Ch.1

Reliable, Scalable, and Maintainable Applications

“Many applications today are data-intensive, as opposed to compute-intensive. Raw CPU power is rarely a limiting factor for these applications—bigger problems are usually the amount of data, the complexity of data, and the speed at which it is changing.

A data-intensive application is typically built from standard building blocks that pro‐ vide commonly needed functionality. For example, many applications need to:

* Store data so that they, or another application, can find it again later (databases)
* Remember the result of an expensive operation, to speed up reads (caches)
* Allow users to search data by keyword or filter it in various ways (search indexes)
* Send a message to another process, to be handled asynchronously (stream pro‐ cessing)
* Periodically crunch a large amount of accumulated data (batch processing)”

**Thinking About Data Systems**

Data systems are typically comprised of various components that can do their respective jobs efficiently (database, cache, message queues). These services are stitched together in the application code.

APIs provide an interface for a service. The API hides the implementation of the service from the client.

“In this book, we focus on three concerns that are important in most software systems:

* *Reliability* The system should continue to work correctly (performing the correct function at the desired level of performance) even in the face of adversity (hardware or soft‐ ware faults, and even human error)
* *Scalability* As the system grows (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth
* *Maintainability* Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be able to work on it productively”

**Reliability**

* Application performs as expected
* It can tolerate user error and unexpected use of the software
* It performs well under the expected load and volume for the use case
* The system prevents unauthorized access and abuse (it is secure)

Reliability can be roughly defined as “continuing to work correctly, even when things go wrong.”

*Faults*: one component of the system deviating from its spec

*Failures*: the system as a whole stops providing the required service to the user

*Fault-tolerant* or *resilient* systems anticipate faults and handle them

To ensure fault tolerance and catch bugs companies often intentionally trigger faults (e.g. Netflix Chaos Monkey)

**Hardware Faults**

“Hard disks crash, RAM becomes faulty, the power grid has a blackout, someone unplugs the wrong network cable. Anyone who has worked with large datacenters can tell you that these things happen all the time when you have a lot of machines.”

One solution is to “add redundancy to the individual hardware components in order to reduce the failure rate of the system. Disks may be set up in a RAID configuration, servers may have dual power supplies and hot-swappable CPUs, and datacenters may have batteries and diesel generators for backup power”

This solution doesn’t prevent the failure of an entire machine from taking down the system.

Software fault-tolerance techniques can be used on distributed systems so that entire servers can fail without disrupting the application

**Software Errors**

While hardware faults are typically uncorrelated, software faults can be correlated across nodes and can trigger further system failures. Examples:

* A software bug that causes every instance of an application server to crash when given a particular bad input. For example, consider the leap second on June 30, 2012, that caused many applications to hang simultaneously due to a bug in the Linux kernel
* A runaway process that uses up some shared resource—CPU time, memory, disk space, or network bandwidth
* A service that the system depends on that slows down, becomes unresponsive, or starts returning corrupted responses
* Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults

**Human Errors**

Humans build, design, and operate software systems, and are liable to make mistakes in the design, code, or configuration of the system.

Ways to make systems reliable in spite of potential human error:

* Well-designed abstractions, APIs, and admin interfaces limit the way users interact with the system
* Provide sandbox environments for people to explore and experiment safely
* Test thoroughly, from unit tests to whole-system integration tests and manual tests. Use automated testing
* Enable fast and easy recovery, with the ability to roll back config changes, roll out new code gradually to subsets of users, and provide tools to recompute data in case the original computation was wrong
* Set up detailed monitoring like performance metrics and error rates (referred to in other engineering disciplines as *telemetry*)
* Utilize logging

**Scalability**

“Even if a system is working reliably today, that doesn’t mean it will necessarily work reliably in the future.”

A common reason for performance degradation is increased load. 10xing your user base can result in an overloaded system.

*Scalability* is defined as a system’s ability to cope with increased load

Scalability must be discussed in specific terms. If the system grows in a particular way, how can it handle the increase? How can we add computing resources to handle the increased load?

**Describing Load**

First, describe the current load on the system. Then contemplate what-if scenarios impact on the system (e.g. load doubles).

*Load parameters* are used to quantitatively describe load

“The best choice of parameters depends on the architecture of your system: it may be requests per second to a web server, the ratio of reads to writes in a database, the number of simultaneously active users in a chat room, the hit rate on a cache, or something else. Perhaps the average case is what matters for you, or perhaps your bottleneck is dominated by a small number of extreme cases”

Twitter example is great, won’t cover here

**Describing Performance**

Once load parameters are defined, you can investigate load increases in 2 ways:

* When you increase a load parameter and keep system resources unchanged, how is performance impacted
* When you increase a load parameter, how much do resources need to increase to keep performance the same

These questions require a quantification of performance

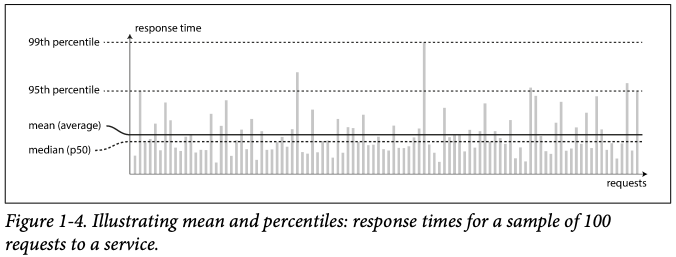
**Batch processing systems** care about *throughput* – total time it takes to run a job on a dataset of a certain size

**Online systems** care about *response time* – time between a client sending a request and receiving a response

Aside: “In an ideal world, the running time of a batch job is the size of the dataset divided by the throughput. In practice, the running time is often longer, due to skew (data not being spread evenly across worker processes) and needing to wait for the slowest task to complete.”

Latency vs response time: “The response time is what the client sees: besides the actual time to process the request (the service time), it includes network delays and queueing delays. Latency is the duration that a request is waiting to be handled—during which it is latent, awaiting service”

Response time can vary for the same request, so it is best to think of it in terms of distributions. There are occasionally outliers



**Approaches for Coping with Load**

* *Scaling up* (increase capacity of single same machine) vs *scaling out* (across multiple machines)
* Scaling out also known as a *shared-nothing* architecture
* *Elastic* systems can add or deallocate computing resources based on the load – good if load is highly unpredictable
* Manually scaled systems are often simpler and have fewer operational surprises
* Stateless services are easy to scale across nodes, but stateful data systems can be highly complex to scale horizontally

“The architecture of systems that operate at large scale is usually highly specific to the application—there is no such thing as a generic, one-size-fits-all scalable architecture (informally known as magic scaling sauce). The problem may be the volume of reads, the volume of writes, the volume of data to store, the complexity of the data, the response time requirements, the access patterns, or (usually) some mixture of all of these plus many more issues.

For example, a system that is designed to handle 100,000 requests per second, each 1 kB in size, looks very different from a system that is designed for 3 requests per minute, each 2 GB in size—even though the two systems have the same data through‐ put.”

**Maintainability**

Majority of software costs are not in developing new software, but maintaining existing software

Dealing with *legacy* systems can be painful due to challenges like fixing the mistakes of previous devs, working with outdated platforms, and dealing with systems that were forced to do things they were not intended for

3 design principles can improve the maintainability of software:

* **Operability:** Make it easy for operations teams to keep the system running smoothly
* **Simplicity:** Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system. (Note this is not the same as simplicity of the user interface.)
* **Evolvability:** Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as extensibility, modifiability, or plasticity

Operability: Making Life Easy for Operations

“Operations teams are vital to keeping a software system running smoothly. A good operations team typically is responsible for the following, and more

* Monitoring the health of the system and quickly restoring service if it goes into a bad state
* Tracking down the cause of problems, such as system failures or degraded performance
* Keeping software and platforms up to date, including security patches
* Keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage
* Anticipating future problems and solving them before they occur (e.g., capacity planning)
* Establishing good practices and tools for deployment, configuration management, and more
* Performing complex maintenance tasks, such as moving an application from one platform to another
* Maintaining the security of the system as configuration changes are made
* Defining processes that make operations predictable and help keep the production environment stable
* Preserving the organization’s knowledge about the system, even as individual people come and go

**Simplicity: Managing Complexity**

Some symptoms of complexity

* Explosion of the state space
* Tight coupling of modules
* Tangled dependencies
* Inconsistent naming and terminology
* Hacks aimed at solving performance problems
* Special-casing to work around issues elsewhere

Causes budgets and timelines to get overextended

Makes bugs more likely

*Abstraction* is a great tool for removing accidental complexity. It provides a clean, simple interface for using the application or service, and makes reuse simpler and more efficient

“**For example, high-level programming languages are abstractions that hide machine code, CPU registers, and syscalls**”

**Evolvability: Making Change Easy**

Requirements for the application are in constant flux

*Agile* provides a framework for adaptive change, including patterns like TDD and refactoring

“In this book, we search for ways of increasing agility on the level of a larger data system, perhaps consisting of several different applications or services with different characteristics”

*Evolvability* is used to refer to agility on a data system level

**Summary**

“An application has to meet various requirements in order to be useful. There are *functional requirements* (what it should do, such as allowing data to be stored, retrieved, searched, and processed in various ways), and some *nonfunctional requirements* (general properties like security, reliability, compliance, scalability, compatibility, and maintainability). In this chapter we discussed reliability, scalability, and maintainability in detail.

**Reliability** means making systems work correctly, even when faults occur. Faults can be in hardware (typically random and uncorrelated), software (bugs are typically systematic and hard to deal with), and humans (who inevitably make mistakes from time to time). Fault-tolerance techniques can hide certain types of faults from the end user.

**Scalability** means having strategies for keeping performance good, even when load increases. In order to discuss scalability, we first need ways of describing load and performance quantitatively. We briefly looked at Twitter’s home timelines as an example of describing load, and response time percentiles as a way of measuring performance. In a scalable system, you can add processing capacity in order to remain reliable under high load.

**Maintainability** has many facets, but in essence it’s about making life better for the engineering and operations teams who need to work with the system. Good abstractions can help reduce complexity and make the system easier to modify and adapt for new use cases. Good operability means having good visibility into the system’s health, and having effective ways of managing it.”