Distributed systems

for fun and profit

Previous Chapter | Home | Next Chapter

5. Replication: weak consistency model protocols

Now that we've taken a look at protocols that can enforce single-copy consistency under an increasingly realistic set of supported failure cases, let's turn our attention at the world of options that opens up once we let go of the requirement of single-copy consistency.

By and large, it is hard to come up with a single dimension that defines or characterizes the protocols that allow for replicas to diverge. Most such protocols are highly available, and the key issue is more whether or not the end users find the guarantees, abstractions and APIs useful for their purpose in spite of the fact that the replicas may diverge when node and/or network failures occur.

Why haven't weakly consistent systems been more popular?

As I stated in the introduction, I think that much of distributed programming is about dealing with the implications of two consequences of distribution:

- that information travels at the speed of light
- that independent things fail independently

The implication that follows from the limitation on the speed at which information travels is that nodes experience the world in different, unique ways. Computation on a single node is easy, because everything happens in a predictable global total order. Computation on a distributed system is difficult, because there is no global total order.

For the longest while (e.g. decades of research), we've solved this problem by introducing a global total order. I've discussed the many methods for achieving strong consistency by creating order (in a fault-tolerant manner) where there is no naturally occurring total order.

Of course, the problem is that enforcing order is expensive. This breaks down in particular with large scale internet systems, where a system needs to remain available. A system enforcing strong consistency doesn't behave like a distributed system: it behaves like a single system, which is bad for availability during a partition.

Furthermore, for each operation, often a majority of the nodes must be contacted - and often not just once, but twice (as you saw in the discussion on 2PC). This is particularly painful in systems that need to be geographically distributed to provide adequate performance for a global user base.

So behaving like a single system by default is perhaps not desirable.

Perhaps what we want is a system where we can write code that doesn't use expensive coordination, and yet returns a "usable" value. Instead of having a single truth, we will allow different replicas to diverge from each other - both to keep things efficient but also to tolerate partitions - and then try to find a way to deal with the divergence in some manner.

Eventual consistency expresses this idea: that nodes can for some time diverge from each other, but that eventually they will agree on the value.

Within the set of systems providing eventual consistency, there are two types of system designs:

Eventual consistency with probabilistic guarantees. This type of system can detect conflicting writes at some later point, but does not guarantee that the results are equivalent to some correct sequential execution. In other words, conflicting updates will sometimes result in overwriting a newer value with an older one and some anomalies can be expected to occur during normal operation (or during partitions).

In recent years, the most influential system design offering single-copy consistency is Amazon's Dynamo, which I will discuss as an example of a system that offers eventual consistency with probabilistic guarantees.

Eventual consistency with strong guarantees. This type of system guarantees that the results converge to a common value equivalent to some correct sequential execution. In other words, such systems do not produce in anomalous results; without any coordination you can build replicas of the same service, and those replicas can communicate in any pattern and receive the updates in any order, and they will eventually agree on the end result as long as they all see the same information.

CRDT's (convergent replicated data types) are data types that guarantee convergence to the same value in spite of network delays, partitions and message reordering. They are provably convergent, but the data types that can be implemented as CRDT's are limited.

The CALM (consistency as logical monotonicity) conjecture is an alternative expression of the same principle: it equates logical monotonicity with convergence. If we can conclude that something is logically monotonic, then it is also safe to run without coordination. Confluence analysis - in particular, as applied for the Bloom programming language - can be used to guide programmer decisions about when and where to use the coordination techniques from strongly consistent systems and when it is safe to execute without coordination.

Reconciling different operation orders

What does a system that does not enforce single-copy consistency look like? Let's try to make this more concrete by looking a few examples.

Perhaps the most obvious characteristic of systems that do not enforce single-copy consistency is that they allow replicas to diverge from each other. This means that there is no strictly defined pattern of communication: replicas can be separated from each other and yet continue to be available and accept writes.

Let's imagine a system of three replicas, each of which is partitioned from the others. For example, the replicas might be in different datacenters and for some reason unable to communicate. Each replica remains available during the partition, accepting both reads and writes from some set of clients:

```
[Clients]   - > [A]

--- Partition ---

[Clients]   - > [B]

--- Partition ---

[Clients]   - > [C]
```

After some time, the partitions heal and the replica servers exchange information. They have received different updates from different clients and have diverged each other, so some sort of reconciliation needs to take place. What we would like to happen is that all of the replicas converge to the same result.

```
[A] \
    --> [merge]
[B] /     |
          |
[C] ----[merge]---> result
```

Another way to think about systems with weak consistency guarantees is to imagine a set of clients sending messages to two in some order. Because there is no coordination protocol that enforces a single total order, the messages can get delivered in different orders at the two replicas:

```
[Clients]  --> [A]  1, 2, 3
[Clients]  --> [B]  2, 3, 1
```

This is, in essence, the reason why we need coordination protocols. For example, assume that we are trying to concatenate a string and the operations in messages 1, 2 and 3 are:

```
1: { operation: concat('Hello ') }
2: { operation: concat('World') }
3: { operation: concat('!') }
```

Then, without coordination, A will produce "Hello World!", and B will produce "World!Hello ".

```
A: concat(concat(concat('', 'Hello '), 'World'), '!') = 'Hello World!'
B: concat(concat(concat('', 'World'), '!'), 'Hello ') = 'World!Hello '
```

This is, of course, incorrect. Again, what we'd like to happen is that the replicas converge to the same result.

Keeping these two examples in mind, let's look at Amazon's Dynamo first to establish a baseline, and then discuss a number of novel approaches to building systems with weak consistency guarantees, such as CRDT's and the CALM theorem.

Amazon's Dynamo

Amazon's Dynamo system design (2007) is probably the best-known system that offers weak consistency guarantees but high availability. It is the basis for many other real world systems, including LinkedIn's Voldemort, Facebook's Cassandra and Basho's Riak.

Dynamo is an eventually consistent, highly available key-value store. A key value store is like a large hash table: a client can set values via `set(key, value)` and retrieve them by key using `get(key)`. A Dynamo cluster consists of N peer nodes; each node has a set of keys which is it responsible for storing.

Dynamo prioritizes availability over consistency; it does not guarantee single-copy consistency. Instead, replicas may diverge from each other when values are written; when a key is read, there is a read reconciliation phase that attempts to reconcile differences between replicas before returning the value back to the client.

For many features on Amazon, it is more important to avoid outages than it is to ensure that data is perfectly consistent, as an outage can lead to lost business and a loss of credibility. Furthermore, if the data is not particularly important, then a weakly consistent system can provide better performance and higher availability at a lower cost than a traditional RDBMS.

Since Dynamo is a complete system design, there are many different parts to look at beyond the core replication task. The diagram below illustrates some of the tasks; notably, how a write is routed to a node and written to multiple replicas.

```
[ Client ]
    |
( Mapping keys to nodes )
    |
    V
[ Node A ]
    |     \
( Synchronous replication task: minimum durability )
    |        \
[ Node B]  [ Node C ]
    A
    |
( Conflict detection; asynchronous replication task:
  ensuring that partitioned / recovered nodes recover )
    |
    V
[ Node D]
```

After looking at how a write is initially accepted, we'll look at how conflicts are detected, as well as the asynchronous replica synchronization task. This task is needed because of the high availability design, in which nodes may be temporarily unavailable (down or partitioned). The replica synchronization task ensures that nodes can catch up fairly rapidly even after a failure.

Consistent hashing

Whether we are reading or writing, the first thing that needs to happen is that we need to locate where the data should live on the system. This requires some type of key-to-node mapping.

In Dynamo, keys are mapped to nodes using a hashing technique known as consistent hashing (which I will not discuss in detail). The main idea is that a key can be mapped to a set of nodes responsible for it by a simple calculation on the client. This means that a client can locate keys without having to query the system for the location of each key; this saves system resources as hashing is generally faster than performing a remote procedure call.

Partial quorums

Once we know where a key should be stored, we need to do some work to persist the value. This is a synchronous task; the reason why we will immediately write the value onto multiple nodes is to provide a higher level of durability (e.g. protection from the immediate failure of a node).

Just like Paxos or Raft, Dynamo uses quorums for replication. However, Dynamo's quorums are sloppy (partial) quorums rather than strict (majority) quorums.

Informally, a strict quorum system is a quorum system with the property that any two quorums (sets) in the quorum system overlap. Requiring a majority to vote for an update before accepting it guarantees that only a single history is admitted since each majority quorum must overlap in at least one node. This was the property that Paxos, for example, relied on.

Partial quorums do not have that property; what this means is that a majority is not required and that different subsets of the quorum may contain different versions of the same data. The user can choose the number of nodes to write to and read from:

- the user can choose some number W-of-N nodes required for a write to succeed; and
- the user can specify the number of nodes (R-of-N) to be contacted during a read.

`W` and `R` specify the number of nodes that need to be involved to a write or a read. Writing to more nodes makes writes slightly slower but increases the probability that the value is not lost; reading from more nodes increases the probability that the value read is up to date.

The usual recommendation is that `R + W > N`, because this means that the read and write quorums overlap in one node - making it less likely that a stale value is returned. A typical configuration is `N = 3` (e.g. a total of three replicas for each value); this means that the user can choose between:

```
R = 1, W = 3;
R = 2, W = 2 or
R = 3, W = 1
```

More generally, again assuming `R + W > N`:

- `R = 1`, `W = N`: fast reads, slow writes
- `R = N`, `W = 1`: fast writes, slow reads
- `R = N/2` and `W = N/2 + 1`: favorable to both

N is rarely more than 3, because keeping that many copies of large amounts of data around gets expensive!

As I mentioned earlier, the Dynamo paper has inspired many other similar designs. They all use the same partial quorum based replication approach, but with different defaults for N, W and R:

- Basho's Riak (N = 3, R = 2, W = 2 default)
- Linkedin's Voldemort (N = 2 or 3, R = 1, W = 1 default)
- Apache's Cassandra (N = 3, R = 1, W = 1 default)

There is another detail: when sending a read or write request, are all N nodes asked to respond (Riak), or only a number of nodes that meets the minimum (e.g. R or W; Voldemort). The "send-to-all" approach is faster and less sensitive to latency (since it only waits for the fastest R or W nodes of N) but also less efficient, while the "send-to-minimum" approach is more sensitive to latency (since latency communicating with a single node will delay the operation) but also more efficient (fewer messages / connections overall).

What happens when the read and write quorums overlap, e.g. (`R + W > N`)? Specifically, it is often claimed that this results in "strong consistency".

Is R + W > N the same as "strong consistency"?

No.

It's not completely off base: a system where `R + W > N` can detect read/write conflicts, since any read quorum and any write quorum share a member. E.g. at least one node is in both quorums:

```
  1     2   N/2+1     N/2+2    N
[...] [R]  [R + W]   [W]    [...]
```

This guarantees that a previous write will be seen by a subsequent read. However, this only holds if the nodes in N never change. Hence, Dynamo doesn't qualify, because in Dynamo the cluster membership can change if nodes fail.

Dynamo is designed to be always writable. It has a mechanism which handles node failures by adding a different, unrelated server into the set of nodes responsible for certain keys when the original server is down. This means that the quorums are no longer guaranteed to always overlap. Even `R = W = N` would not qualify, since while the quorum sizes are equal to N, the nodes in those quorums can change during a failure. Concretely, during a partition, if a sufficient number of nodes cannot be reached, Dynamo will add new nodes to the quorum from unrelated but accessible nodes.

Furthermore, Dynamo doesn't handle partitions in the manner that a system enforcing a strong consistency model would: namely, writes are allowed on both sides of a partition, which means that for at least some time the system does not act as a single copy. So calling `R + W > N` "strongly consistent" is misleading; the guarantees merely probabilistic - which is not what strong consistency refers to.

Conflict detection and read repair

Systems that allow replicas to diverge must have a way to eventually reconcile two different values. As briefly mentioned during the partial quorum approach, one way to do this is to detect conflicts at read time, and then apply some conflict resolution method. But how is this done?

In general, this is done by tracking the causal history of a piece of data by supplementing it with some metadata. Clients must keep the metadata information when they read data from the system, and must return back the metadata value when writing to the database.

We've already encountered a method for doing this: vector clocks can be used to represent the history of a value. Indeed, this is what the original Dynamo design uses for detecting conflicts.

However, using vector clocks is not the only alternative. If you look at many practical system designs, you can deduce quite a bit about how they work by looking at the metadata that they track.

No metadata. When a system does not track metadata, and only returns the value (e.g. via a client API), it cannot really do anything special about concurrent writes. A common rule is that the last writer wins: in other words, if two writers are writing at the same time, only the value from the slowest writer is kept around.

Timestamps. Nominally, the value with the higher timestamp value wins. However, if time is not carefully synchronized, many odd things can happen where old data from a system with a faulty or fast clock overwrites newer values. Facebook's Cassandra is a Dynamo variant that uses timestamps instead of vector clocks.

Version numbers. Version numbers may avoid some of the issues related with using timestamps. Note that the smallest mechanism that can accurately track causality when multiple histories are possible are vector clocks, not version numbers.

Vector clocks. Using vector clocks, concurrent and out of date updates can be detected. Performing read repair then becomes possible, though in some cases (concurrent changes) we need to ask the client to pick a value. This is because if the changes are concurrent and we know nothing more about the data (as is the case with a simple key-value store), then it is better to ask than to discard data arbitrarily.

When reading a value, the client contacts `R` of `N` nodes and asks them for the latest value for a key. It takes all the responses, discards the values that are strictly older (using the vector clock value to detect this). If there is only one unique vector clock + value pair, it returns that. If there are multiple vector clock + value pairs that have been edited concurrently (e.g. are not comparable), then all of those values are returned.

As is obvious from the above, read repair may return multiple values. This means that the client / application developer must occasionally handle these cases by picking a value based on some use-case specific criterion.

In addition, a key component of a practical vector clock system is that the clocks cannot be allowed to grow forever - so there needs to be a procedure for occasionally garbage collecting the clocks in a safe manner to balance fault tolerance with storage requirements.

Replica synchronization: gossip and Merkle trees

Given that the Dynamo system design is tolerant of node failures and network partitions, it needs a way to deal with nodes rejoining the cluster after being partitioned, or when a failed node is replaced or partially recovered.

Replica synchronization is used to bring nodes up to date after a failure, and for periodically synchronizing replicas with each other.

Gossip is a probabilistic technique for synchronizing replicas. The pattern of communication (e.g. which node contacts which node) is not determined in advance. Instead, nodes have some probability `p` of attempting to synchronize with each other. Every `t` seconds, each node picks a node to communicate with. This provides an additional mechanism beyond the synchronous task (e.g. the partial quorum writes) which brings the replicas up to date.

Gossip is scalable, and has no single point of failure, but can only provide probabilistic guarantees.

In order to make the information exchange during replica synchronization efficient, Dynamo uses a technique called Merkle trees, which I will not cover in detail. The key idea is that a data store can be hashed at multiple different level of granularity: a hash representing the whole content, half the keys, a quarter of the keys and so on.

By maintaining this fairly granular hashing, nodes can compare their data store content much more efficiently than a naive technique. Once the nodes have identified which keys have different values, they exchange the necessary information to bring the replicas up to date.

Dynamo in practice: probabilistically bounded staleness (PBS)

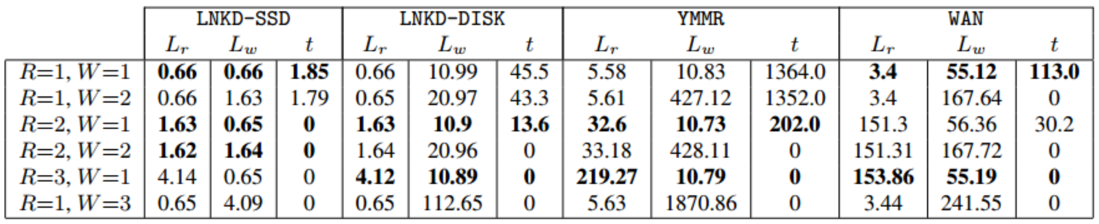
And that pretty much covers the Dynamo system design:

- consistent hashing to determine key placement
- partial quorums for reading and writing
- conflict detection and read repair via vector clocks and
- gossip for replica synchronization

How might we characterize the behavior of such a system? A fairly recent paper from Bailis et al. (2012) describes an approach called PBS (probabilistically bounded staleness) uses simulation and data collected from a real world system to characterize the expected behavior of such a system.

PBS estimates the degree of inconsistency by using information about the anti-entropy (gossip) rate, the network latency and local processing delay to estimate the expected level of consistency of reads. It has been implemented in Cassandra, where timing information is piggybacked on other messages and an estimate is calculated based on a sample of this information in a Monte Carlo simulation.

Based on the paper, during normal operation eventually consistent data stores are often faster and can read a consistent state within tens or hundreds of milliseconds. The table below illustrates amount of time required from a 99.9% probability of consistent reads given different `R` and `W` settings on empirical timing data from LinkedIn (SSD and 15k RPM disks) and Yammer:



	LNKD-SSD			LNKD-DISK			YMMR			WAN		
	L_r	L_w	t	L_r	L_w	t	L_r	L_w	t	L_r	L_w	t
R=1, W=1	**0.66**	**0.66**	**1.85**	0.66	10.99	45.5	5.58	10.83	1364.0	**3.4**	**55.12**	**113.0**
R=1, W=2	0.66	1.63	1.79	0.65	20.97	43.3	5.61	427.12	1352.0	3.4	167.64	0
R=2, W=1	**1.63**	**0.65**	**0**	**1.63**	**10.9**	**13.6**	**32.6**	**10.73**	**202.0**	151.3	56.36	30.2
R=2, W=2	**1.62**	**1.64**	**0**	1.64	20.96	0	33.18	428.11	0	151.31	167.72	0
R=3, W=1	4.14	0.65	0	**4.12**	**10.89**	**0**	**219.27**	**10.79**	**0**	**153.86**	**55.19**	**0**
R=1, W=3	0.65	4.09	0	0.65	112.65	0	5.63	1870.86	0	3.44	241.55	0

Table 4: t-visibility for $p_{st} = .001$ (99.9% probability of consistency for 50,000 reads and writes) and 99.9th percentile read (L_r) and write latencies (L_w) across R and W, N=3 (1M reads and writes). Significant latency-staleness trade-offs are in bold.

For example, going from `R=1`, `W=1` to `R=2`, `W=1` in the Yammer case reduces the inconsistency window from 1352 ms to 202 ms - while keeping the read latencies lower (32.6 ms) than the fastest strict quorum (`R=3`, `W=1`; 219.27 ms).

For more details, have a look at the PBS website and the associated paper.

Disorderly programming

Let's look back at the examples of the kinds of situations that we'd like to resolve. The first scenario consisted of three different servers behind partitions; after the partitions healed, we wanted the servers to converge to the same value. Amazon's Dynamo made this possible by reading from `R` out of `N` nodes and then performing read reconciliation.

In the second example, we considered a more specific operation: string concatenation. It turns out that there is no known technique for making string concatenation resolve to the same value without imposing an order on the operations (e.g. without expensive coordination). However, there are operations which can be applied safely in any order, where a simple register would not be able to do so. As Pat Helland wrote:

> ... operation-centric work can be made commutative (with the right operations and the right semantics) where a simple READ/WRITE semantic does not lend itself to commutativity.

For example, consider a system that implements a simple accounting system with the `debit` and `credit` operations in two different ways:

- using a register with `read` and `write` operations, and
- using a integer data type with native `debit` and `credit` operations

The latter implementation knows more about the internals of the data type, and so it can preserve the intent of the operations in spite of the operations being reordered. Debiting or crediting can be applied in any order, and the end result is the same:

```
100 + credit(10) + credit(20) = 130 and
100 + credit(20) + credit(10) = 130
```

However, writing a fixed value cannot be done in any order: if writes are reordered, the one of the writes will overwrite the other:

```
100 + write(110) + write(130) = 130 but
100 + write(130) + write(110) = 110
```

Let's take the example from the beginning of this chapter, but use a different operation. In this scenario, clients are sending messages to two nodes, which see the operations in different orders:

```
[Clients]  --> [A]  1, 2, 3
[Clients]  --> [B]  2, 3, 1
```

Instead of string concatenation, assume that we are looking to find the largest value (e.g. MAX()) for a set of integers. The messages 1, 2 and 3 are:

```
1: { operation: max(previous, 3) }
2: { operation: max(previous, 5) }
3: { operation: max(previous, 7) }
```

Then, without coordination, both A and B will converge to 7, e.g.:

```
A: max(max(max(0, 3), 5), 7) = 7
B: max(max(max(0, 5), 7), 3) = 7
```

In both cases, two replicas see updates in different order, but we are able to merge the results in a way that has the same result in spite of what the order is. The result converges to the same answer in both cases because of the merge procedure (`max`) we used.

It is likely not possible to write a merge procedure that works for all data types. In Dynamo, a value is a binary blob, so the best that can be done is to expose it and ask the application to handle each conflict.

However, if we know that the data is of a more specific type, handling these kinds of conflicts becomes possible. CRDT's are data structures designed to provide data types that will always converge, as long as they see the same set of operations (in any order).

CRDTs: Convergent replicated data types

CRDTs (convergent replicated datatypes) exploit knowledge regarding the commutativity and associativity of specific operations on specific datatypes.

In order for a set of operations to converge on the same value in an environment where replicas only communicate occasionally, the operations need to be order-independent and insensitive to (message) duplication/redelivery. Thus, their operations need to be:

- Associative (`a+(b+c)=(a+b)+c`), so that grouping doesn't matter
- Commutative (`a+b=b+a`), so that order of application doesn't matter
- Idempotent (`a+a=a`), so that duplication does not matter

It turns out that these structures are already known in mathematics; they are known as join or meet semilattices.

A lattice is a partially ordered set with a distinct top (least upper bound) and a distinct bottom (greatest lower bound). A semilattice is like a lattice, but one that only has a distinct top or bottom. A join semilattice is one with a distinct top (least upper bound) and a meet semilattice is one with a distinct bottom (greatest lower bound).

Any data type that be expressed as a semilattice can be implemented as a data structure which guarantees convergence. For example, calculating the `max()` of a set of values will always return the same result regardless of the order in which the values were received, as long as all values are eventually received, because the `max()` operation is associative, commutative and idempotent.

For example, here are two lattices: one drawn for a set, where the merge operator is `union(items)` and one drawn for a strictly increasing integer counter, where the merge operator is `max(values)`:

```
   { a, b, c }              7
  /      |    \            /  \
{a, b} {b,c} {a,c}        5    7
  |  \  /  | /           /   |   \
  {a} {b} {c}            3   5   7
```

With data types that can be expressed as semilattices, you can have replicas communicate in any pattern and receive the updates in any order, and they will eventually agree on the end result as long as they all see the same information. That is a powerful property that can be guaranteed as long as the prerequisites hold.

However, expressing a data type as a semilattice often requires some level of interpretation. Many data types have operations which are not in fact order-independent. For example, adding items to a set is associative, commutative and idempotent. However, if we also allow items to be removed from a set, then we need some way to resolve conflicting operations, such as `add(A)` and `remove(A)`. What does it mean to remove an element if the local replica never added it? This resolution has to be specified in a manner that is order-independent, and there are several different choices with different tradeoffs.

This means that several familiar data types have more specialized implementations as CRDT's which make a different tradeoff in order to resolve conflicts in an order-independent manner. Unlike a key-value store which simply deals with registers (e.g. values that are opaque blobs from the perspective of the system), someone using CRDTs must use the right data type to avoid anomalies.

Some examples of the different data types specified as CRDT's include:

- Counters
  - Grow-only counter (merge = max(values); payload = single integer)
  - Positive-negative counter (consists of two grow counters, one for increments and another for decrements)
- Registers
  - Last Write Wins -register (timestamps or version numbers; merge = max(ts); payload = blob)

    - Multi-valued -register (vector clocks; merge = take both)
- Sets
    - Grow-only set (merge = union(items); payload = set; no removal)
    - Two-phase set (consists of two sets, one for adding, and another for removing; elements can be added once and removed once)
    - Unique set (an optimized version of the two-phase set)
    - Last write wins set (merge = max(ts); payload = set)
    - Positive-negative set (consists of one PN-counter per set item)
    - Observed-remove set
- Graphs and text sequences (see the paper)

To ensure anomaly-free operation, you need to find the right data type for your specific application - for example, if you know that you will only remove an item once, then a two-phase set works; if you will only ever add items to a set and never remove them, then a grow-only set works.

Not all data structures have known implementations as CRDTs, but there are CRDT implementations for booleans, counters, sets, registers and graphs in the recent (2011) survey paper from Shapiro et al.

Interestingly, the register implementations correspond directly with the implementations that key value stores use: a last-write-wins register uses timestamps or some equivalent and simply converges to the largest timestamp value; a multi-valued register corresponds to the Dynamo strategy of retaining, exposing and reconciling concurrent changes. For the details, I recommend that you take a look at the papers in the further reading section of this chapter.

The CALM theorem

The CRDT data structures were based on the recognition that data structures expressible as semilattices are convergent. But programming is about more than just evolving state, unless you are just implementing a data store.

Clearly, order-independence is an important property of any computation that converges: if the order in which data items are received influences the result of the computation, then there is no way to execute a computation without guaranteeing order.

However, there are many programming models in which the order of statements does not play a significant role. For example, in the MapReduce model, both the Map and the Reduce tasks are specified as stateless tuple-processing tasks that need to be run on a dataset. Concrete decisions about how and in what order data is routed to the tasks is not specified explicitly, instead, the batch job scheduler is responsible for scheduling the tasks to run on the cluster.

Similarly, in SQL one specifies the query, but not how the query is executed. The query is simply a declarative description of the task, and it is the job of the query optimizer to figure out an efficient way to execute the query (across multiple machines, databases and tables).

Of course, these programming models are not as permissive as a general purpose programming language. MapReduce tasks need to be expressible as stateless tasks in an acyclic dataflow program; SQL statements can execute fairly sophisticated computations but many things are hard to express in it.

However, it should be clear from these two examples that there are many kinds of data processing tasks which are amenable to being expressed in a declarative language where the order of execution is not explicitly specified. Programming models which express a desired result while leaving the exact order of statements up to an optimizer to decide often have semantics that are order-independent. This means that such programs may be possible to execute without coordination, since they depend on the inputs they receive but not necessarily the specific order in which the inputs are received.

The key point is that such programs may be safe to execute without coordination. Without a clear rule that characterizes what is safe to execute without coordination, and what is not, we cannot implement a program while remaining certain that the result is correct.

This is what the CALM theorem is about. The CALM theorem is based on a recognition of the link between logical monotonicity and useful forms of eventual consistency (e.g. confluence / convergence). It states that logically monotonic programs are guaranteed to be eventually consistent.

Then, if we know that some computation is logically monotonic, then we know that it is also safe to execute without coordination.

To better understand this, we need to contrast monotonic logic (or monotonic computations) with non-monotonic logic (or non-monotonic computations).

Monotony if sentence φ is a consequence of a set of premises Γ, then it can also be inferred from any set Δ of premises extending Γ

Most standard logical frameworks are monotonic: any inferences made within a framework such as first-order logic, once deductively valid, cannot be invalidated by new information. A non-monotonic logic is a system in which that property does not hold - in other words, if some conclusions can be invalidated by learning new knowledge.

Within the artificial intelligence community, non-monotonic logics are associated with defeasible reasoning - reasoning, in which assertions made utilizing partial information can be invalidated by new knowledge. For example, if we learn that Tweety is a bird, we'll assume that Tweety can fly; but if we later learn that Tweety is a penguin, then we'll have to revise our conclusion.

Monotonicity concerns the relationship between premises (or facts about the world) and conclusions (or assertions about the world). Within a monotonic logic, we know that our results are retraction-free: monotone computations do not need to be recomputed or coordinated; the answer gets more accurate over time. Once we know that Tweety is a bird (and that we're reasoning using monotonic logic), we can safely conclude that Tweety can fly and that nothing we learn can invalidate that conclusion.

While any computation that produces a human-facing result can be interpreted as an assertion about the world (e.g. the value of "foo" is "bar"), determining whether a computation in a von Neumann -machine based programming model is monotonic difficult because it is not exactly clear what the relationship between facts and assertions are and whether those relationships are monotonic.

However, there are a number of programming models for which determining monotonicity is possible. In particular, relational algebra (e.g. the theoretical underpinnings of SQL) and Datalog provide highly expressive languages that have well-understood interpretations.

Both basic Datalog and relational algebra (even with recursion) are known to be monotonic. More specifically, computations expressed using a certain set of basic operators are known to be monotonic (selection, projection, natural join, cross product, union and recursive Datalog without negation), and non-monotonicity is introduced by using more advanced operators (negation, set difference, division, universal quantification, aggregation).

This means that computations expressed using a significant number of operators (e.g. map, filter, join, union, intersection) in those systems are logically monotonic; any computations using those operators are also monotonic and thus safe to run without coordination. Expressions that make use of negation and aggregation, on the other hand, are not safe to run without coordination.

It is important to realize the connection between non-monotonicity and operations that are expensive to perform in a distributed system. Specifically, both distributed aggregation and coordination protocols can be considered to be a form of negation. As Joe Hellerstein writes:

> To establish the veracity of a negated predicate in a distributed setting, an evaluation strategy has to start "counting to 0" to determine emptiness, and wait until the distributed counting process has definitely terminated. Aggregation is the generalization of this idea.

and:

> This idea can be seen from the other direction as well. Coordination protocols are themselves aggregations, since they entail voting: Two-Phase Commit requires unanimous votes, Paxos consensus requires majority votes, and Byzantine protocols require a 2/3 majority. Waiting requires counting.

If, then we can express our computation in a manner in which it is possible to test for monotonicity, then we can perform a whole-program static analysis that detects which parts of the program are eventually consistent and safe to run without coordination (the monotonic parts) - and which parts are not (the non-monotonic ones).

Note that this requires a different kind of language, since these inferences are hard to make for traditional programming languages where sequence, selection and iteration are at the core. Which is why the Bloom language was designed.

What is non-mononicity good for?

The difference between monotonicity and non-monotonicity is interesting. For example, adding two numbers is monotonic, but calculating an aggregation over two nodes containing numbers is not. What's the difference? One of these is a computation (adding two numbers), while the other is an assertion (calculating an aggregate).

How does a computation differ from an assertion? Let's consider the query "is pizza a vegetable?". To answer that, we need to get at the core: when is it acceptable to infer that something is (or is not) true?

There are several acceptable answers, each corresponding to a different set of assumptions regarding the information that we have and the way we ought to act upon it - and we've come to accept different answers in different contexts.

In everyday reasoning, we make what is known as the open-world assumption: we assume that we do not know everything, and hence cannot make conclusions from a lack of knowledge. That is, any sentence may be true, false or unknown.

```
                                OWA +                 |  OWA +
                                Monotonic logic       |  Non-monotonic logic
Can derive P(true)        |  Can assert P(true)      |  Cannot assert P(true)
Can derive P(false)       |  Can assert P(false)     |  Cannot assert P(true)
Cannot derive P(true)     |  Unknown                 |  Unknown
or P(false)
```

When making the open world assumption, we can only safely assert something we can deduce from what is known. Our information about the world is assumed to be incomplete.

Let's first look at the case where we know our reasoning is monotonic. In this case, any (potentially incomplete) knowledge that we have cannot be invalidated by learning new knowledge. So if we can infer that a sentence is true based on some deduction, such as "things that contain two tablespoons of tomato paste are vegetables" and "pizza contains two tablespoons of tomato paste", then we can conclude that "pizza is a vegetable". The same goes for if we can deduce that a sentence is false.

However, if we cannot deduce anything - for example, the set of knowledge we have contains customer information and nothing about pizza or vegetables - then under the open world assumption we have to say that we cannot conclude anything.

With non-monotonic knowledge, anything we know right now can potentially be invalidated. Hence, we cannot safely conclude anything, even if we can deduce true or false from what we currently know.

However, within the database context, and within many computer science applications we prefer to make more definite conclusions. This means assuming what is known as the closed-world assumption: that anything that cannot be shown to be true is false. This means that no explicit declaration of falsehood is needed. In other words, the database of facts that we have is assumed to be complete (minimal), so that anything not in it can be assumed to be false.

For example, under the CWA, if our database does not have an entry for a flight between San Francisco and Helsinki, then we can safely conclude that no such flight exists.

We need one more thing to be able to make definite assertions: logical circumscription. Circumscription is a formalized rule of conjecture. Domain circumscription conjectures that the known entities are all there are. We need to be able to assume that the known entities are all there are in order to reach a definite conclusion.

```
                                CWA +              |  CWA +
                                Circumscription +  |  Circumscription +
                                Monotonic logic    |  Non-monotonic logic
Can derive P(true)      |    Can assert P(true)    |  Can assert P(true)
Can derive P(false)     |    Can assert P(false)   |  Can assert P(false)
```

```
Cannot derive P(true)   |   Can assert P(false)   |  Can assert P(false)
or P(false)
```

In particular, non-monotonic inferences need this assumption. We can only make a confident assertion if we assume that we have complete information, since additional information may otherwise invalidate our assertion.

What does this mean in practice? First, monotonic logic can reach definite conclusions as soon as it can derive that a sentence is true (or false). Second, nonmonotonic logic requires an additional assumption: that the known entities are all there is.

So why are two operations that are on the surface equivalent different? Why is adding two numbers monotonic, but calculating an aggregation over two nodes not? Because the aggregation does not only calculate a sum but also asserts that it has seen all of the values. And the only way to guarantee that is to coordinate across nodes and ensure that the node performing the calculation has really seen all of the values within the system.

Thus, in order to handle nonmonotonicity one needs to either use distributed coordination to ensure that assertions are made only after all the information is known or make assertions with the caveat that the conclusion can be invalidated later on.

Handling non-monotonicity is important for reasons of expressiveness. This comes down to being able to express non-monotone things; for example, it is nice to be able to say that the total of some column is X. The system must detect that this kind of computation requires a global coordination boundary to ensure that we have seen all the entities.

Purely monotone systems are rare. It seems that most applications operate under the closed-world assumption even when they have incomplete data, and we humans are fine with that. When a database tells you that a direct flight between San Francisco and Helsinki does not exist, you will probably treat this as "according to this database, there is no direct flight", but you do not rule out the possibility that that in reality such a flight might still exist.

Really, this issue only becomes interesting when replicas can diverge (e.g. during a partition or due to delays during normal operation). Then there is a need for a more specific consideration: whether the answer is based on just the current node, or the totality of the system.

Further, since nonmonotonicity is caused by making an assertion, it seems plausible that many computations can proceed for a long time and only apply coordination at the point where some result or assertion is passed to a 3rd party system or end user. Certainly it is not necessary for every single read and write operation within a system to enforce a total order, if those reads and writes are simply a part of a long running computation.

The Bloom language

The Bloom language is a language designed to make use of the CALM theorem. It is a Ruby DSL which has its formal basis in a temporal logic programming language called Dedalus.

In Bloom, each node has a database consisting of collections and lattices. Programs are expressed as sets of unordered statements which interact with collections (sets of facts) and lattices (CRDTs). Statements are order-independent by default, but one can also write non-monotonic functions.

Have a look at the Bloom website and tutorials to learn more about Bloom.

Further reading

The CALM theorem, confluence analysis and Bloom

Joe Hellerstein's talk @RICON 2012 is a good introduction to the topic, as is Neil Conway's talk @Basho. For Bloom in particular, see Peter Alvaro's talk@Microsoft.

- The Declarative Imperative: Experiences and Conjectures in Distributed Logic - Hellerstein, 2010
- Consistency Analysis in Bloom: a CALM and Collected Approach - Alvaro et al., 2011
- Logic and Lattices for Distributed Programming - Conway et al., 2012
- Dedalus: Datalog in Time and Space - Alvaro et al., 2011

CRDTs

Marc Shapiro's talk @ Microsoft is a good starting point for understanding CRDT's.

- CRDTs: Consistency Without Concurrency Control - Letitia et al., 2009
- A comprehensive study of Convergent and Commutative Replicated Data Types, Shapiro et al., 2011
- An Optimized conflict-free Replicated Set - Bieniusa et al., 2012

Dynamo; PBS; optimistic replication

- Dynamo: Amazon’s Highly Available Key-value Store - DeCandia et al., 2007
- PNUTS: Yahoo!'s Hosted Data Serving Platform - Cooper et al., 2008
- The Bayou Architecture: Support for Data Sharing among Mobile Users - Demers et al. 1994
- Probabilistically Bound Staleness for Practical Partial Quorums - Bailis et al., 2012
- Eventual Consistency Today: Limitations, Extensions, and Beyond - Bailis & Ghodsi, 2013
- Optimistic replication - Saito & Shapiro, 2005

Previous Chapter | Home | Next Chapter

"Distributed systems: for fun and profit" by Mikito Takada.