Chapter 1 Reliable, Scalable, and Maintainable Applications

Introduction

- Many applications are data-intensive (amount, the complexity of data are high), as opposed to computation-intensive.
- Most data-intensive applications are built from standard building blocks, such as databases, caches, etc.
- This book is about the principles and practicalities of data systems and how to use them to build data-intensive applications.

Thinking About Data Systems

- Why lump sum all under the name term data systems?
    - The boundary may not be clear, i.e., Redis, Kafka.
    - No single tool meets all needs. Need to stitch many systems with application code.
- Stitching smaller systems together results in a larger data system, with different characteristics.
- This book focus on three qualities that are important for data systems: Reliability, Scalability, and Maintainability.

Reliability

- Definition: Continue to work when faults (NOT failure!) occur. We say such a system is fault-tolerant.
- Faults are defined as components of the system deviating from the spec while failure is defined as a system stop working entirely.
- It is impossible to reduce faults to 0; therefore, we should design a system that tolerates faults.
- Introducing random faults (as in Netflix Chaos Monkey) could improve confidence in fault-tolerant systems.
- For security issues, we would prefer to prevent faults over tolerating them, as security breaches cannot be cured.

Hardware Faults

- In the past, people use redundant hardware to keep machines/services running.
- Recently, platforms are designed to prioritize flexibility and elasticity. Systems can tolerate the loss of whole machines. No downtime scheduled needed for single machine maintenance.

Software Errors

- Software errors/bugs are more systematic. They impact all machines in the same service.
- Alerts can help check the SLA guarantees.

Human Errors

- Most outages are human errors. We can
    - Minimize opportunities for errors through designs.
    - Decouple where mistakes are made (sandbox) and where the mistakes cause failures (production).
    - Test thoroughly, including unit, integration, and manual tests.

    - Allow quick and easy recovery to minimize impact.
    - Set up clear monitoring on performance metrics and error rates.

How Important Is Reliability

- Reliability is important for the business. Any downtime could be a revenue loss.
- There are situations where we may tradeoff reliability for lower development costs, but we should be very conscious when we are cutting corners.

Scalability

- Scalability is the term we use to describe a system's ability to cope with increased **load**.

Load

- Described with *load parameters*, which has different meaning under different architectures. It can be requests per second for services, read write ratio for databases, number of simultaneous users. Sometimes the average case matters, and sometimes the bottleneck is dominated by a few extreme cases.
- Twitter example
    - Twitter has two main operations: post Tweet and home timeline (~100x more requests than post Tweet).
    - Approach 1: If we store Tweets in a simple database, home timeline queries may be slow.
    - Approach 2: We can push Tweets into the home timeline cache of each follower when a Tweet is published.
    - Approach 2 does not work for users with many followers, since the approach would need to update too many home timeline caches.
    - Distribution of followers in this case is a load parameter.
    - We can use approach 1 for users with many followers and approach 2 for the others.

Performance

- Two ways to look at performance.
    - When we increase load parameters and keep resources unchanged. How is the performance affected.
    - When we increase load parameters how much resource do we need to keep performance unchanged.
- Batch processing systems cares about throughput (number of records processed per second).
- Online systems cares about the response time, which is measured in percentiles like p50, p90, p99, p999.
- Tail latencies (p999) are sometimes important as they are usually requests from users with a lot of data.
- Percentiles are often used in service level objectives (SLOs) and service level agreements (SLAs)
- Queuing delays often account for a large part of high percentiles. Since parallelism is limited in servers. Slow requests may cause *head-of-line blocking* and make subsequent requests slow.
- The latency from an end user request is the slowest of all the parallel calls. The more backend calls we make, the higher the chance that one of the requests were slow. This is known as *tail latency amplification*.

Coping with Load

- Scaling up (vertical scaling, with a more powerful machine) and scaling out (horizontal scaling, distributing the load across multiple machines, the *shared-nothing* architecture) are two popular approaches to cope with increasing load. Good architectures usually involves a mixture of both.
- Elastic systems can add computing resources when load increases automatically but it may have more surprises.

- Scaling up stateful data systems can be complex. For this reason, common wisdom is to use a single node until cost or availability requirements are no longer satisfied. Of course this may change in the future.
- The architecture for large scale systems is usually highly specific and built around its assumptions on which operations will be common or rare. There is no one-size-fits-all scalable architecture.

Maintainability

- Three design principles to minimize pain for maintenance.

Operability: Easier Life for Operations

- Operations are for keeping a software system running smoothly.
- Good operability means making routine tasks easy. Data systems can
    - Provide visibility into the runtime behavior
    - Provide support for automation and integration with standard tools
    - Avoid dependencies on individual machines
    - Provide good documentation
    - Provide good default behavior
    - Self-healing where appropriate
    - Minimize surprises

Simplicity: Managing Complexity

- Complexity slows down engineers working on the system and increases the cost of maintenance.
- Possible complexity symptoms: explosion of state space, tight coupling of modules, tangled dependencies, inconsistent naming and terminology, hacks for solving performance problems, special cases for workarounds, etc.
- We can remove accidental complexity, which is complexity not inherent in the business problem. This can be done through abstraction and hiding implementation details.

Evolvability: Making Changes Easy

- System requirements will change so we need to make making changes easy.
- Test-driven development and refactoring are tools for building software that is easier to change.
- Refactoring large data systems is different from refactoring a small local application (Agile); therefore, we use the term evolvability to refer to ease to make changes in a data system.

Summary

- Reliability means making systems work correctly even when faults occur. Faults can be from hardware, software, or humans.
- Scalability means having strategies to keep good performance even when load increases. The metrics to measure load and performance is system dependent.
- Maintainability is about making life easier for engineering and operation teams. Good abstraction reduces complexity and makes system easier to modify and adapt for new use cases.