data being accessed, and read from the fastest node available. We will discuss gathering a read and write quorum shortly. For a write, we collect a write quorum, place a write lock on all participating nodes, and write the new values in parallel to all participating copies [5].

3.2 Versioning

To keep track of the most recent data, a version number is passed while attempting to obtain a quorum. Any quorum will be ensured to have at least one node with the most recent version number, and any quorum obtained must contain only machines with the most recent version number. Any node with an outdated version number cannot participate in the quorum because it does not have the most recent data [5]. If a node dies and reboots or gets removed from the network for some reason, when it reconnects it must clear its version number to indicate that it has no idea what the most recent data is.

3.3 Obtaining a Quorum

Read quorums and write quorums are similar except in the case of updating outdated nodes. A read or write quorum is obtained by asking all nodes for data representing its current state. If the state of a majority of nodes indicate that the proper quorum is present and up to date, then the read or write proceeds. If not, several things can happen. In the case of a read quorum, the database must simply wait. This is one major downside of this system in comparison to a Paxos based system [2]. The Paxos algorithm makes a concerted effort to propagate changes across nodes in a safe and consistent manner once a change has been accepted, while the weighted voting algorithm must obtain a read quorum to do anything at all [5]. In fact, this system can easily be deadlocked by high loads because no process makes a concerted effort to resolve a deadlock with the exception of killing all transactions involved, in which case the operations will simply be retried again. Paxos, on the other hand, guarantees that consensus can be reached when given enough time [8].

In the case where a read quorum can be reached but a write quorum cannot, the database attempts to update the outdated nodes in the cluster until a write quorum is reached. Since a read quorum has been reached, we know the most recent version number. We then check each node to ensure it has the latest version and propagate changes from a node in the read quorum when we find one that is not updated [5].

3.4 Weak Consistency

It is possible to have $R + W < N$ in this model, where N is the number of machines in the cluster. However, this cannot guarantee consistency because it is possible for two distinct sets of nodes to be contacted for a conflicting write [5]. Unlike in an system such as Dynamo, this is incredibly unsafe unless the user is unable to create inconsistency in the data [3]. In most cases, some other process must be responsible for working out conflicts [1]. Generally applications do this for performance and availability reasons. In a system where performance is critical, forcing your application to contact more than half of the nodes in a cluster can cause serious performance issues, especially if more than $(N - R)$ nodes have stopped responding or are exceedingly slow [9].

4. DATABASE-LEVEL CONSISTENCY

In applications where consistency is absolutely required, database level consistency is still supreme. Applications like Chubby require database systems which are highly scalable, but still require very high degrees of database consistency and durability. The class of applications like Chubby all rely on voting systems as discussed earlier, but these voting systems did not handle conflicting actions or failures well [9].

4.1 The Paxos Algorithm

The Paxos algorithm provides a guarantee of consensus within a partitioned set of nodes. Although no system can guarantee that a set of nodes will progress in a partitioned, asynchronous environment [6], Paxos provides a guarantee of consistency, at the expense of availability in a small subset of cases [7], but as stated earlier, will make a concerted effort to avoid deadlocks. In contrast to a master/slave style system, Paxos provides a much more highly available environment. The algorithm provides, in general, a way for multiple nodes to send messages, where only one is accepted by the quorum in a safe manner.

Formally, one run of the Paxos algorithm ensures that [8]:

1. Only a proposed value may be chosen,
2. Only a single value is chosen,
3. and a process only ever learns of a value that has actually been chosen.

These items are our safety requirements for the algorithm. Several values may be proposed by different nodes, but the algorithm guarantees that only one will be chosen, even in

the midst of failures (up to $N/2-1$ failures, where N is the number of nodes) [7].

In the Paxos algorithm, there several roles a node can play: an Acceptor, a Proposer, a Learner, or a Leader [8]. A server node could act as several different participants, depending on the current configuration. A proposer is a node that proposes values to acceptors. The acceptors are the groups of nodes which participate in a quorum, that provide fault-tolerant storage for each proposal that is accepted. A leader is a "distinguished proposer" [7] that is the only node allowed to submit proposals. There may be more than one node that believes it is a leader, but only one leader can exist before progress can be made.

4.1.1 Algorithm Details

The most basic of Paxos protocols involves four steps between the Proposer and the Acceptor. The first step involves the proposer choosing a unique sequence number. In Chubby's back end, this proposal number is essentially the sequence number of a particular Leader node [2]. Therefore, this "proposal" step is only used when the Leader changes from one node to another. This will be discussed in more detail later on. After a unique sequence number is chosen, the proposer sends a "proposal" message to the acceptor asking the acceptor to invalidate all requests with a proposal number less than the currently chosen [8].

In the second step, the acceptors reply to the "proposal" messages sent earlier. If an acceptor has not seen a higher sequence number than the one in the message, it replies with the currently accepted value and its sequence number [8]. Then the acceptor must reject any further messages with smaller sequence numbers, which in Chubby's case could be a replica acting as the leader that has failed previously, but has come back up [2].

The third step then involves the proposer again. It will receive responses back from the acceptors if it has the most recent sequence number. If a quorum is reached, the proposer is free to send an "accept" message to the acceptors asking that the proposal be accepted. Because the acceptors have returned a value with the "proposal" message response, the proposers must choose the most recent one and return only its value with its "accept" message [8]. In this fashion, it does not matter which quorum of agents we receive between requests, because we will make sure they are all up to date with the "accept" message. This is a usually redundant operation that has significant advantages over simple weighted voting, where we wait for node responses before attempting to ensure that each is up to date [5].

In the final step, the acceptor simply accepts the proposal as long as the acceptor has not already received a higher sequence number in a "proposal" message [8]. After a proposal is accepted, there are several Learners who broadcast the newest message to nodes not participating in the quorum, including other proposers.

4.1.2 Proof of Consistency

The proof of consistency of the basic Paxos protocol is available, but it is lengthy and will not be discussed here [7]. However, to convince the reader that Paxos actually works, we will present a non-trivial example here. In this example, imagine that we have four nodes, with two currently acting as proposers and two as acceptors. Thus a quorum is three nodes. No values are selected, but both proposers currently believe that they are each leaders. Now imagine that the following sequence of events occurs:

1. Proposer 1 has sent a proposal message to both acceptors with sequence number 1. (Step 1)
2. Both acceptors agree to ignore messages with sequence numbers less than 1, and returns nothing as the last accepted value. (Step 2)
3. Proposer 2 sends a proposal message to both acceptors with sequence number 2. (Step 1)
4. Only acceptor 1 receives the message sent by proposer 2, and agrees to ignore messages with sequence numbers less than 2. Returns nothing as the last accepted value. (Step 2)
5. Proposer 1 acknowledges messages sent by the acceptors and sends an accept message with sequence number 1 and value "B", because no value was previously chosen (Step 3)
6. Acceptor 2 accepts value "B"; acceptor 1 rejects because it has guaranteed not to accept sequence numbers less than 2. (Step 4)
7. Acceptor 2 passes the new value to the learners, who distribute the value to other nodes.
8. Proposer 2 tries to get its value accepted in the next iteration of Paxos, if we're using Multi-Paxos.

In this example, two proposers try to simultaneously submit a value, and the request from proposer 2 to acceptor 2 is lost for some reason. Proposer 1 still tries to send an "accept" message, and although it is only accepted by one node (Acceptor 2), the value is still consistent because proposer 2 failed to obtain a quorum.

4.2 Chubby's Fault-Tolerant Database

As previously stated, Chubby uses a fault-tolerant data store that uses the Paxos consensus algorithm. Chubby implements Multi-Paxos, which is simply the Paxos algorithm repeated continuously. This works perfectly with the architecture of Chubby's fault-tolerant DB, which is kept consistent using a fault-tolerant log. This log is the system which actually implements the Paxos algorithm [2]. In relation to Paxos, a "value" is a log message indicating some action the database took. So once a log message has been accepted by Paxos, the replica is free to write the log entry to the database.

Paxos is also used to elect a leader of the replicas (during steps 1 and 2). That replica responds to all requests to keys within its range. However, unlike a master/slave configuration, failure of the master does not affect the operation of the database in the slightest. With the Paxos algorithm, in the event that the leader failed, another proposer would simply take over for it after a finite period of time [2].

4.2.1 Paxos Modifications

Chubby's implementors decided to make several modifications of the Paxos algorithm to fit their needs. The Chubby log implementation uses the following set of steps for Paxos [2]:

1. Elect a replica to be the coordinator (steps 1 and 2)
2. The new coordinator selects a value and broadcasts it to all replicas (steps 3 and 4)
3. Once a majority of nodes have acknowledged the coordinator, the coordinator sends a commit message (new step)

Electing a replica to be a coordinator is essentially the same as a node receiving promise messages from a quorum of acceptors (step 2). The new item is a step after the traditional Paxos algorithm, and is done to assure that a quorum of nodes have actually persisted the item to disk, in case a majority of nodes fail.

Additionally, once a node has been assigned as a coordinator, it can continue to use its sequence number to submit values in subsequent iterations of Paxos. Additionally, a coordinator can ignore steps 1 and 2 of the Paxos algorithm on subsequent iterations [2]. This allows Chubby to save time while processing log entry messages: each step of Chubby's Paxos algorithm requires a total of 5 sequential writes, representing each step in the algorithm. Allowing a continuous leader to skip the first two steps saves two sequential writes.

Another modification made is the addition of "master leases" [2]. Since reads dominate the number of requests within Chubby, Paxos requires a high overhead to run, compounded with the fact that the master/leader changed frequently. Thus, Chubby's designers provided a mechanism that allows the coordinator to hold its authority for a specified amount of time. Paxos is modified to simply refuse to respond to propose messages during this time [2].

4.2.2 Consistency Versus Availability in Chubby

Obviously, Chubby requires a high level of consistency over availability, because it is not focused on client satisfaction. However, Chubby must still agree on a value at some point. Master leases exemplify this issue. It is assumed that the lease time is some significant amount, say 10 seconds. If the node with the lease fails or is cut off from the network, the other nodes in the same cluster are forced to wait up to 10 seconds until the lease expires. Latency of this sort for one failed node would be absolutely infeasible for a database like Dynamo, which requires similar availability to 99.9% of its users [3].

5. APPLICATION-LEVEL CONSISTENCY

Many recent data stores, including Amazon's Dynamo, work by writing and reading data immediately and resolving consistency issues later [3]. There are many advantages to this model that Amazon exploited to engineer Dynamo to fit their needs. These types of stores, at their core, still use a type of voting system to determine consistency. However, they are usually implemented using weak consistency - that is, using $R + W < N$ [9]. Dynamo allows the application engineer to select the values for R and W, providing a highly configurable environment that fits many of Amazon's needs [3].

5.1 Achieving Application Consistency

Leaving applications to manage their own consistency has its upsides and downsides. It is not trivial to choose to manage consistency at the application level, because this requires significantly more work than handling consistency at the database level.

Let's take the example of an ATM, which is discussed briefly in [1]. This is a classic example of a partitioned network of nodes. Because we're a partitioned network, we have to choose between guaranteed consistency and guaranteed availability [6]. If we choose the former, we are guaranteed that a user of the ATM will never be able to withdraw past their available balance: if the machine dispenses money, it will have contacted a central server and determined if the user did indeed have enough money. If we choose the latter, the user will always be able to withdraw money from his or her account: if we lose access to the network, we believe the user's claim that enough money is in their account.

Both of these options are feasible for an ATM under certain restrictions that must be determined by the system designers. But from a business standpoint, it is much more desireable to guarantee availability [1]. Why? The user being denied money when the funds are available is worse for business than a user being given money they don't have. This is more or less obvious when you consider the facts that a user can only withdraw a maximum amount of money (usually $200) and that a user's identity is always logged on the machine when money is withdrawn. There is a limited effect on the business when the user withdraws too much (because he or she is simply charged a fee), but a higher effect when a user is denied a withdrawal (because he or she will complain about the bank) [1].

Therefore, an application that chooses to use application-based consistency must be able to deal with data inconsistencies in an acceptable fashion. Applications that rely on real-time data to make decisions, such as a financial market application for example, would never be acceptable relying on application based consistency - if there's a partition and some data is stale or unavailable, we should just wait until the data is available.

5.2 Dynamo

5.2.1 Strategy

Dynamo makes heavy use of application consistency. One of the goals of the system is to allow the applications themselves to perform design tradeoffs between consistency, durability, availability, and performance [9]. Dynamo provides a very simple get-put interface to meet the design requirements of the applications it services, which works well because the applications it targets are simple but require extremely high availability [3]. Most applications which use Dynamo face the World Wide Web, and attempt to make applications that seem to be one single system, but simultaneously provide stable availability and performance to 99.9%

of its users [3]. In an application that faces millions of users a day, Amazon must ensure that every user has a similar experience.

In order to ensure that writes are never rejected (which causes a bad user experience), Dynamo was forced to push the complexity of conflict resolution on subsequent reads [3]. If writes are made in isolation and a partition is then repaired, there are suddenly two data items that occupy the same key. Then, when the application reads the key again, the conflict returned as a set of two values for one key. Dynamo offers conflict resolution on both the database and the application, although conflict resolution on the database is usually limited to strategies such as "last write wins" or "include all values, create new keys for duplicates." The latter would be a suitable solution for a database consisting simply of a union of log files: we want all writes to be kept, no matter what.

5.2.2 Architecture

Amazon's base architecture is made failure tolerant by a quorum scheme, as described earlier [5]. Since conflict resolution isn't an issue for Dynamo, a quorum scheme is good enough. Like Couchbase, Dynamo uses a consistent hashing function to distribute keys around a set of nodes of size N. This easily supports incremental scalability, the addition or removal of one node at a time, because they hashing function can be redistributed around the nodes with a simple calculation change.

Dynamo's architecture assumes a ring-like structure of nodes that represents the hash function. The highest value hash wraps around to the lowest value hash, representing the "ring" [3]. If we have N nodes in the ring, every key will be replicated on K nodes around the ring. The node Q mapped to by a key h is the 1st replica for key h, and the node $Q + K$ represents the last replica.

5.2.3 Data Versioning

At any point in time, a partition may occur in the database, resulting in two or more distinct sets of nodes. In order to provide a guarantee that no put(...) operation is ever lost or overwritten when relying on application-level consistency, Dynamo must treat each modification as a new version of the data [3]. Thus obviously multiple copies can be in the system at the same time, even if no partition has occurred. Remember that Dynamo can be configured such that $R + W < N$, so if a read does not contact a node where a write was just saved for the same data, stale data will be returned.

In these cases, using application-level consistency, the application itself must consolidate the multiple versions. As stated previously, multiple versions are returned as a set. However, it is not necessarily this simple. Dynamo stores a context with each put(...) operation, which stores the vector clocks for each version [3]. Essentially, a vector clock is a list of updates that have been performed on a certain version. Updating from one version will result in simply tacking on another item to the vector clock.

Obviously, not all writes require application intervention to resolve. One can easily tell if the version being saved is a direct ancestor of the previous version by making sure the previous version's clock is a subset of the version being saved. If not, we must forward both versions and their version numbers to the application, who specifies which version is being updated [3]. An update to a context containing two divergent versions is assumed to have reconciled the split and is saved as one new version [3].

5.2.4 A Use Case

The flagship Dynamo application is Amazon's shopping cart. The key attributes of this application is that an item should never be dropped from a cart without an explicit delete command from the user [3]. To achieve this, however, the application may keep items that were previously deleted by the user [3].

6. CONCLUSION

By examining several types of consistency constraints in recent database architectures, we've highlighed a few of the different ways that consistency is maintained while working with distributed systems. Certainly there are some applications which can handle stale data and some that cannot.

When working with systems that must have consistency guarantees, the Paxos algorithm assures that these applications can run in a partition-tolerant environment and still retain some, although limited, availability. Even though Paxos would not work well for any system which directly faces the user, it is very well suited for applications that do background work - exactly the kinds of applications that Chubby serves.

However, when working with systems that do not require consistency guarantees and will allow application-based conflict resolution, an architecture like Dynamo is a solid option. Although the work to make the application consistent is dropped onto the programmer, Dynamo has proved that an extraordinary level of availability is possible, which in some cases makes up for the extra work.

7. REFERENCES

[1] E. Brewer. Cap twelve years later: How the"rules" have changed. *Computer*, 45(2):23–29, 2012.

[2] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live-an engineering perspective (2006 invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing-PODC*, volume 7, 2007.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[4] A. Fox and E. Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 174–178. IEEE, 1999.

[5] D. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[6] 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.

[7] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[8] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[9] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.