Distributed systems

for fun and profit

Previous Chapter | Home | Next Chapter

1. Distributed systems at a high level

Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers.

There are two basic tasks that any computer system needs to accomplish:

- storage and
- computation

Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers - usually, because the problem no longer fits on a single computer.

Nothing really demands that you use distributed systems. Given infinite money and infinite R&D time, we wouldn't need distributed systems. All computation and storage could be done on a magic box - a single, incredibly fast and incredibly reliable system that you pay someone else to design for you.

However, few people have infinite resources. Hence, they have to find the right place on some real-world cost-benefit curve. At a small scale, upgrading hardware is a viable strategy. However, as problem sizes increase you will reach a point where either the hardware upgrade that allows you to solve the problem on a single node does not exist, or becomes cost-prohibitive. At that point, I welcome you to the world of distributed systems.

It is a current reality that the best value is in mid-range, commodity hardware - as long as the maintenance costs can be kept down through fault-tolerant software.

Computations primarily benefit from high-end hardware to the extent to which they can replace slow network accesses with internal memory accesses. The performance advantage of high-end hardware is limited in tasks that require large amounts of communication between nodes.

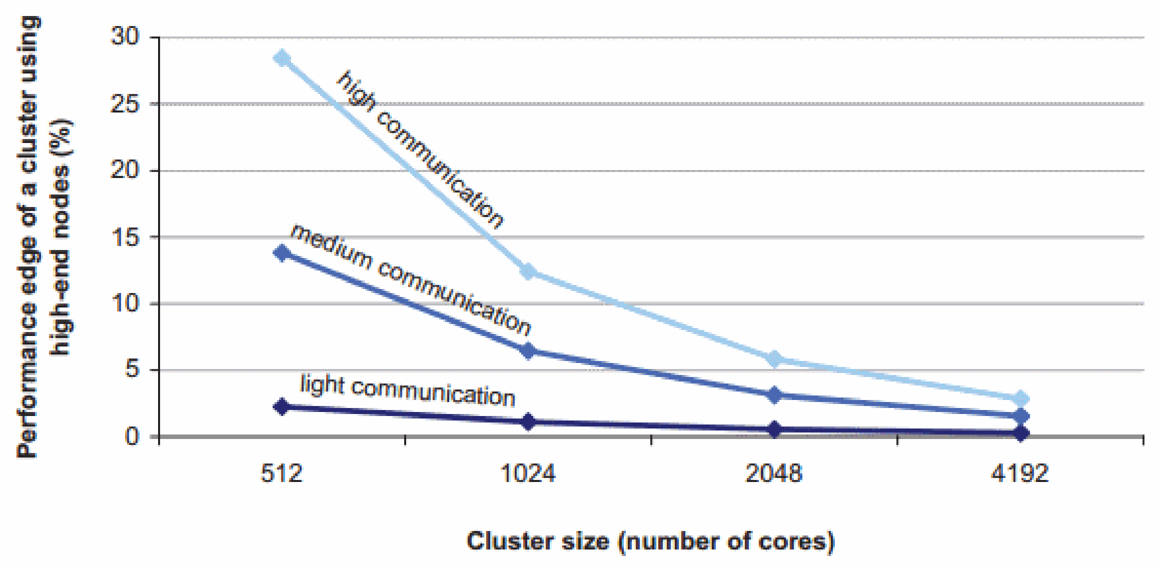


FIGURE 3.2: Performance advantage of a cluster built with high-end server nodes (128-core SMP) over a cluster with the same number of processor cores built with low-end server nodes (four-core SMP), for clusters of varying size.

As the figure above from Barroso, Clidaras & Hölzle shows, the performance gap between high-end and commodity hardware decreases with cluster size assuming a uniform memory access pattern across all nodes.

Ideally, adding a new machine would increase the performance and capacity of the system linearly. But of course this is not possible, because there is some overhead that arises due to having separate computers. Data needs to be copied around, computation tasks have to be coordinated and so on. This is why it's worthwhile to study distributed algorithms - they provide efficient solutions to specific problems, as well as guidance about what is possible, what the minimum cost of a correct implementation is, and what is impossible.

The focus of this text is on distributed programming and systems in a mundane, but commercially relevant setting: the data center. For example, I will not discuss specialized problems that arise from having an exotic network configuration, or that arise in a shared-memory setting. Additionally, the focus is on exploring the system design space rather than on optimizing any specific design - the latter is a topic for a much more specialized text.

What we want to achieve: Scalability and other good things

The way I see it, everything starts with the need to deal with size.

Most things are trivial at a small scale - and the same problem becomes much harder once you surpass a certain size, volume or other physically constrained thing. It's easy to lift a piece of chocolate, it's hard to lift a mountain. It's easy to count how many people are in a room, and hard to count how many people are in a country.

So everything starts with size - scalability. Informally speaking, in a scalable system as we move from small to large, things should not get incrementally worse. Here's another definition:

Scalability is the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

What is it that is growing? Well, you can measure growth in almost any terms (number of people, electricity usage etc.). But there are three particularly interesting things to look at:

- Size scalability: adding more nodes should make the system linearly faster; growing the dataset should not increase latency
- Geographic scalability: it should be possible to use multiple data centers to reduce the time it takes to respond to user queries, while dealing with cross-data center latency in some sensible manner.
- Administrative scalability: adding more nodes should not increase the administrative costs of the system (e.g. the administrators-to-machines ratio).

Of course, in a real system growth occurs on multiple different axes simultaneously; each metric captures just some aspect of growth.

A scalable system is one that continues to meet the needs of its users as scale increases. There are two particularly relevant aspects - performance and availability - which can be measured in various ways.

Performance (and latency)

Performance	is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.

Depending on the context, this may involve achieving one or more of the following:

- Short response time/low latency for a given piece of work
- High throughput (rate of processing work)
- Low utilization of computing resource(s)

There are tradeoffs involved in optimizing for any of these outcomes. For example, a system may achieve a higher throughput by processing larger batches of work thereby reducing operation overhead. The tradeoff would be longer response times for individual pieces of work due to batching.

I find that low latency - achieving a short response time - is the most interesting aspect of performance, because it has a strong connection with physical (rather than financial) limitations. It is harder to address latency using financial resources than the other aspects of performance.

There are a lot of really specific definitions for latency, but I really like the idea that the etymology of the word evokes:

Latency	The state of being latent; delay, a period between the initiation of something and the occurrence.

And what does it mean to be "latent"?

Latent	From Latin latens, latentis, present participle of lateo ("lie hidden"). Existing or present but concealed or inactive.

This definition is pretty cool, because it highlights how latency is really the time between something happened and the time it has an impact or becomes visible.

For example, imagine that you are infected with an airborne virus that turns people into zombies. The latent period is the time between when you became infected, and when you turn into a zombie. That's latency: the time during which something that has already happened is concealed from view.

Let's assume for a moment that our distributed system does just one high-level task:

given a query, it takes all of the data in the system and calculates a single result. In other words, think of a distributed system as a data store with the ability to run a single deterministic computation (function) over its current content:

```
result = query(all data in the system)
```

Then, what matters for latency is not the amount of old data, but rather the speed at which new data "takes effect" in the system. For example, latency could be measured in terms of how long it takes for a write to become visible to readers.

The other key point based on this definition is that if nothing happens, there is no "latent period". A system in which data doesn't change doesn't (or shouldn't) have a latency problem.

In a distributed system, there is a minimum latency that cannot be overcome: the speed of light limits how fast information can travel, and hardware components have a minimum latency cost incurred per operation (think RAM and hard drives but also CPUs).

How much that minimum latency impacts your queries depends on the nature of those queries and the physical distance the information needs to travel.

Availability (and fault tolerance)

The second aspect of a scalable system is availability.

Availability	the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable.

Distributed systems allow us to achieve desirable characteristics that would be hard to accomplish on a single system. For example, a single machine cannot tolerate any failures since it either fails or doesn't.

Distributed systems can take a bunch of unreliable components, and build a reliable system on top of them.

Systems that have no redundancy can only be as available as their underlying components. Systems built with redundancy can be tolerant of partial failures and thus be more available. It is worth noting that "redundant" can mean different things depending on what you look at - components, servers, datacenters and so on.

Formulaically, availability is: `Availability = uptime / (uptime + downtime)`.

Availability from a technical perspective is mostly about being fault tolerant. Because the probability of a failure occurring increases with the number of components, the system should be able to compensate so as to not become less reliable as the number of components increases.

For example:

Availability %	How much downtime is allowed per year?
90% ("one nine")	More than a month
99% ("two nines")	Less than 4 days
99.9% ("three nines")	Less than 9 hours
99.99% ("four nines")	Less than an hour
99.999% ("five nines")	~ 5 minutes
99.9999% ("six nines")	~ 31 seconds

Availability is in some sense a much wider concept than uptime, since the availability of a service can also be affected by, say, a network outage or the company owning the service going out of business (which would be a factor which is not really relevant to fault tolerance but would still influence the availability of the system). But without knowing every single specific aspect of the system, the best we can do is design for fault tolerance.

What does it mean to be fault tolerant?

Fault tolerance	ability of a system to behave in a well-defined manner once faults occur

Fault tolerance boils down to this: define what faults you expect and then design a system or an algorithm that is tolerant of them. You can't tolerate faults you haven't considered.

What prevents us from achieving good things?

Distributed systems are constrained by two physical factors:

- the number of nodes (which increases with the required storage and computation capacity)
- the distance between nodes (information travels, at best, at the speed of light)

Working within those constraints:

- an increase in the number of independent nodes increases the probability of failure in a system (reducing availability and increasing administrative costs)
- an increase in the number of independent nodes may increase the need for communication between nodes (reducing performance as scale increases)
- an increase in geographic distance increases the minimum latency for communication between distant nodes (reducing performance for certain operations)

Beyond these tendencies - which are a result of the physical constraints - is the world of system design options.

Both performance and availability are defined by the external guarantees the system makes. On a high level, you can think of the guarantees as the SLA (service level agreement) for the system: if I write data, how quickly can I access it elsewhere? After the data is written, what guarantees do I have of durability? If I ask the system to run a computation, how quickly will it return results? When components fail, or are taken out of operation, what impact will this have on the system?

There is another criterion, which is not explicitly mentioned but implied: intelligibility. How understandable are the guarantees that are made? Of course, there are no simple metrics for what is intelligible.

I was kind of tempted to put "intelligibility" under physical limitations. After all, it is a hardware limitation in people that we have a hard time understanding anything that involves more moving things than we have fingers. That's the difference between an error and an anomaly - an error is incorrect behavior, while an anomaly is unexpected behavior. If you were smarter, you'd expect the anomalies to occur.

Abstractions and models

This is where abstractions and models come into play. Abstractions make things more manageable by removing real-world aspects that are not relevant to solving a problem. Models describe the key properties of a distributed system in a precise manner. I'll discuss many kinds of models in the next chapter, such as:

- System model (asynchronous / synchronous)
- Failure model (crash-fail, partitions, Byzantine)
- Consistency model (strong, eventual)

A good abstraction makes working with a system easier to understand, while capturing the factors that are relevant for a particular purpose.

There is a tension between the reality that there are many nodes and with our desire for systems that "work like a single system". Often, the most familiar model (for example, implementing a shared memory abstraction on a distributed system) is too expensive.

A system that makes weaker guarantees has more freedom of action, and hence potentially greater performance - but it is also potentially hard to reason about. People are better at reasoning about systems that work like a single system, rather than a collection of nodes.

One can often gain performance by exposing more details about the internals of the system. For example, in columnar storage, the user can (to some extent) reason about the locality of the key-value pairs within the system and hence make decisions that influence the performance of typical queries. Systems which hide these kinds of details are easier to understand (since they act more like single unit, with fewer details to think about), while systems that expose more real-world details may be more performant (because they correspond more closely to reality).

Several types of failures make writing distributed systems that act like a single system difficult. Network latency and network partitions (e.g. total network failure between some nodes) mean that a system needs to sometimes make hard choices about whether it is better to stay available but lose some crucial guarantees that cannot be enforced, or to play it safe and refuse clients when these types of failures occur.

The CAP theorem - which I will discuss in the next chapter - captures some of these tensions. In the end, the ideal system meets both programmer needs (clean semantics) and business needs (availability/consistency/latency).

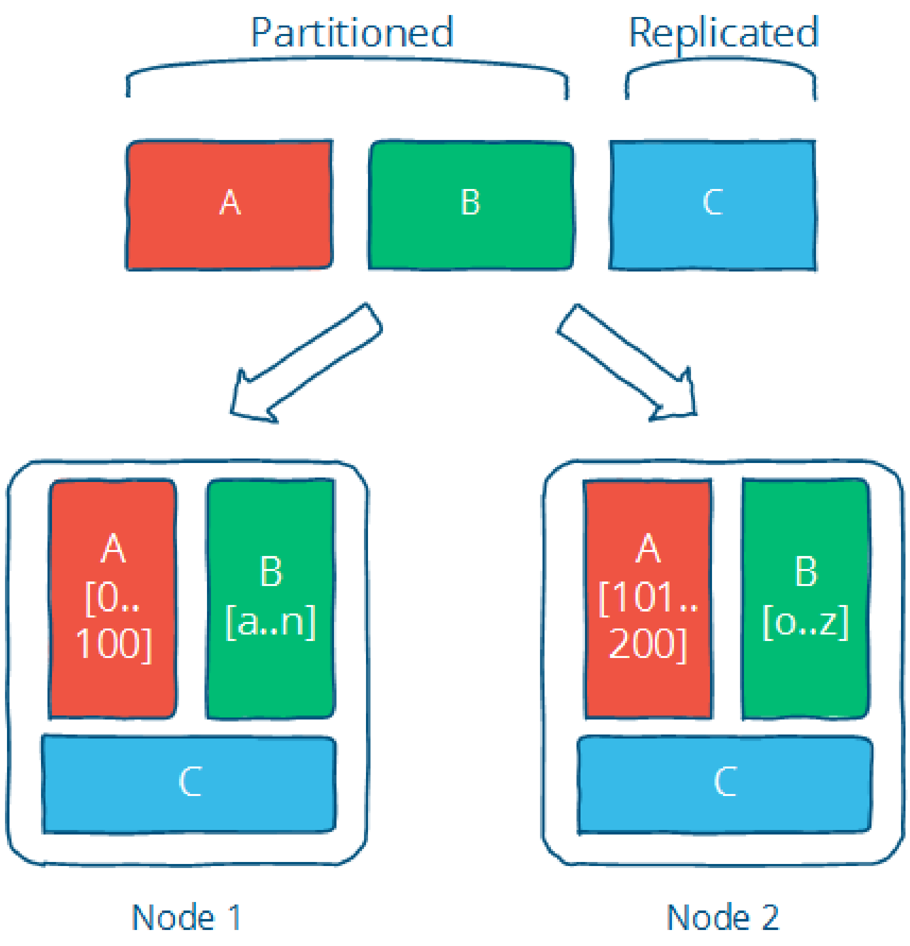
Design techniques: partition and replicate

The manner in which a data set is distributed between multiple nodes is very important. In order for any computation to happen, we need to locate the data and then act on it.

There are two basic techniques that can be applied to a data set. It can be split over multiple nodes (partitioning) to allow for more parallel processing. It can also be copied or cached on different nodes to reduce the distance between the client and the server and for greater fault tolerance (replication).

Divide and conquer - I mean, partition and replicate.

The picture below illustrates the difference between these two: partitioned data (A and B below) is divided into independent sets, while replicated data (C below) is copied to multiple locations.



This is the one-two punch for solving any problem where distributed computing plays a role. Of course, the trick is in picking the right technique for your concrete implementation; there are many algorithms that implement replication and partitioning, each with different limitations and advantages which need to be assessed against your design objectives.

Partitioning

Partitioning is dividing the dataset into smaller distinct independent sets; this is used to reduce the impact of dataset growth since each partition is a subset of the data.

- Partitioning improves performance by limiting the amount of data to be examined and by locating related data in the same partition
- Partitioning improves availability by allowing partitions to fail independently, increasing the number of nodes that need to fail before availability is sacrificed

Partitioning is also very much application-specific, so it is hard to say much about it without knowing the specifics. That's why the focus is on replication in most texts, including this one.

Partitioning is mostly about defining your partitions based on what you think the primary access pattern will be, and dealing with the limitations that come from having independent partitions (e.g. inefficient access across partitions, different rate of growth etc.).

Replication

Replication is making copies of the same data on multiple machines; this allows more servers to take part in the computation.

Let me inaccurately quote Homer J. Simpson:

> To replication! The cause of, and solution to all of life's problems.

Replication - copying or reproducing something - is the primary way in which we can fight latency.

- Replication improves performance by making additional computing power and bandwidth applicable to a new copy of the data
- Replication improves availability by creating additional copies of the data, increasing the number of nodes that need to fail before availability is sacrificed

Replication is about providing extra bandwidth, and caching where it counts. It is also about maintaining consistency in some way according to some consistency model.

Replication allows us to achieve scalability, performance and fault tolerance. Afraid of loss of availability or reduced performance? Replicate the data to avoid a bottleneck or single point of failure. Slow computation? Replicate the computation on multiple systems. Slow I/O? Replicate the data to a local cache to reduce latency or onto multiple machines to increase throughput.

Replication is also the source of many of the problems, since there are now independent copies of the data that has to be kept in sync on multiple machines - this means ensuring that the replication follows a consistency model.

The choice of a consistency model is crucial: a good consistency model provides clean semantics for programmers (in other words, the properties it guarantees are easy to reason about) and meets business/design goals such as high availability or strong consistency.

Only one consistency model for replication - strong consistency - allows you to program as-if the underlying data was not replicated. Other consistency models expose some internals of the replication to the programmer. However, weaker consistency models can provide lower latency and higher availability - and are not necessarily harder to understand, just different.

Further reading

- The Datacenter as a Computer - An Introduction to the Design of Warehouse-Scale Machines - Barroso & Hölzle, 2008
- Fallacies of Distributed Computing
- Notes on Distributed Systems for Young Bloods - Hodges, 2013

Previous Chapter | Home | Next Chapter

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

cv · 24 days ago
You write pretty good books man!. Very concise and detailed at the same time. May be you should think about publishing more of these.
· Reply · Share ›

asloob · 4 months ago
Very good introduction. Presents all the relevant information in a non-textbook (read interesting) way.
· Reply · Share ›