Brayan Quezada

IST 690: Designing Data-Intensive Applications

Dr. Gregory Block

Final Research Paper

## **Introduction**

Many systems that are used today are data-intensive instead of computer-intensive. A system is referred to as data-intensive when data is the systems’ primary challenge. Many of these challenges include: the amount of data, complexity, and the speed at which the data is changing. Whereas compute-intensive main challenge is focused on CPU cycles. It’s also important to note that different systems have different requirements, and it is required to figure out which approaches are most appropriate for the task at hand especially since many systems have demanding requirements that a single tool is not sufficient enough to meet data processing and storage needs (Kleppmann, 2019, p. 6). Instead, the work would be broken down into tasks that can be performed efficiently on a single tool, and the different tools are stitched together using system pipelines. When combining the tools to provide a service, a robust orchestration platform would usually hide the implementations from a client, providing a smooth process on their end (Kleppmann, 2019, p. 18). When designing a data system it is important to consider the following questions:

* How to ensure that the data remains correct and complete, even when things go wrong internally?
* How do you provide consistently good performance to clients, even when parts of your system are degraded?
* How do you scale to handle an increase in load?
* What does a good API for the service look like?

In order to answer these questions there are three main things to focus on: reliability, scalability, and maintainability (Kleppmann, 2019, p. 22).

## **Reliability**

The meaning of reliability refers to the system performing the function that the process expects, it can tolerate the process making mistakes or using the software in an unexpected way, and prevents any unauthorized access and abuse. Now, if something goes wrong, this will be known as faults. A fault is defined as one component of a system deviating from its specifications. This should not be confused with a failure as a failure is when a system as a whole stops providing the required service to the process. A system that can anticipate and handle a fault would be called fault-tolerant, however, it is impossible to reduce the probability of a fault to zero. Counterintuitively, increasing the rate of faults by triggering them deliberately would ensure the fault-tolerance machinery is constantly being tested which boosts confidence that faults are being handled correctly when they naturally occur. There are many types of faults that can occur in a system, three in particular are hardware faults, software faults, and human faults (or errors) (Yuan et al, 2014).

A hardware fault occurs when the hard disk crashes, RAM becomes faulty, power grid has a blackout, or the net work cable was unplugged. When such fault occurs, the first response would be to add redundancy to an individual hardware component which would reduce the failure rate of system. For example, a disk can be set up in a RAID configuration, where servers have a dual power supply and swap able CPUS. So when one component dies, the redundant component will take its place. This approach, however, will not prevent hardware problems but it’ll keep the machine running for years. As the volume of data and computing demand increases, more systems will be using a larger number of machines, which increases the rate of hardware faults occurring.When using a cloud platform as a service (PaaS), it’s common for virtual machines to shut down without warning, as cloud platforms are designed to prioritize flexibility and elasticity over single-machine reliability (Ford et al, 2010).

A software fault is harder to anticipate since the fault is correlated across nodes, and tends to cause more system failures. An example of this would be a software bug that would cause the server to crash when a bad input is entered into the process. These type of bugs lie dormant for a long time and would be triggered by an unusual circumstance. This means that the software is making the assumption that the bug is some form of malware. Furthermore, unlike hardware faults, these problems are not easy to solve. However small things can help, such as: allowing processes to crash and restart, thorough testing, and analyzing the system behavior in production among other things (Gunawi et al, 2014).

Humans are the ones who design and build these systems and are also the operators who help keep the systems running. However, even humans are unreliable; for example, a study of large internet services found that configuration errors by operators (human errors) were the leading cause of outages, whereas hardware faults only play a role in about 10-25% of outages. One of many ways to overcome this is to design the system in a way that’ll minimize opportunities for error. A well designed API, and admin interface would make it easier to “follow directions”. However, be careful with this because if the interface is too restrictive people will tend to work around them, which would defeat its purpose (Oppenheimer et al).

Reliability is extremely important to keep in mind when designing a system. Bugs in a business system causes lost productivity, and outages which result in huge costs in lost revenue and reputation damage. But even if a system is reliable today, it doesn’t guarantee that it will be reliable in the near future. One common reason for this is an increase in load. This is where scalability comes into play (Kleppmann, 2019, p. 27).

## **Scalability**

The ability for a system to cope with an increase in load is known as scalability. However, it is not a one dimensional label that can be attached to a system. When testing the scalability of a system it is important to consider asking questions such as “if the system grows in a particular way, what are our options for coping with growth” or “how can we add computing resources to handle additional load?” To figure this out, the first step would be to describe the current load of the system. The load can be described with some numbers known as load parameters. The best choice of load parameter will depend on the architecture of the system. For instance, requests per second to a web server, or the number of simultaneously active users in a chat room (Kleppmann, 2019, p. 28).

The next step would be to investigate what happens when the load increases. There’s two questions that must be answered in order to figure this out: “when you increase a load parameter and keep the system resources unchanged, how is the performance of your system affected?” and “when you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged? The answer to both of these questions require performance numbers. In a batch processing system, it is important to pay attention to throughput, or the number of records that are processed per second, or the total time it takes to run a job on a dataset. In an online system, the main focus is the the service’s response time, which is the time between a client sending a request and receiving a response. Beware, however, it’s important to note that an architecture that can cope with one level of load is unlikely to cope with 10 times that load. Therefore, when working on a fast-growing service, it is important to rethink the architecture on every order of load increase (Kleppmann, 2019, p. 33).

There are many approaches to take in order to cope with an increase in load. It can be thought of either scaling up (vertical scaling, or moving to a more powerful machine) and scaling out (horizontal scaling, or distributing loads across multiple smaller machines) which is also known as a shared-nothing architecture. Some systems, however, are elastic, meaning they automatically add computing resources when detecting a load increase. An elastic system can be useful if the load is highly unpredictable, but a manually scaled system is much more simple and less operational surprises (Kleppmann, 2019, p. 38).

## **Maintainability**

The majority of software costs usually arises from its ongoing maintenance such as, fixing bugs, keeping its systems operational, investigating failures, and even adding new features. It is important to pay attention to three design principles to ensure maintainability within the system; operability, simplicity, and evolvability (Kleppmann, 2019, p. 39).

The process of making it easy for operations teams to keep the system running smoothly is known as operability. Even though some aspects of operations can be automated (and should be), it is up to the user (or human) to set up the automation and make sure it’s functioning correctly. The operations team is vital in keeping the system running smoothly. Some of their responsibilities are: monitoring the health of a system and restarting the service if it goes into a bad state, keeping tabs on how different systems affect each other, and maintaining the security of the system as configuration changes are made. Aside from operations, the system itself can do several things to ensure routine tasks run smoothly. It can provide visibility into the runtime behavior of the system, provide good documentation and easy-to-understand operational model, and allow the system to fix itself while giving administrators the flexibility to control the system state when needed (Hamilton, 2007).

Managing complexity would be known as simplicity. There are various symptoms of complexity to keep in mind of, such as: explosion of the state space, inconsistent naming/terminology, and tangled dependencies. Complexity tends to make maintenance hard and when that occurs, budgets and scheduled often overrun. Reducing complexity will improve the maintainability of the software and it’s a key goal that must be ensured when building a system. Furthermore, making a system simpler does not mean the functionality is being reduced. It means removing “accidental complexity.” Mosley and Marks define complexity as accidental if it’s ”not inherent in the problem that the software solves but arises only from implementation.” The best way to get rid of these accidental complexities is abstraction. It’ll hide implementation detail behind an easier-to-understand façade (Foote and Yoder, 1997).

The ability of making easy changes is known as evolvability. This is extremely important as it is unlikely a system requirements will remain unchanged forever. In a constant flux, many things can occur, such as: business priorities may change, new platforms replace old platforms, or the system may force architectural changes. The ability to easily modify a system, and adapt it to changing requirements is linked to simplicity and abstractions. Simple and easy-to-understand systems are easier to modify than complex systems. In other words, the more agile a data system is the higher the chance of evolvability (Breivold et al, 2008).

## **Conclusion**

In order to test the ideas mentioned above. A virtual machine was generated in order to create a system using the platforms Docker, Apache Kafka and Apache Cassandra. The idea would be to initially load the data onto Kafka using a command-line Python driver; which would generate traffic and serve as a Kafka producer. The next steps would be to use a Kafka/Cassandra connector, which would allow Cassandra to be configured as a Kafka consumer. This would allow the Kafka engine to automatically start writing records to Cassandra without using a third-party process and eliminate the need for additional providers to push the data along. Unfortunately, the process wasn’t successful since the Docker implementation on the virtual machine was too basic and as a result wasn’t able to support the code and settings needed to integrate the Kafka/Cassandra connector. However, this was an interesting short-lived project that visualize just how complex it is to build a system. It is important to take into account the type of platform being used in the system, since each platform functions differently form the other. it is also important to ensure the system can handle the data that is being generated and/or loaded and ensure that it is reliable, maintainable, and scalable.

**References**

Kleppmann, M. (2019). Designing Data-Intensive Applications the Big Ideas behind Reliable, Scalable, and Maintainable Systems. O'Reilly.

[3] Yuan, D., Luo, Y., Zhuang, X., et al. (2014). Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[5] Ford, D., Labelle, F., Popovici, F., et al. (2010). Availability in Globally Distributed Storage Systems. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[8] Gunawi, H., Hao, M., Leesatapornwongsa, T., et al. (2014). What Bugs Live in the Cloud?. 5th ACM Symposium on Cloud Computing (SoCC).

[13] Oppenheimer, D., Ganapathi, A., and Patterson, D. (2003). Why Do Internet Services Fail, and What Can Be Done About It?. 4th USENIX Symposium on Internet Technologies and Systems (USITS).

[29] Hamilton, J. (2007). On Designing and Deploying Internet-Scale Services. 21st Large Installation System Administration Conference (LISA).

[30] Foote, B., and Yoder, J. (1997). Big Ball of Mud. 4th Conference on Pattern Languages of Programs (PLoP).

[34] Breivold, H., Crnkovic, I., and Eriksson, P. (2008). Analyzing Software Evolvability. at 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC).