Designing Data-Intensive Applications

Introduction

- Driving forces for developments in databases
    - Businesses need to be agile, test hypotheses cheaply, and respond quickly to new market insights by keeping **development cycles short** and **data models flexible**.
    - Huge volumes of data and traffic force companies to create **new tools** that enable efficiently **handle such scale**
    - **Free and open-source software** is preferred to commercial or bespoke in-house software environments
    - Parallelism is increasing
    - Distributed systems
    - Many **services** are now expected to be **highly available**; extended **downtime** due to outages or maintenance **is becoming increasingly unacceptable**

Data Intensive Applications – applications where the data is its primary challenge

- Quantity of data
- Complexity of data
- Speed at which the data is changing

PART I – Foundations of Data Systems

Chapter I – Reliable, Scalable, and Maintainable Applications

An application must meet various requirements to be useful:

- **Functional requirements** – What it should do?
- **Nonfunctional requirements** – security, reliability, compliance, scalability, compatibility, and maintainability.

RELIABILITY

- Tolerating hardware and software faults
- Human error

SCALABILITY

- Measuring load and performance
- Latency, percentiles, throughput

MAINTAINABILITY

- Operability, simplicity, and evolvability

A data-intensive application is typically built from standard building blocks that provide commonly needed functionality:

- **Databases** – store data
- **Caches** – speed up reads
- **Search Indexes** - Facilitate search data and filtering
- **Stream processing** – Send message to another process
- **Batch processing** – Periodically crunch a large amount of accumulated data

When you are designing a data system or service, you need to focus on three concerns:

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 software faults, and even human errors).

For software, the typical expectations are:

- The application performs the function that the user expected
- It can tolerate the user making mistakes or using the software in unexpected ways
- Its performance is good enough for the required use case, under the expected load and data volume
- The system prevents any unauthorized access and abuse

Systems that **anticipate faults** and can cope with them are called **fault-tolerant** or **resilient**. It is not possible to make a system tolerant of every possible kind of fault (things that can go wrong), but we can work on tolerating "certain types" of faults.

A fault is not the same as a failure

- Fault – one component of the system deviating from its spec
- Failure – the system as a whole stops providing the required service to the user

It is impossible to reduce the probability of a fault to zero, therefore it is usually best to design **fault-tolerance** mechanisms that prevent faults from causing failures.

Generally, we prefer **tolerating faults over preventing faults**, there are cases where prevention is better than the cure (e.g., because no cure exists).

Types of errors and techniques to prevent them:

- Hardware Errors
  - Redundancy of hardware components
  - Software fault-tolerance techniques
- Software Errors

    - Carefully thinking about assumptions and interactions in the system
        - Testing
        - Process isolation
        - Allowing processes to crash and restart
        - Measuring, monitoring, and analyzing system behavior in production.
    - The system constantly check itself while it is running and raise an alert if a discrepancy is found
- Human Errors
    - Design systems in a way that minimizes opportunities for an error
    - Decouple the places where people make the most mistakes from places where they can cause failures
    - Test thoroughly at all levels, from unit testing to whole-system integration tests and manual tests.
    - Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure
    - Set up detailed and clear monitoring, such as performance metrics and error rates – Telemetry
    - Implement good management practices and training

Scalability

As the system grows (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth.

Scalability is the term we use to describe a system's ability to **cope with increased load**. Discussing scalability means considering questions like "If the system grows in a particular way, what are our **options for coping with the growth**?" and "How can we **add computing resources** to handle additional load?"

To think in scalability:

1. Describe the load on your system. **Load** can be described with a few numbers which we call **load parameters**.
2. Describe Performance. Investigate what happens when the load increases
    a. When you increase a load parameter and **keep the system resources** (CPU, memory, network bandwidth, etc.) **unchanged**, how is the **performance** of your system affected?
    b. When you increase a load parameter, how much do you need to **increase the resources** if you want **to keep performance unchanged**?

    Performance metrics
    - Service's response time – the time between a client sending a request and receiving a response.
    - Latency – duration that a request is waiting to be handled
    - Percentiles – list your response times and sorted them from fastest to slowest

3. Select the approach for coping with load
    a. **Scaling up** – vertical scaling – moving to **a more powerful machine**
    b. **Scaling out** – horizontal scaling – **distributing** the load across multiple smaller machines. It is also known as **sharing-nothing architecture**
    c. **Hybrid** – pragmatic mixture of approaches

    Type of systems:
    - **Elastic** – Add computer resources automatically when they detect a load increase. Useful if the load is highly unpredictable.
    - **Manually** – a human analyzes the capacity and decides to add more machines to the system. It is simpler and may have fewer operational surprises.

Maintainability

Over the time, many different people work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should be able to work on it productively.

The majority of the cost of software is not in its initial development, but in its ongoing maintenance – fixing bugs, keeping its system operational, investigating failures, adapting it to new platforms, modifying it for new uses cases, repaying technical debt, and adding new features.

When building software, please pay special attention to three design principles to minimize pain during maintenance:

- **Operability** – Make it easy for operations teams to keep the system running smoothly.
    - Good **operability** means making **routine tasks easy**, allowing the operations team to focus on their efforts on high-value activities.
    - Data systems can make routine tasks easy by
        - Providing **visibility** to **runtime behavior** and **internals of the system**, with **good monitoring**
        - Providing good **support** for **automation** and **integration** with standard tools
        - **Avoiding dependency** in individual machines
        - Providing **good documentation** and an easy-to-understand operational model
        - Providing good default behavior
        - Self-healing where appropriate
        - Exhibiting predictable behavior
- **Simplicity** – Make it easier for new engineers to understand the system, by removing as much complexity as possible from the system.
    - In **complex software**, there is also **a great risk** of introducing **bugs** when making a change. Reducing complexity greatly improves the maintainability of the software, and thus simplicity should be a key goal for the systems we build.
    - **Abstraction** helps us to remove **accidental complexity**

        - **Accidental complexity –** if the complexity is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation.
- **Evolvability** – Make it easier for engineers to make changes to the system in the future, adapting for unanticipated use cases as requirements change. Also known as extensibility, modifiability, or plasticity.
    - **Agile Working patterns** provide a framework for adapting to change
    - The ease with which you can modify a data system, and adapt it to changing requirements, is closely linked to its simplicity and its abstractions: **simple and easy-to-understand systems are usually easier to modify than complex ones**.

Glossary

- **Transmission Control Protocol (TCP)** – it is a transport protocol that is used on top of IP to ensure reliable transmission of packets.
- **Free and Open-Source Software (FOSS) –** Software that people can modify and share because its design is publicly accessible
- **Proprietary Software** – Closed-software, software as a service
- **Application Programming Interface (API)** - a software intermediary that allows two applications to talk to each other
- **Mean Time to Failure (MTTF)** – It is a maintenance metric that measures the average amount of time a non-repairable asset operates before it fails.
- **Redundant Array of Independent Disks (RAID)** - data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both.
- **Telemetry** - the process of recording and transmitting the readings of an instrument.
- **Service Level Objectives (SLOs)** - key element of a service-level agreement between a service provider and a customer. SLOs are agreed upon as a means of measuring the performance of the Service Provider and are outlined as a way of avoiding disputes between the two parties based on misunderstanding.
- **Service Level Agreements (SLA)** – It is a commitment between a service provider and a client. Particular aspects of the service – quality, availability, responsibilities – are agreed between the service provider and the service user.