# Scalable Web Architecture and Distributed Systems

Open Source software has become a fundamental building block for some of the biggest web sites online. And as those websites have grown, best practices and guiding principles around these architectures have emerged. This chapter seeks to cover some of the key things to consider when designing these types of systems, as well as some of the building blocks used to achieve these goals.

In the interest of brevity, this chapter is largely focused on web systems, although some of the material is applicable in other areas as well.

## Principles of Web Distributed Systems Design

What exactly does it mean to build and operate a scalable web site or application? The primitive level is just connecting users with remote resources via the Internet – and the part that makes it scalable is that the resources, or access to those resources, are distributed across multiple servers.

Like most things in life, taking the time to plan ahead can help in the long run; and since most systems do not start out large, understanding some of the considerations behind the big online sites can result in smarter decisions at the beginning. Below are some of the key principles that influence the design of large-scale web systems:

* **Availability** 
  1. The uptime of a website is absolutely critical to the reputation and functioning of many companies. For some of the larger online retail sites, being unavailable for even minutes can result in thousands to millions of dollars in lost revenue; so designing their systems to be constantly available and resilient to failure is both a fundamental business and technology requirement. High availability in distributed systems requires the careful consideration of redundancy for key components, rapid recovery in the event of partial system failures, and graceful degradation when problems occur.
* **Performance**

Website performance has become an important consideration for most online sites. The speed of a website affects usage and user satisfaction, as well as search engine rankings, a factor that directly correlates to lost revenue and retention. As a result creating a system that is optimized for fast responses and low latency is key to these large web systems.

* **Reliability**
  1. A system needs to be reliable, such that a request for data will consistently return the same data. In the event the data changes or is updated, then that same request should return the new data. Users need to know that if something is written to the system, or stored, it will persist and can be relied on for future retrieval.
* **Scalability**
  1. When it comes to any large distributed system, its size is just one thing to be considered. Just as important is the effort required to increase capacity to handle greater amounts of load, which is commonly referred to the scalability of the system. Scalability can refer to lots of different parameters of the system – how much additional traffic can it handle, how easy is it to add more storage capacity, or even how many more transactions can be processed.
* **Manageability** 
  1. Similar to scalability, designing a system that is easy to operate is another important consideration – in other words, the scalability of operations is the manageability of the system. Things to consider for manageability are the ease of diagnosis and understanding problems when they occur, making updates or modifications, and how simple the system is to operate (i.e. does it routinely operate without failure or exceptions).
* **Cost** 
  1. And with any system an important factor is the cost. This can include hardware and software costs, but it is also important to consider the amount of developer time the system takes to build, the amount of operational effort required to run the system, and even the amount of training required. It is the total cost of ownership.

Each of these principles provides the basis for decisions of tradeoff (compromise) in a distributed web architecture. However, they also can be at odds with one another, such that achieving one objective comes at the cost of another. A really basic example: choosing to address capacity by simply adding more servers (scalability) can come at the price of manageability (you have to operate an additional server) and cost (the price of the servers).

When thinking about designing any sort of web application it is important to consider these key principles, even if it is to acknowledge that a design may sacrifice one or more of them.

## The Basics

When it comes to system architecture there are a few things to consider: what are the right pieces, how these pieces fit together, and what are the right tradeoffs. When it comes to architecture, investing in scaling before it is needed is generally not a smart business proposition, however, some forethought in the design can save substantial time and resources in the future.

This section is focused on some of the core values that are central to almost all large web applications – **services**, **redundancy**, **partitions**, and **handling failure**. In order to explain these in detail, though, it is best to start with an example.

*Example*: Image Hosting Application

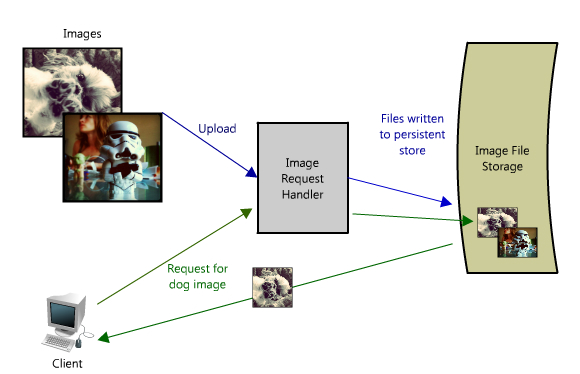
At some point you have probably posted an image online, and for big sites that host and deliver lots of images there are certainly challenges in building an architecture that is cost effective, highly available, and with low latency (fast).

For the sake of simplicity, let’s assume that this application has 2 key parts – the ability to upload (write) an image to the server, and the ability to query for an image. While we certainly want the upload to be efficient, we care the most about having very fast delivery when someone requests an image (for example, these images could be requested for a web page, or other application). This is very similar functionality to what a web server, or CDN edge server might provide.

Other important aspects to the system are:

* There is no limit to the number of images that need to be stored, so storage scalability, in terms of image count, needs to be considered
* There needs to be low latency for image downloads/requests
* If a user uploads an image, the image should always be there (data reliability for images)
* The system should be easy to maintain (manageability)
* Since image hosting doesn’t have high profit margins, the system needs to be cost effective

Here is a simplified diagram of the functionality:



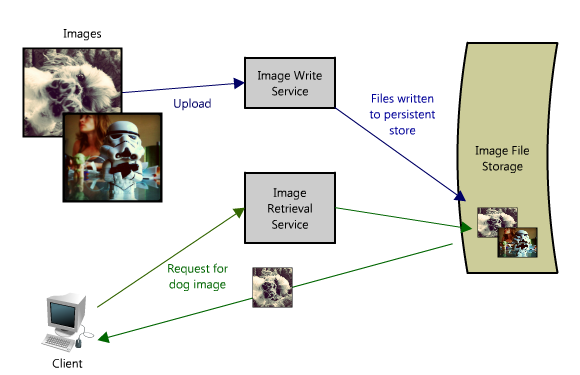
In this image hosting example, the system must be perceivably fast, its data stored reliably and all of these attributes highly scalable. Building a small version of this application would be trivial, easily hosted on a single server, however, that would not be interesting for this chapter. Let’s assume that we want to build something that could (eventually) grow as big as a Flickr, or another photo sharing site.

Services

When considering scalable system design, it helps to decouple functionality and think about each part of the system as its own service with a clearly defined interface. In practice, systems designed in this way are said to have a Service Oriented Architecture (SOA). For these types of systems, each service has its own distinct functional context and interaction with anything outside of that context takes place through an abstract interface, typically the public facing API of another service.

Deconstructing a system into a set of complementary services decouples the operation of those pieces from one another. This *abstraction* helps establish a clear relationship between the service, its underlying environment, and the consumers of that service. By creating these clear delineations, it can help isolate problems (making operations and manageability easier), but also allows each piece to scale independently of one another (this includes the development time around making changes to different pieces). This sort of service-oriented design for systems is very similar to object-oriented design for programming.

In our example, all requests to upload (write) and retrieve images are being processed by the identical server. Fast forward and assume that there is a ton of reads and writes coming in – and because writes/uploads tend to be longer (the connection has to be maintained over a longer period of time as the file is uploaded), it is easy to imagine that the time required to retrieve the image could be drastically impacted by a large amount of write activity. Planning for this sort of bottleneck in the future makes a good case to split out the reads and writes of images into their own services. This allows us to scale each of them independently (since it is likely we will always do more reading than writing), but also helps clarify what is going on at each point. Finally, this separates future concerns, which would make it easier to troubleshoot and scale a problem like slow reads.



The advantage of this approach is that we are able to solve each problem independently of one another - we don’t have to worry about writing new images, and retrieving them in the same context. Each of these services still leverage the global corpus of images, but both are free to create their own intermediate results optimal for achieving performance (for example, queuing up requests, or caching popular images – more on this below). And from a maintenance and cost perspective each service can scale independently as needed (which is great because if they were combined and intermingled one could inadvertently impact the performance of the other, as in the scenario discussed above).

Redundancy

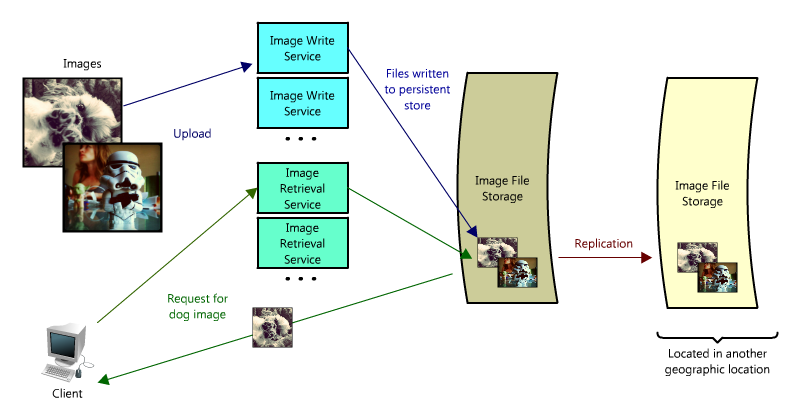
In order to handle failure gracefully a web architecture must have redundancy of its services and data. For example, if there is only one copy of a file stored on a single server then losing that server also means losing that file. Losing data is seldom a good thing, and a common way of handling it is to create multiple, or redundant, copies.

This same principle also applies to services. If there is a core piece of functionality for an application, ensuring that multiple copies or versions are running simultaneously can secure against the failure of a single node.

Creating redundancy in a system can remove single points of failure, and provide a backup or spare functionality if needed in a crisis. For example, if there are two instances of the same service running in production, and one fails or degrades, the system can *failover* to the healthy copy. When it comes to failover, this can happen automatically or require manual intervention.

Another key part of service redundancy, is creating a *Shared Nothing Architecture*. With this architecture, each node is able to operate independently of one another and there is no central “brain” managing state or coordinating activities for the other nodes. This helps a lot with scalability since new nodes can be added without special conditions or knowledge. However, and most importantly, there is no single point of contention in these systems, so they are much more resilient to failure.

So for example in our image server application, all images would have redundant copies on another piece of hardware somewhere (ideally in a different geographic location in the event of a catastrophe like an earthquake or fire in the data center), and the services to access the images would be redundant copies, all potentially servicing requests (load balancers are a great way to make this possible, but there is more on that below).



Partitions

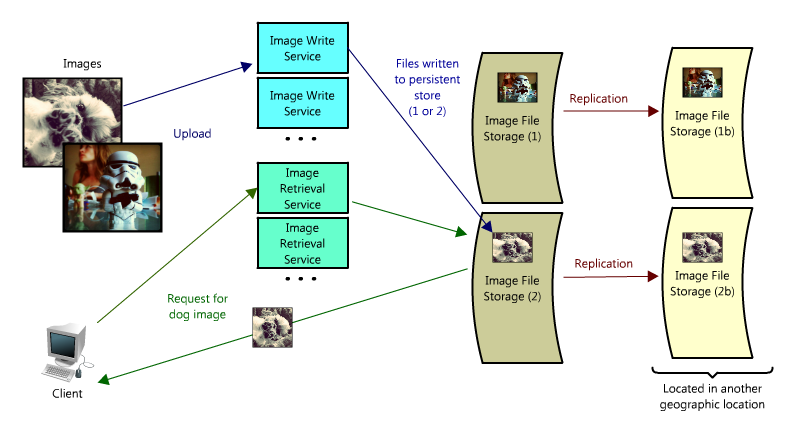
There may be really big data sets that are unable to fit on a single server. It may also be the case that an operation requires too many computing resources, diminishing performance, and therefore making it necessary to add capacity. In either case you have two choices: scale vertically or horizontally.

Scaling vertically means adding more resources to an individual server. So for a very large data set, this might mean adding more (or bigger) hard drives so a single server can contain its entirety. In the case of the compute operation, this could mean moving the computation to a bigger server with a faster CPU or more memory. In each case vertical scaling is accomplished by making the individual resource better and able to handle more on its own.

To scale horizontally, on the other hand, is to add more nodes. In the case of the data, this might be a second server to store the additional parts of the data set; and for the computing resource it would mean splitting the operation or load across some additional nodes. To take full advantage of horizontal scaling, the system should include it as an intrinsic design principle of the system architecture, otherwise it can be quite cumbersome to modify and separate out the context to make this possible.

And when it comes to horizontally scaling, one of the more common techniques is to break-up your services into partitions (some people may also refer to these pieces as shards). The partitions can be federated such that each logical set of functionality is separate – this could be done by geographic boundaries, or another criteria like free versus paying users. The advantage of these schemes is to provide a service or data store with added capacity.

In our image server example, it is possible that the single file server used to store images could be replaced by multiple file servers, each containing its own unique set of images. Such an architecture would allow the system to fill each file server with images, adding additional servers as the disks become full. This sort of design would require a naming scheme that tied an image’s filename with the server containing it. An image’s name could be formed from a consistent hashing scheme mapped across the servers. Or alternatively, each image could be assigned an incremental ID, so that when a client makes a request for an image, the image retrieval service only needs to maintain the range of IDs that are mapped to each of the servers (like an index).



Of course there are challenges distributing data or functionality across multiple servers. One of the key issues is *data locality*; in distributed systems any time the data is closer to the operation or point of computation, the better performance of the system. Therefore it is potentially problematic to have the data spread across multiple servers, as any time it is needed it may not be local forcing the servers to perform a costly fetch of the required information across the network.

Another potential issue comes in the form of *inconsistency*. When there are different services reading and writing from a shared resource (potentially another service or data store) there is the chance for race conditions, where some data was supposed to be updated, but the read happens prior to the update – and in those cases the data is inconsistent. For example, in the image hosting scenario, a race condition could occur if one client sent a request to update the dog image with a new title (changing it from “Dog” to “Gizmo”), but at the same time another client was reading the image. In that circumstance it is unclear which title (“Dog” or “Gizmo”) would be the one received by the 2nd client.

There are certainly some obstacles associated with partitioning data, but partitioning allows each problem to be split (by data, load, usage patterns, etc.) into manageable chunks. This can help with scalability and manageability, but is not without risk.

Handling Failures

When it comes to failures, most fall into one of two buckets: hardware or software related.

* Hardware failures used to be more common, but with all of the recent innovations in hardware design and manufacturing they tend to be fewer and far between with most of these physical failures tending to be network or drive related.
* Software failures, on the other hand, come in many more varieties. And software bugs in distributed systems can be difficult to replicate and, consequently, fix.

In small, self-contained systems it is much easier to simulate the conditions required to replicate and debug issues, with most of these issues classified as being a Bohrbug, that is a bug “that manifests itself consistently under a well-defined (but possibly unknown) set of conditions” [1]. However, in more complex systems or production environments having many servers, it can be extremely difficult to find and diagnose more unusual bugs; like the Heisenbug “that disappears or alters its characteristics when an attempt is made to study it” [1].

With more hardware the probability goes up that there will be a failure somewhere. Add more software and the complex interactions between different programs creates greater chance for more bugs, including the unusual ones. As a result, any distributed design will carefully consider failure and diagnostic scenarios.

When designing distributed systems it is said that the following (perhaps normal) assumptions should be considered false (and these are so well known that they commonly referred to as the Fallacies of Distributed Computing):

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous. [2]

There are lots of different things to consider when it comes to guarding against failure, redundancy was covered above, but two other important techniques are fault tolerance and monitoring.

Fault Tolerance:

Another important part of service based architectures is to set up each service to be fault tolerant, such that in the event one of its dependencies are unavailable or return an error, it is able to handle those cases and degrade gracefully. There are many methods for achieving fault tolerance in a distributed system, for example: redundancy (as described above), standbys, feature flags, and asynchrony.

Standbys – a standby is exactly that, a redundant set of functionality or data waiting on standby that may be swapped to replace another failing instance. Replication can be utilized to maintain real time copies of the master database so that data may be replaced without loss or disruption.

Feature flags – a feature flag is used to enable or disable functionality in a production system. In the event of a failure for a particular system, features that depend on that system can be turned off and made unavailable until that system comes back online.

Asynchrony – this is probably one of the more important design considerations in any distributed application. It essentially means that each service, or functional piece of the system, communicates with each of its external dependencies asynchronously, so that slow or unavailable services do not directly impact the primary functioning of the application. This also typically implies that more operations aren’t tightly coupled, like a transaction, and don’t require services to be available to handle requests.

Monitoring:

Extensive monitoring and logging is essential to any complex distributed system. Having many services each with a different purpose, yet still interacting with one another, can lead to highly unusual bugs when they occur. It can be hard to tell where the problem lies and where the issue needs to be resolved. One of the best ways to mitigate this confusion and help diagnose problems quickly is to be sure that all system interfaces and APIs are monitored.

However, monitoring in large-scale web systems can be challenging.

**Separation and services adds complexity**.

As noted earlier, a key part of scaling systems is to break up the pieces of the system into services, but because each of these services is independent of one another it can make problems harder to diagnose. There are many different points of control and they don’t necessarily operate in sync with one another, making traditional sequential monitoring, or tracing the pattern of execution (like a debugger would with breakpoints in a program) much more complicated.

Furthermore, with most of these systems the communication between them can be delayed and complicated through mechanisms like retries (which is what happens when one service makes a request, like fetching an image, and the other responds with an error, such as being too busy to serve the request, and the requesting service retries the request at a later time), which only compounds the problem of tracing sequential events.

**Monitoring can have an impact.**

Another challenge with these large-scale systems is adding the monitoring to the system. Too much monitoring or logging can cause delays, take up space, and potentially interfere with normal operations. This is one of the reasons why many software systems have a debug (or noisy) mode and a production mode with less logging and information. Monitoring may or may not live on the same physical hardware, and in the event that it doesn’t, there is additional communication required for the monitoring systems to track and record metrics. This extra step is another layer of interaction, which can add more complexity in understanding these systems.

Despite all the obstacles with monitoring though, extensive metrics are a key part to understanding and diagnosing problems.

Here are some best practices in distributed system monitoring:

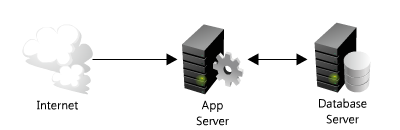
* **Monitor end-to-end functionality**. End to end monitoring typically consists of an operation, or path from start to finish within the system. So for the image hosting example, this would be uploading an image, and ensuring that the image is written as expected and can be retrieved from its storage location by the client. It is the complete use case for the web system. Of times there are more than one of these cases and each one should be monitored independently.
* **Monitor each service independently of one another.** Since each service has its own focus of control, making sure that each one is monitored is a key part to recognizing or diagnosing problems within that service. The monitoring for each service may not be the same, but often there is some overlap in monitored metrics (like standard system metrics, i.e. CPU usage, memory usage, disk and network i/o etc.). In the image hosting example, some of the service specific monitoring would be: the speed of reads, how many concurrent reads were happening for the Image retrieval Service, whereas the Image Write Service would watch the write queue, number of connections.
* **Monitor “to the glass” metrics** – look at things from the end users perspective. An important part of any large web system is to really understand what the end user’s experience is with the web site. It is not enough to understand the internal workings, but it is important to also monitor and track the overall experience from the client. Typically this is done using an external service that will in its simplest form ping the site to ensure that it is up, and in more complex cases actually execute end-to-end use cases. This sort of metric can help diagnose problems that occur somewhere between the client and the website, and can be one of the fastest ways to uncover network problems. For global user bases, this sort of monitoring may be geographically distributed such that will take into account the different networks.
* **Monitor at a frequency to detect issues before they impact customers** (this also means having enough time to address the issue, so also understanding the rate of change and next steps to address it). Another key part of any sort of monitoring is ensuring that the data is sampled frequently enough to detect problems, and raise alerts before it becomes an issue. For example, if there are too many requests to upload images simultaneously but throughput is only logged every few minutes, it may not be frequent enough to detect these peak usage periods that are causing problems.
* **Establish a baseline on historical performance.** In order to really make the most of the metrics, one must understand what is “normal” for the system. Without a baseline or clear history it is really difficult to determine if something is wrong and should be investigated. For example, problems caused from too much load or traffic on the system would be very hard to diagnose if there was not an easy way to understand what type of load or throughput was typical. Therefore it is not just enough to track metrics, but to also look at trends over time.
* **Monitor the monitoring systems**. In addition to having monitoring systems keeping track of events within the system, it is also key to make sure that the monitoring is up and reporting as expected (otherwise there is no way to know that there is even a problem!). Most of the time it is sufficient to have something as simple as an external ping to ensure the monitoring software is responsive (and most have web front ends that make this easy.

While these are not necessarily required, thinking about these things ahead of time will certainly help ensure more resiliency and faster recovery in the event of failure. There are many great open source options for monitoring, logging and tracking events in web systems, [Nagios](http://www.nagios.org/), [Zabbix](http://www.zabbix.com/), [Flume](https://github.com/cloudera/flume/wiki), and [Munin](http://munin-monitoring.org/) are all popular choices; and it is not uncommon to use more than one for the same system.

Having covered some of the core considerations in designing distributed systems, let’s now talk about the hard part – scaling access to the data.

## The building blocks of fast and scalable data access

Most very simple web applications (for example, LAMP stack applications) often look something like the following:



As they grow, there are two main challenges – scaling access to the app server and to the database. In a highly scalable application design, the app (or web) server is typically minimized and often embodies a *shared nothing architecture* (where each server is a copy of and operates independently of the other). This makes the app server layer of the system horizontally scalable. As a result of this design, the heavy lifting is pushed down the stack to the database server and supporting services; it’s at this layer where the real scaling and performance challenges come into play.

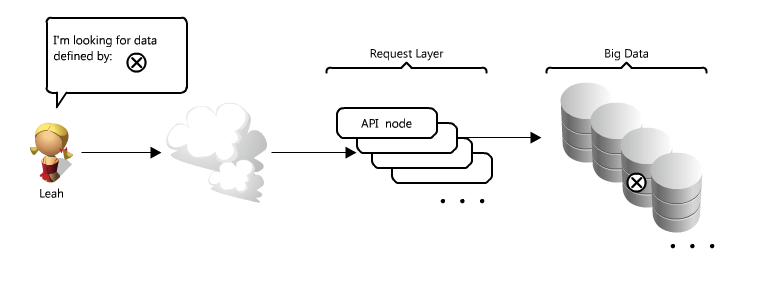
The rest of this chapter is devoted to some of the more common strategies and methods for making these types of services fast and scalable by providing fast access to your data.

Most systems can be oversimplified to this:



This is a great place to start. If you have a lot of data, you want fast and easy access, like a secret stash of candy in the top drawer of your desk. Though overly simplified, the previous statement hints at two hard problems: scalability of storage and fast access of data.

For the sake of this section, let’s assume you have many terabytes of data (TB) and you want to allow users to access small portions (KBs) of that data at random. This is similar to locating an image file somewhere on the larger file server in the image application example.



This is particularly challenging because it can be very costly to load TBs of data into memory – this directly translates to disk IO. Reading from disk is many times slower than from memory – memory access is almost as fast as Chuck Norris, whereas disk access is slower than the line at the DMV; and this speed difference really adds up for large data sets. Moreover, even with unique ID’s – solving the problem of knowing where to find that little bit of data can be an arduous task. It is like searching for a needle in a really big haystack, or trying to get that last Jolly Rancher in your candy stash without looking.

Thankfully there are many options that you can employ to make this easier; and three of the more important ones are caches, proxies, indexes and load balancers. The rest of this section goes through each of these concepts and discusses how each can be used to make data access a lot faster.

Caches

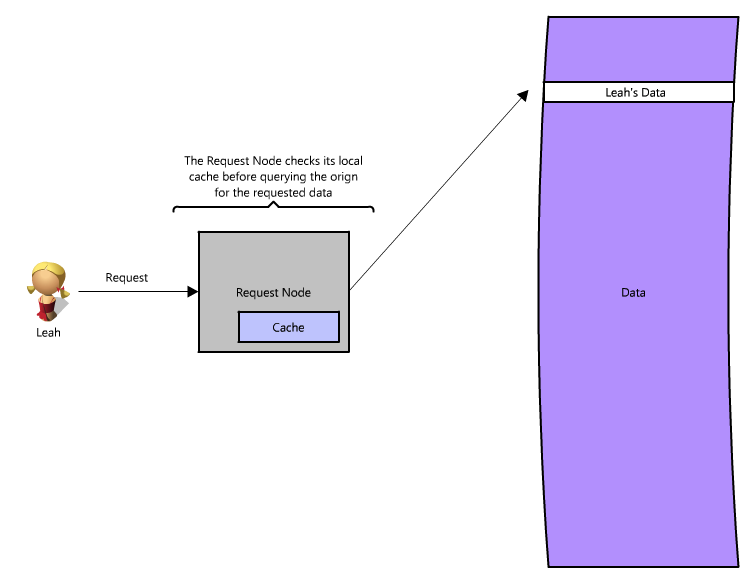
What is a cache? At the basic level, a cache is a transparent storage layer that provides fast access to select portions of the data set.

Caches take advantage of the locality of reference principle – recently requested data is likely to be requested again. They are used in almost every layer of computing, such as your hardware, the operating system, web browsers, web applications and more. A cache is like short-term memory; it has a limited amount of space, but is typically faster than the original data source, and contains the most recently accessed items. Caches can exist at all levels in your architecture, but it's often found at the level nearest to the frontend, where it is implemented to quickly return data without taxing the downstream levels

How can a cache be used to make your data access faster in our API example?

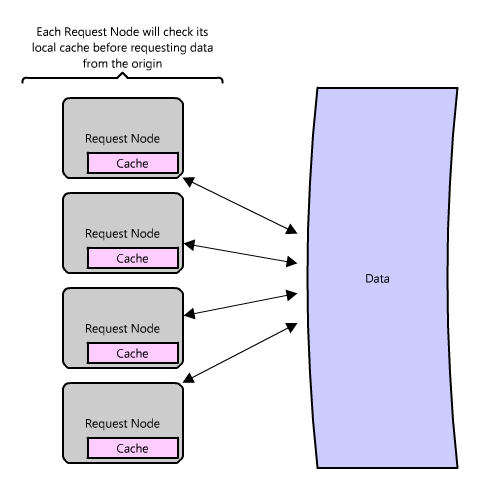
In this case, there are a couple of places you can insert a cache:

One option is to insert a cache on your request layer node:



Placing a cache directly on a request layer node enables the local storage of response data. Each time a request was made to the service, the node would quickly return local, cached data if it existed. If it were not there, the request node would query the data from disk. The cache on one request layer node could also be located both in memory (which is very fast) and on the node’s local disk (faster than going to network storage).

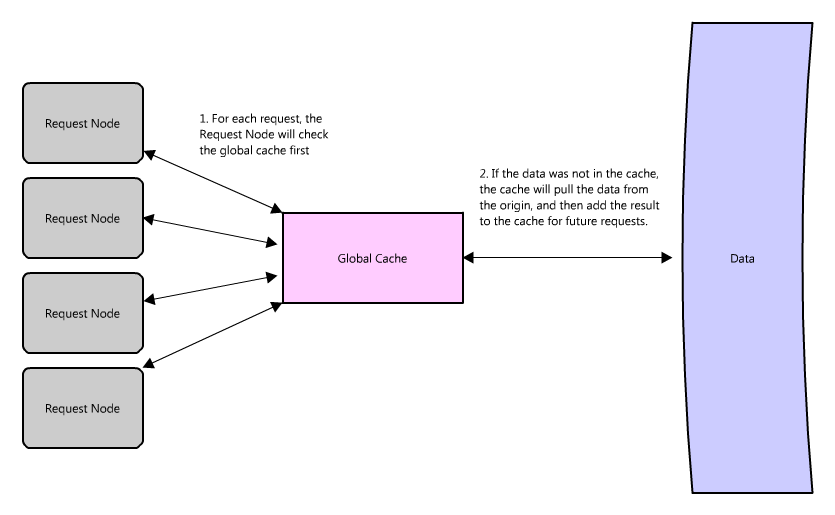
What happens when you expand this to many nodes?

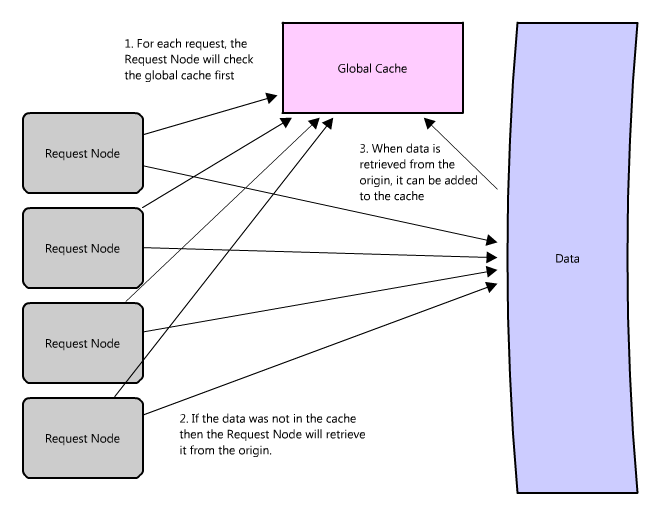


If the request layer were expanded to multiple nodes, it’s still quite possible to have it each host its own cache. However, if your load balancer randomly distributes requests across the nodes, the same request will go to different nodes, thus increasing cache misses. Two choices for overcoming this hurdle are global caches and distributed caches. Read on to learn more!

**Global Cache**

A global cache is just as it sounds – all the nodes use a single cache space. This involves adding a server, or file store of some sort; faster than your original store and accessible by all the request layer nodes. Each of the request nodes queries the cache in the same way it would a local one. This kind of caching scheme can get a bit complicated, but is very effective in some architectures (particularly ones with specialized hardware that make this global cache very fast).



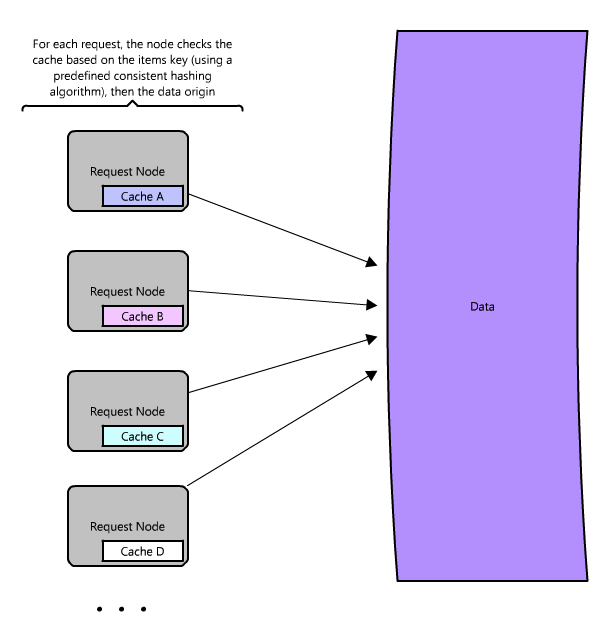
****

There are two common forms of global caches depicted in the diagrams. In the first diagram, when a cached response is not found in the cache, the cache itself becomes responsible for retrieving the missing piece of data from the underlying store; and in the second it is the responsibility of request nodes to retrieve any data that is not found in the cache.

**Distributed Cache**

In a distributed cache, each of its nodes own part of the cached data, in the same way that countries make up Planet Earth. Typically the cache is divided up using a consistent hashing function, such that if a request node is looking for a certain piece of data it can quickly know where to look within the distributed cache to determine if that data is available. In this case, each node has a small piece of the cache, and will then send a request to another node for the data, before going to the origin. Therefore, one of the advantages of a distributed cache is the increased cache space that can be had just by adding additional nodes to the request pool.

A disadvantage of distributed caching is remedying a missing node. Some distributed caches get around this by storing multiple copies of the data on different nodes, however you can imagine how this logic can get complicated quickly, especially when you add or remove nodes from the request layer. Although, even if a node disappears and part of the cache is lost, the requests will just pull from the origin – so it isn’t necessarily catastrophic!



The great thing about caches is that they usually make things much faster (implemented correctly of course!) The methodology you choose just allows you to make it faster for even more requests. However all this caching comes at the cost of having to maintain additional storage space, typically in the form of expensive memory; nothing is free. Caches are wonderful for making things generally faster, but moreover provide system functionality under high load conditions when otherwise there would be complete service degradation

One example of a popular open source cache is [Memcached](http://memcached.org/) (which can work both as a local cache and distributed cache), however there are many other options (including many language or framework specific options). Memcached is used in many large web sites, and even though it can be very powerful, it is simply an in-memory key value store, optimized for arbitrary data storage and fast lookups (O(1)).

Now let’s talk about what to do when the data isn’t in the cache…. ([say what!?](http://www.youtube.com/watch?v=Qw9oX-kZ_9k))

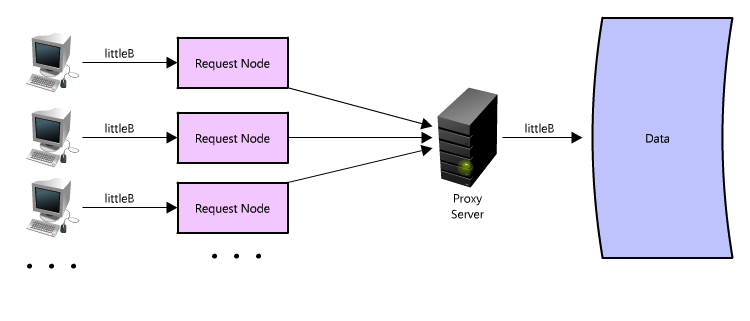
Proxies

At a basic level, a proxy server is an intermediary piece of hardware/software that receives requests from clients and relays them to the backend origin servers. Typically, proxies are used to filter requests, log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compression).

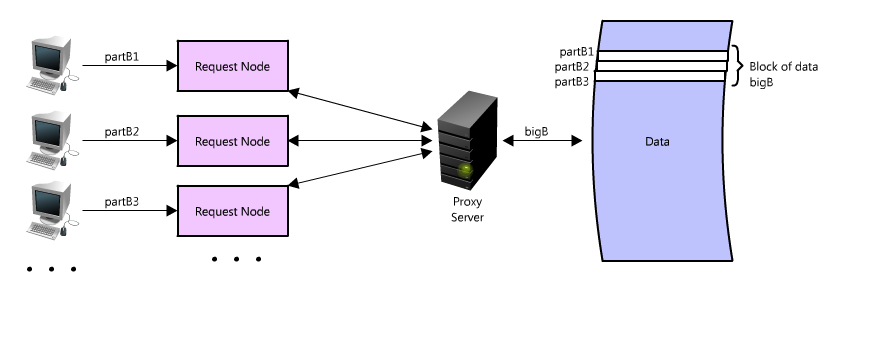


Proxies are also immensely helpful when coordinating requests from multiple servers, providing opportunities to optimize request traffic from a system-wide perspective. One way to use a proxy to speed up data access is to collapse the same (or similar) requests together into one request, and then return the single result to the requesting clients. This is known as collapsed forwarding.

Imagine there is a request for the same data (let’s call it littleB) across several nodes, and that piece of data is not in the cache. If that request is routed thought the proxy, then all of those requests can be collapsed into one; which means we only have to read littleB off disk once. This will improve performance in high load situations, particularly when that same data is requested over and over.



Another great way to use the proxy is to not just collapse requests for the same data, but also to collapse requests for data that is spatially close together in the origin store (consecutively on disk); employing such a strategy maximizes data locality for the requests, which can result in decreased request latency. For example, let’s say a bunch of nodes request parts of B – partB1, partB2, etc. We can setup our proxy to recognize the spatial locality of the individual requests, collapsing them into a single request and returning only bigB, greatly minimizing the reads from the data origin. This can make a really big difference in request time when you are randomly accessing across TBs of data! Proxies are especially helpful under high load situations, or when you have limited caching since they essentially can batch several requests into one.



It is worth noting that you can use proxies and caches together, but generally it is best to put the cache in front of the proxy, similarly, it is best to put the horse in front of the carriage. This is because the cache is serving data from memory, it is very fast, and doesn’t mind multiple requests for the same result. But if the cache was located on the other side of the proxy server, then there would be additional latency with every request before the cache, and this could hinder performance.

If you are looking at adding a proxy to your systems, there are many options to consider (there are some listed here: <http://en.wikipedia.org/wiki/Web_accelerator>); [Squid](http://www.squid-cache.org/) and [Varnish](https://www.varnish-cache.org/) have both been road tested and are widely used in many production web sites. These proxy solutions offer many optimizations to make the most of client-server communication. Installing one of these as a reverse proxy at the web server layer can improve web server performance considerably, reducing the amount of work required to handle incoming client requests.

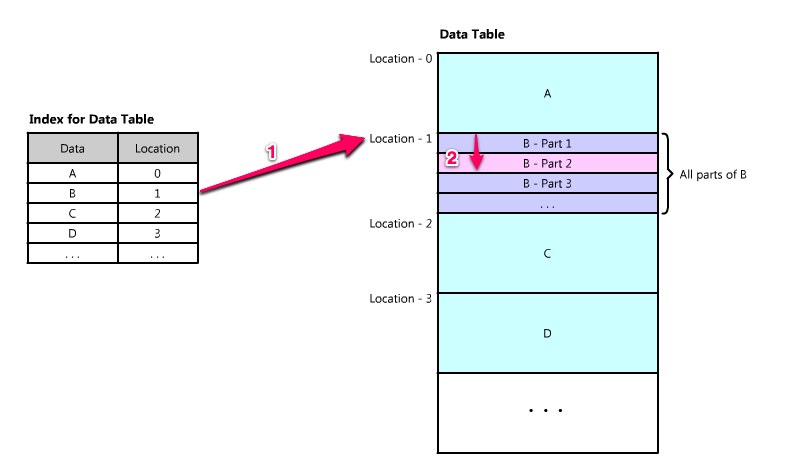
Indexes

Using an index to access your data quickly is a well-known strategy for optimizing data access performance, and probably the most well known when it comes to databases. An index makes the trade-offs of increased storage overhead and slower writes (since you must both write the data and update the index) for the benefit of faster reads.

Just as to a traditional relational data store, you can also apply this concept to larger data sets. The trick with indexes is you must carefully consider how users will access your data. In the case of data sets that are many TBs large, with very small payloads (e.g. 1kb), indexes are a necessity for optimizing data access. Finding a small payload in such a large data set can be a real challenge since you can’t possibly iterate over that much data in any reasonable time. Furthermore, it is very likely that such a large data set is spread over several (or many!) physical devices – this means you need some way to find the correct physical location of the desired data.

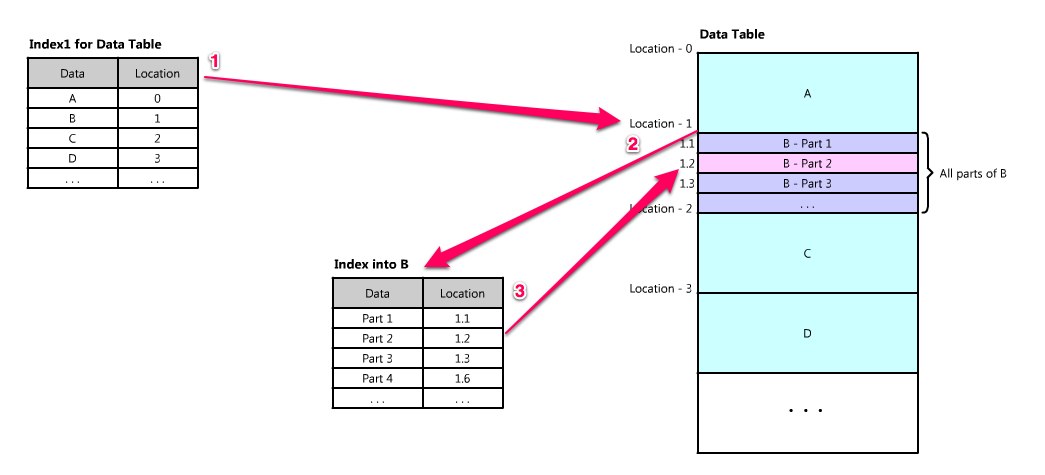
What is the best way to do this? Enter indexes!

An index can be used like a table of contents that directs you to the right location where your data lives. These indexes provide a fast lookup directing you to the locality of the data; just like the card catalog in a library. For example, let’s say you are looking for a piece of data, part 2 of B – how will you know where to find it? If you have an index that is sorted by data type – say data A, B, C – it would tell you the location of data B at the origin. Then, you just have to seek to that location and read the part of B you want.



These indexes are often stored in memory, or somewhere very local to the incoming client request. Berkeley DBs (BDBs) and other tree like data structures are commonly used to store data in ordered lists ideal for access with an index.

Often times, there are many layers of indexes that serve as a map – moving you from one location to the next, and so forth – until you get the specific piece of data you want.

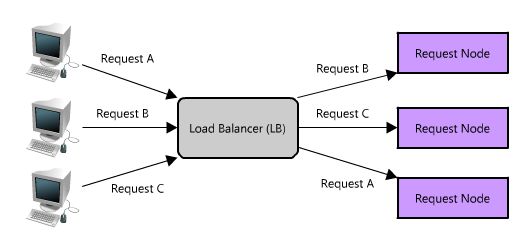


Indexes can also be used to create several different views of the same data. For large data sets, this is a great way to define different filters and sorts without resorting to creating many additional copies of the data.

Being able to find your data quickly and easily is important; indexes are an effective and simple tool to achieve this.

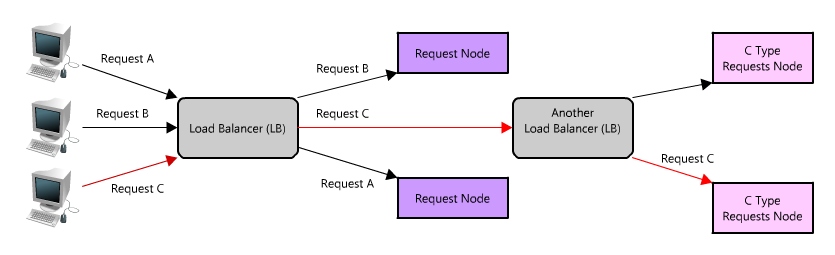
Load Balancers

Finally another critical piece of any distributed system are load balancers (LBs). LBs are a principal part of any architecture as their role is to distribute load across a set of nodes responsible for servicing requests. This allows multiple nodes to transparently service the same function in a system. Their main purpose is to handle a lot of simultaneous connections and route those connections to one of the request nodes – allowing the system to scale to service more requests by just adding nodes.



There are many different algorithms that can be used to service requests, including picking a random node, round robin (http://en.wikipedia.org/wiki/Round-robin), or even selecting the node based on certain criteria – such as memory or CPU utilization. LBs can be implemented as software or hardware appliances. One open source software load balancer that has received wide adoption is [HAProxy](http://haproxy.1wt.eu/).

In a distributed system, load balancers are often found at the very front of the system such that all incoming requests are routed accordingly. In a complex distributed system, it is not uncommon for a request to be routed to multiple load balancers as shown below.

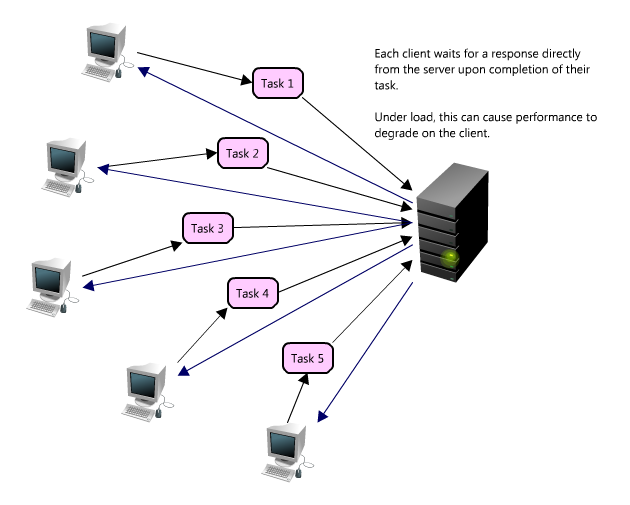


Similar to proxies, some load balancers can also route a request differently depending on the type of request (technically these are also known as reverse proxies). LBs are an easy way to allow you to expand system capacity and like the other techniques in this article, play an essential role in distributed system architecture. Load balancers also provide a critical function of being able to test the health of a node, such that if a node is unresponsive or over loaded, it can be removed from the pool handling requests – taking advantage of the redundancy of different nodes in your system.

Queues

So far we have covered a lot of ways to read data, quickly, but another important part of scaling the data layer is effective management of writes. When systems are simple, having minimal processing loads and small databases, writes can be predictably fast, however in more complex systems writes can take an almost non-deterministically long time (for example, data may have to be written several places on different servers or indexes, or the system could just be under high load). In the cases where writes, or any task for that matter, may take a long time, achieving performance and availability requires building asynchrony into the system; and a common way to do that is with queues.

Imagine a system where each client is requesting a task to be remotely serviced. Each of these clients sends their request to the server, where the server completes the tasks as quickly as possible and returns the results to their respective clients. In small systems where one server (or logical service) can service incoming clients just as fast as they come in this sort of situation should work just fine. However, when the server receives more requests than it can handle, then each client is forced to wait for the other clients’ requests to complete before a response can be generated. This is an example of a synchronous request and depicted below.

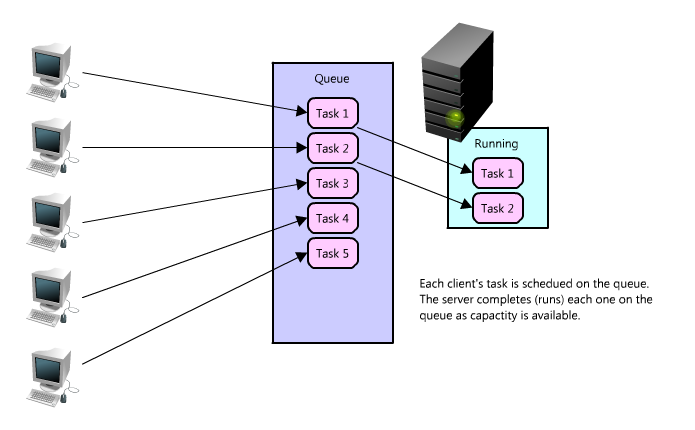


This kind of synchronous behavior can severely degrade client performance; the client is forced to wait, effectively performing zero work until its request can be answered. Adding additional servers to address system load does not solve the problem either, even with effective load balancing in place it is extremely difficult to ensure the even and fair distribution of work required to maximize client performance. Further, if the server handling requests is unavailable, or fails, then the clients upstream will also fail. Solving this problem effectively requires abstraction between the client’s request and the actual work performed to service it.

Enter queues. A queue is as simple as it sounds, a task comes in, is added to the queue and then workers pick up the next task as they have the capacity to process it. These tasks could represent writes to a database, or something as complex as generating a thumbnail preview image for a document. When a client submits task requests to a queue they are no longer forced to wait for the results, instead they need only acknowledgement that the request was properly received. This acknowledgement can later serve as a reference for the results of the work when the client requires it.

Queues enable clients to work in an asynchronous manner, providing a strategic abstraction of a client’s request and its response. On the other hand, in a synchronous system, there is no differentiation between request and reply, and therefore cannot be managed separately. In an asynchronous system the client requests a task, the service responds with a message acknowledging the task was received, and then the client can periodically check the status of the task, only requesting the result once it has completed. While the client is waiting for an asynchronous request to be completed it is free to perform other work, even making asynchronous requests of other services. The latter is an example of how queues and messages are leveraged in distributed systems.

Queues also provide some protection from service outages and failures. For instance, it is quite easy to create a highly robust queue that can retry service requests having failed to due transient server failures. It is more preferable utilizing a queue to enforce quality-of-service guarantees than exposing clients directly to intermittent service outages requiring complicated and often-inconsistent client-side error handling.



Queues are fundamental in managing distributed communication between different parts of any large-scale distributed system, and there are lots of ways to implement them. There are quite a few open source queues like [RabbitMQ](http://www.rabbitmq.com/), [ActiveMQ](http://activemq.apache.org/), [BeanstalkD](http://kr.github.com/beanstalkd/), but some also use services like [Zookeeper](http://zookeeper.apache.org/), or even data stores like [Redis](http://redis.io/).

## In conclusion

Designing efficient systems with fast access to lots of data is exciting, and there are lots of great tools that enable all kinds of new applications. This chapter covered just a few examples, barely scratching the surface, but there are many more—and there will only continue to be more innovation in the space.

Footnotes:

[1] Unusual software bug, <http://en.wikipedia.org/wiki/Unusual_software_bug>

[2] Fallacies of Distributed Computing, <http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing>)

Fault tolerance techniques for distributed systems, http://www.ibm.com/developerworks/rational/library/114.html

<http://highscalability.squarespace.com>

Building for the Cloud is Building for Scalability, <http://www.productionscale.com/home/2010/9/28/building-for-the-cloud-is-building-for-scalability.html>

<http://en.wikipedia.org/wiki/CAP_theorem>

Towards Robust Distributed Systems, <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>

You Can’t Sacrifice Partition Tolerance, <http://codahale.com/you-cant-sacrifice-partition-tolerance/>

Lessons from Giant Scale Services, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.4274&rep=rep1&type=pdf>

Amdahl’s Law, http://en.wikipedia.org/wiki/Amdahl%27s\_law