Distributed systems

for fun and profit

Previous Chapter | Home | Next Chapter

2. Up and down the level of abstraction

In this chapter, we'll travel up and down the level of abstraction; look at some impossibility results (CAP and FLP) and then travel back down for the sake of performance.

If you've done any programming, the idea of levels of abstraction is probably familiar to you. You'll always work at some level of abstraction, interface with a lower level layer through some API, and probably provide some higher-level API or user interface to your users. The seven-layer OSI model of computer networking is a good example of this.

Distributed programming is, I'd assert, in large part dealing with consequences of distribution (duh!). That is, there is a tension between the reality that there are many nodes and with our desire for systems that "work like a single system". That means finding a good abstraction that balances what is possible with what is understandable and performant.

What do we mean when say X is more abstract than Y? First, that X does not introduce anything new or fundamentally different from Y. In fact, X may remove some aspects of Y or present them in a way that makes them more manageable. Second, that X is in some sense easier to grasp than Y, assuming that the things that X removed from Y are not important to the matter at hand.

As Nietzsche wrote:

> Every concept originates through our equating what is unequal. No leaf ever wholly equals another, and the concept "leaf" is formed through an arbitrary abstraction from these individual differences, through forgetting the distinctions; and now it gives rise to the idea that in nature there might be something besides the leaves which would be "leaf" - some kind of original form after which all leaves have been woven, marked, copied, colored, curled, and painted, but by unskilled hands, so that no copy turned out to be a correct, reliable, and faithful image of the original form.

Abstractions, fundamentally, are fake. Every situation is unique, as is every node. But abstractions make the world manageable: simpler problem statements - free of reality - are much more analytically tractable and provided that we did not ignore anything essential, the solutions are widely applicable.

Indeed, if the things that we kept around are essential, then the results we can derive will be widely applicable. This is why impossibility results are so important: they take the simplest possible formulation of a problem, and demonstrate that it is impossible to solve within some set of constraints or assumptions.

All abstractions ignore something in favor of equating things that are in reality unique. The trick is to get rid of everything that is not essential. How do you know what is essential? Well, you probably won't know a priori.

Every time we exclude some aspect of a system from our specification of the system, we risk introducing a source of error and/or a performance issue. That's

why sometimes we need to go in the other direction, and selectively introduce some aspects of real hardware and the real-world problem back. It may be sufficient to reintroduce some specific hardware characteristics (e.g. physical sequentiality) or other physical characteristics to get a system that performs well enough.

With this in mind, what is the least amount of reality we can keep around while still working with something that is still recognizable as a distributed system? A system model is a specification of the characteristics we consider important; having specified one, we can then take a look at some impossibility results and challenges.

A system model

A key property of distributed systems is distribution. More specifically, programs in a distributed system:

- run concurrently on independent nodes ...
- are connected by a network that may introduce nondeterminism and message loss ...
- and have no shared memory or shared clock.

There are many implications:

- each node executes a program concurrently
- knowledge is local: nodes have fast access only to their local state, and any information about global state is potentially out of date
- nodes can fail and recover from failure independently
- messages can be delayed or lost (independent of node failure; it is not easy to distinguish network failure and node failure)
- and clocks are not synchronized across nodes (local timestamps do not correspond to the global real time order, which cannot be easily observed)

A system model enumerates the many assumptions associated with a particular

system design.

System model a set of assumptions about the environment and facilities on which a distributed system is implemented

System models vary in their assumptions about the environment and facilities. These assumptions include:

- what capabilities the nodes have and how they may fail
- how communication links operate and how they may fail and
- properties of the overall system, such as assumptions about time and order

A robust system model is one that makes the weakest assumptions: any algorithm written for such a system is very tolerant of different environments, since it makes very few and very weak assumptions.

On the other hand, we can create a system model that is easy to reason about by making strong assumptions. For example, assuming that nodes do not fail means that our algorithm does not need to handle node failures. However, such a system model is unrealistic and hence hard to apply into practice.

Let's look at the properties of nodes, links and time and order in more detail.

Nodes in our system model

Nodes serve as hosts for computation and storage. They have:

- the ability to execute a program
- the ability to store data into volatile memory (which can be lost upon failure) and into stable state (which can be read after a failure)
- a clock (which may or may not be assumed to be accurate)

Nodes execute deterministic algorithms: the local computation, the local state after the computation, and the messages sent are determined uniquely by the message received and local state when the message was received.

There are many possible failure models which describe the ways in which nodes can fail. In practice, most systems assume a crash-recovery failure model: that is, nodes can only fail by crashing, and can (possibly) recover after crashing at some later point.

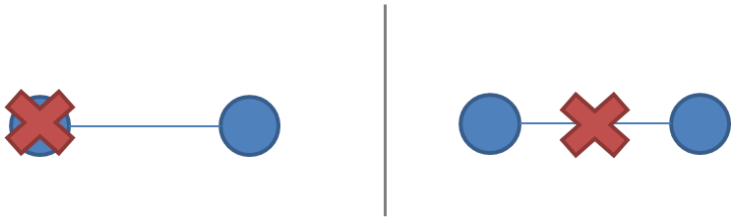
Another alternative is to assume that nodes can fail by misbehaving in any arbitrary way. This is known as Byzantine fault tolerance. Byzantine faults are rarely handled in real world commercial systems, because algorithms resilient to arbitrary faults are more expensive to run and more complex to implement. I will not discuss them here.

Communication links in our system model

Communication links connect individual nodes to each other, and allow messages to be sent in either direction. Many books that discuss distributed algorithms assume that there are individual links between each pair of nodes, that the links provide FIFO (first in, first out) order for messages, that they can only deliver messages that were sent, and that sent messages can be lost.

Some algorithms assume that the network is reliable: that messages are never lost and never delayed indefinitely. This may be a reasonable assumption for some real-world settings, but in general it is preferable to consider the network to be unreliable and subject to message loss and delays.

A network partition occurs when the network fails while the nodes themselves remain operational. When this occurs, messages may be lost or delayed until the network partition is repaired. Partitioned nodes may be accessible by some clients, and so must be treated differently from crashed nodes. The diagram below illustrates a node failure vs. a network partition:



It is rare to make further assumptions about communication links. We could assume that links only work in one direction, or we could introduce different communication costs (e.g. latency due to physical distance) for different links. However, these are rarely concerns in commercial environments except for long-distance links (WAN latency) and so I will not discuss them here; a more detailed model of costs and topology allows for better optimization at the cost of complexity.

Timing / ordering assumptions

One of the consequences of physical distribution is that each node experiences the world in a unique manner. This is inescapable, because information can only travel at the speed of light. If nodes are at different distances from each other, then any messages sent from one node to the others will arrive at a different time and potentially in a different order at the other nodes.

Timing assumptions are a convenient shorthand for capturing assumptions about the extent to which we take this reality into account. The two main alternatives are:

Synchronous system model	Processes execute in lock-step; there is a known upper bound on message transmission delay; each process has an accurate clock
Asynchronous system model	No timing assumptions - e.g. processes execute at independent rates; there is no bound on message transmission delay; useful clocks do not exist

The synchronous system model imposes many constraints on time and order. It essentially assumes that the nodes have the same experience: that messages that are sent are always received within a particular maximum transmission delay, and that processes execute in lock-step. This is convenient, because it allows you as the system designer to make assumptions about time and order, while the asynchronous system model doesn't.

Asynchronicity is a non-assumption: it just assumes that you can't rely on timing (or a "time sensor").

It is easier to solve problems in the synchronous system model, because assumptions about execution speeds, maximum message transmission delays and clock accuracy all help in solving problems since you can make inferences based on those assumptions and rule out inconvenient failure scenarios by assuming they never occur.

Of course, assuming the synchronous system model is not particularly realistic. Real-world networks are subject to failures and there are no hard bounds on message delay. Real world systems are at best partially synchronous: they may occasionally work correctly and provide some upper bounds, but there will be times where messages are delayed indefinitely and clocks are out of sync. I won't really discuss algorithms for synchronous systems here, but you will probably run into them in many other introductory books because they are analytically easier (but unrealistic).

The consensus problem

During the rest of this text, we'll vary the parameters of the system model. Next, we'll look at how varying two system properties:

- whether or not network partitions are included in the failure model, and
- synchronous vs. asynchronous timing assumptions

influence the system design choices by discussing two impossibility results (FLP and CAP).

Of course, in order to have a discussion, we also need to introduce a problem to solve. The problem I'm going to discuss is the consensus problem.

Several computers (or nodes) achieve consensus if they all agree on some value. More formally:

1. Agreement: Every correct process must agree on the same value.
2. Integrity: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
3. Termination: All processes eventually reach a decision.
4. Validity: If all correct processes propose the same value V, then all correct processes decide V.

The consensus problem is at the core of many commercial distributed systems. After all, we want the reliability and performance of a distributed system without having to deal with the consequences of distribution (e.g. disagreements / divergence between nodes), and solving the consensus problem makes it possible to solve several related, more advanced problems such as atomic broadcast and atomic commit.

Two impossibility results

The first impossibility result, known as the FLP impossibility result, is an impossibility result that is particularly relevant to people who design distributed algorithms. The second - the CAP theorem - is a related result that is more relevant to practitioners; people who need to choose between different system designs but who are not directly concerned with the design of algorithms.

The FLP impossibility result

I will only briefly summarize the FLP impossibility result, though it is considered

to be more important in academic circles. The FLP impossibility result (named after the authors, Fischer, Lynch and Patterson) examines the consensus problem under the asynchronous system model (technically, the agreement problem, which is a very weak form of the consensus problem). It is assumed that nodes can only fail by crashing; that the network is reliable, and that the typical timing assumptions of the asynchronous system model hold: e.g. there are no bounds on message delay.

Under these assumptions, the FLP result states that "there does not exist a (deterministic) algorithm for the consensus problem in an asynchronous system subject to failures, even if messages can never be lost, at most one process may fail, and it can only fail by crashing (stopping executing)".

This result means that there is no way to solve the consensus problem under a very minimal system model in a way that cannot be delayed forever. The argument is that if such an algorithm existed, then one could devise an execution of that algorithm in which it would remain undecided ("bivalent") for an arbitrary amount of time by delaying message delivery - which is allowed in the asynchronous system model. Thus, such an algorithm cannot exist.

This impossibility result is important because it highlights that assuming the asynchronous system model leads to a tradeoff: algorithms that solve the consensus problem must either give up safety or liveness when the guarantees regarding bounds on message delivery do not hold.

This insight is particularly relevant to people who design algorithms, because it imposes a hard constraint on the problems that we know are solvable in the asynchronous system model. The CAP theorem is a related theorem that is more relevant to practitioners: it makes slightly different assumptions (network failures rather than node failures), and has more clear implications for practitioners choosing between system designs.

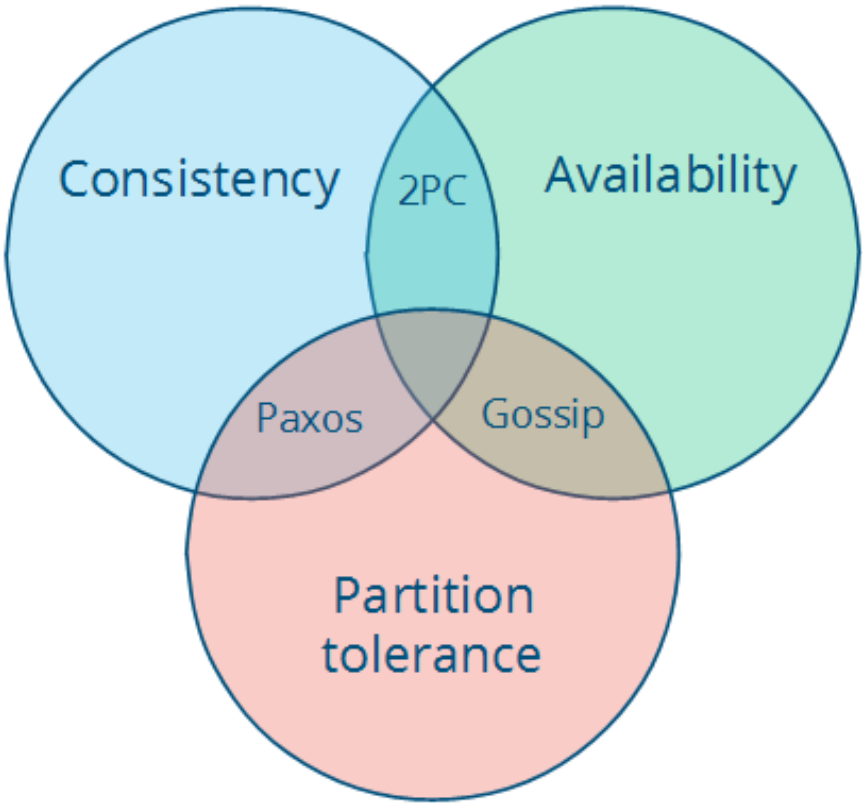
The CAP theorem

The CAP theorem was initially a conjecture made by computer scientist Eric Brewer. It's a popular and fairly useful way to think about tradeoffs in the guarantees that a system design makes. It even has a formal proof by Gilbert and Lynch and no, Nathan Marz didn't debunk it, in spite of what a particular discussion site thinks.

The theorem states that of these three properties:

- Consistency: all nodes see the same data at the same time.
- Availability: node failures do not prevent survivors from continuing to operate.
- Partition tolerance: the system continues to operate despite message loss due to network and/or node failure

only two can be satisfied simultaneously. We can even draw this as a pretty diagram, picking two properties out of three gives us three types of systems that correspond to different intersections:



Note that the theorem states that the middle piece (having all three properties) is not achievable. Then we get three different system types:

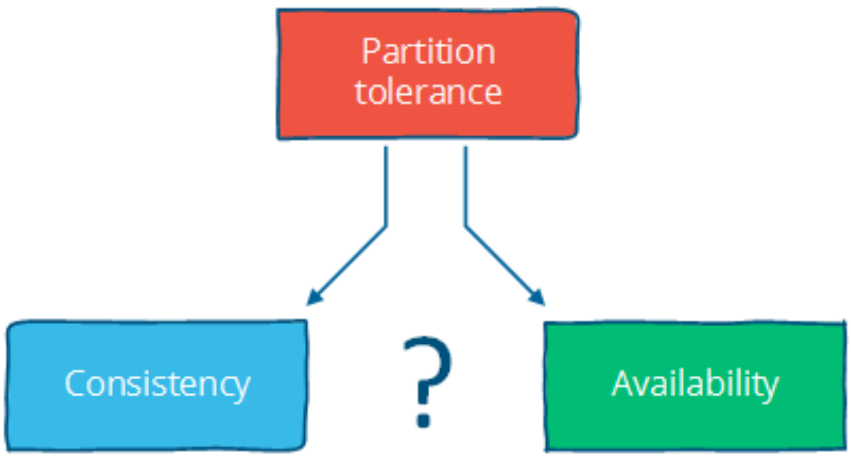
- CA (consistency + availability). Examples include full strict quorum protocols, such as two-phase commit.
- CP (consistency + partition tolerance). Examples include majority quorum protocols in which minority partitions are unavailable such as Paxos.
- AP (availability + partition tolerance). Examples include protocols using conflict resolution, such as Dynamo.

The CA and CP system designs both offer the same consistency model: strong consistency. The only difference is that a CA system cannot tolerate any node failures; a CP system can tolerate up to `f` faults given `2f+1` nodes in a non-Byzantine failure model (in other words, it can tolerate the failure of a minority `f` of the nodes as long as majority `f+1` stays up). The reason is simple:

- A CA system does not distinguish between node failures and network failures, and hence must stop accepting writes everywhere to avoid introducing divergence (multiple copies). It cannot tell whether a remote node is down, or whether just the network connection is down: so the only safe thing is to stop accepting writes.
- A CP system prevents divergence (e.g. maintains single-copy consistency) by forcing asymmetric behavior on the two sides of the partition. It only keeps the majority partition around, and requires the minority partition to become unavailable (e.g. stop accepting writes), which retains a degree of availability (the majority partition) and still ensures single-copy consistency.

I'll discuss this in more detail in the chapter on replication when I discuss Paxos. The important thing is that CP systems incorporate network partitions into their failure model and distinguish between a majority partition and a minority partition using an algorithm like Paxos, Raft or viewstamped replication. CA systems are not partition-aware, and are historically more common: they often use the two-phase commit algorithm and are common in traditional distributed relational databases.

Assuming that a partition occurs, the theorem reduces to a binary choice between availability and consistency.



I think there are four conclusions that should be drawn from the CAP theorem:

First, that many system designs used in early distributed relational database systems did not take into account partition tolerance (e.g. they were CA designs). Partition tolerance is an important property for modern systems, since network partitions become much more likely if the system is geographically distributed (as many large systems are).

Second, that there is a tension between strong consistency and high availability during network partitions. The CAP theorem is an illustration of the tradeoffs that occur between strong guarantees and distributed computation.

In some sense, it is quite crazy to promise that a distributed system consisting of independent nodes connected by an unpredictable network "behaves in a way that is indistinguishable from a non-distributed system".

Strong consistency guarantees require us to give up availability during a partition. This is because one cannot prevent divergence between two replicas that cannot communicate with each other while continuing to accept writes on both sides of the partition.

How can we work around this? By strengthening the assumptions (assume no partitions) or by weakening the guarantees. Consistency can be traded off against availability (and the related capabilities of offline accessibility and low latency). If "consistency" is defined as something less than "all nodes see the same data at the same time" then we can have both availability and some (weaker) consistency guarantee.

Third, that there is a tension between strong consistency and performance in normal operation.

Strong consistency / single-copy consistency requires that nodes communicate and agree on every operation. This results in high latency during normal operation.

If you can live with a consistency model other than the classic one; a consistency model that allows replicas to lag or to diverge, then you can reduce latency during normal operation and maintain availability in the presence of partitions.

When fewer messages and fewer nodes are involved, an operation can complete faster. But the only way to accomplish that is to relax the guarantees: let some of the nodes be contacted less frequently, which means that nodes can contain old data.

This also makes it possible for anomalies to occur. You are no longer guaranteed to get the most recent value. Depending on what kinds of guarantees are made, you might read a value that is older than expected, or even lose some updates.

Fourth - and somewhat indirectly - that if we do not want to give up availability during a network partition, then we need to explore whether consistency models other than strong consistency are workable for our purposes.

For example, even if user data is georeplicated to multiple datacenters, and the link between those two datacenters is temporarily out of order, in many cases we'll still want to allow the user to use the website / service. This means reconciling two divergent sets of data later on, which is both a technical challenge and a business risk. But often both the technical challenge and the business risk are manageable, and so it is preferable to provide high availability.

Consistency and availability are not really binary choices, unless you limit yourself to strong consistency. But strong consistency is just one consistency model: the one where you, by necessity, need to give up availability in order to prevent more than a single copy of the data from being active. As Brewer himself points out, the "2 out of 3" interpretation is misleading.

If you take away just one idea from this discussion, let it be this: "consistency" is not a singular, unambiguous property. Remember:

ACID consistency !=
CAP consistency !=

Instead, a consistency model is a guarantee - any guarantee - that a data store gives to programs that use it.

Consistency model a contract between programmer and system, wherein the system guarantees that if the programmer follows some specific rules, the results of operations on the data store will be predictable

The "C" in CAP is "strong consistency", but "consistency" is not a synonym for "strong consistency".

Let's take a look at some alternative consistency models.

Strong consistency vs. other consistency models

Consistency models can be categorized into two types: strong and weak consistency models:

- Strong consistency models (capable of maintaining a single copy)
    - Linearizable consistency
    - Sequential consistency
- Weak consistency models (not strong)
    - Client-centric consistency models
    - Causal consistency: strongest model available
    - Eventual consistency models

Strong consistency models guarantee that the apparent order and visibility of updates is equivalent to a non-replicated system. Weak consistency models, on the other hand, do not make such guarantees.

Note that this is by no means an exhaustive list. Again, consistency models are just arbitrary contracts between the programmer and system, so they can be almost anything.

Strong consistency models

Strong consistency models can further be divided into two similar, but slightly different consistency models:

- Linearizable consistency: Under linearizable consistency, all operations appear to have executed atomically in an order that is consistent with the global real-time ordering of operations. (Herlihy & Wing, 1991)
- Sequential consistency: Under sequential consistency, all operations appear to have executed atomically in some order that is consistent with the order seen at individual nodes and that is equal at all nodes. (Lamport, 1979)

The key difference is that linearizable consistency requires that the order in which operations take effect is equal to the actual real-time ordering of operations. Sequential consistency allows for operations to be reordered as long as the order observed on each node remains consistent. The only way someone can distinguish between the two is if they can observe all the inputs and timings going into the system; from the perspective of a client interacting with a node, the two are equivalent.

The difference seems immaterial, but it is worth noting that sequential consistency does not compose.

Strong consistency models allow you as a programmer to replace a single server with a cluster of distributed nodes and not run into any problems.

All the other consistency models have anomalies (compared to a system that guarantees strong consistency), because they behave in a way that is distinguishable from a non-replicated system. But often these anomalies are acceptable, either because we don't care about occasional issues or because we've written code that deals with inconsistencies after they have occurred in some way.

Note that there really aren't any universal typologies for weak consistency models, because "not a strong consistency model" (e.g. "is distinguishable from a non-replicated system in some way") can be almost anything.

Client-centric consistency models

Client-centric consistency models are consistency models that involve the notion of a client or session in some way. For example, a client-centric consistency model might guarantee that a client will never see older versions of a data item. This is often implemented by building additional caching into the client library, so that if a client moves to a replica node that contains old data, then the client library returns its cached value rather than the old value from the replica.

Clients may still see older versions of the data, if the replica node they are on does not contain the latest version, but they will never see anomalies where an older version of a value resurfaces (e.g. because they connected to a different replica). Note that there are many kinds of consistency models that are client-centric.

Eventual consistency

The eventual consistency model says that if you stop changing values, then after some undefined amount of time all replicas will agree on the same value. It is implied that before that time results between replicas are inconsistent in some undefined manner. Since it is trivially satisfiable (liveness property only), it is useless without supplemental information.

Saying something is merely eventually consistent is like saying "people are eventually dead". It's a very weak constraint, and we'd probably want to have at least some more specific characterization of two things:

First, how long is "eventually"? It would be useful to have a strict lower bound, or at least some idea of how long it typically takes for the system to converge to the same value.

Second, how do the replicas agree on a value? A system that always returns "42" is eventually consistent: all replicas agree on the same value. It just doesn't converge to a useful value since it just keeps returning the same fixed value. Instead, we'd like to have a better idea of the method. For example, one way to decide is to have the value with the largest timestamp always win.

So when vendors say "eventual consistency", what they mean is some more precise term, such as "eventually last-writer-wins, and read-the-latest-observed-value in the meantime" consistency. The "how?" matters, because a bad method can lead to writes being lost - for example, if the clock on one node is set incorrectly and timestamps are used.

I will look into these two questions in more detail in the chapter on replication methods for weak consistency models.

Further reading

- Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services - Gilbert & Lynch, 2002
- Impossibility of distributed consensus with one faulty process - Fischer, Lynch and Patterson, 1985
- Perspectives on the CAP Theorem - Gilbert & Lynch, 2012
- CAP Twelve Years Later: How the "Rules" Have Changed - Brewer, 2012
- Uniform consensus is harder than consensus - Charron-Bost & Schiper, 2000
- Replicated Data Consistency Explained Through Baseball - Terry, 2011
- Life Beyond Distributed Transactions: an Apostate's Opinion - Helland, 2007

- If you have too much data, then 'good enough' is good enough - Helland, 2011
- Building on Quicksand - Helland & Campbell, 2009

Previous Chapter | Home | Next Chapter

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