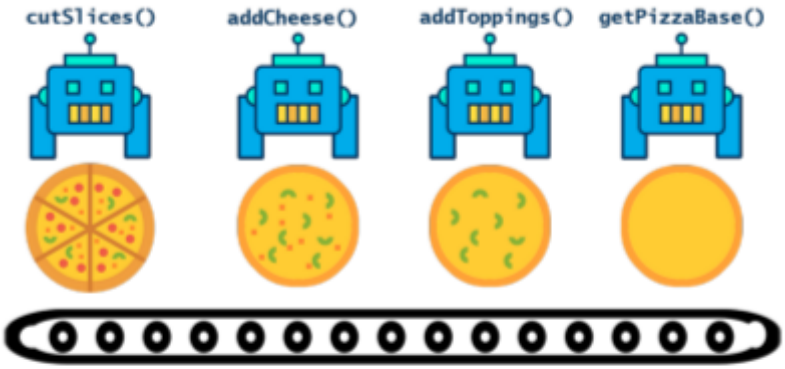
Introduction

- Often, in a pizza outlet, we observe that there are separate sections specific to the various tasks involved in preparing a pizza which may include:
- getting the pizza base
- adding the topping
- adding cheese
- cutting slices.
- In this example, the pizza outlet can be considered a distributed system consisting of separate sub-systems (e.g., getPizzaBase(), addToppings(), etc.) as shown in the image below.



- Figure 1.1: Pizza outlet - a distributed system

`Distributed System also known as distributed computing and distributed databases can be defined as a level of abstraction in which multiple nodes (hardware or software components) coordinate their actions by communicating with each other on a network and still act as a single unit.`

Evolution of Distributed System

- Over the last few decades, the need to make services available to a wide range of people in real time led to the growth of distributed systems. The evolution was focused around availability, reliability, and reusability of the system. This has also led to the

growth of standard communication interfaces between systems.

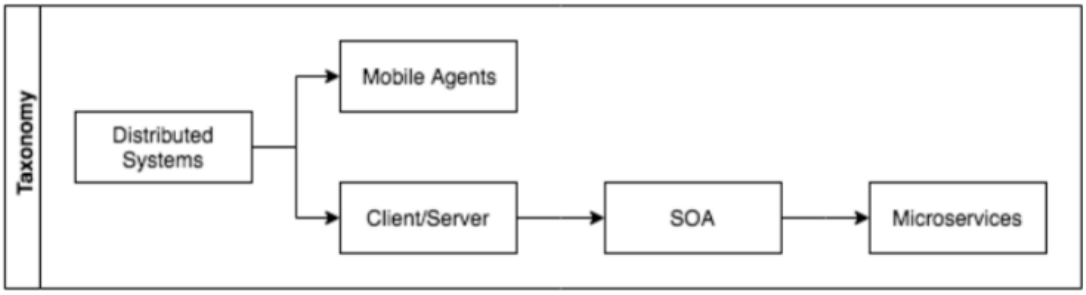


Figure 1.2: Distributed system Taxonomy

- Paradigms such as Mobile Agents technology came into picture which was geared towards overcoming slow networks by enabling the capability to migrate software and data from one computer to another autonomously.
- Due to the growth of available and fast network connections and lightweight communication protocols (APIs), the client-server model gained more attention than its contemporaries. After that, Service Oriented Architecture (SOA) came into the picture to provide integration points between different organizations which relied on simple remote procedure calls such as Simple Object Access Protocol (SOAP).
- However, the need for more lightweight and modular systems led to the growth of Microservices

---

Types of Distributed system

Client server:

- Distributed system architecture consisted of a server as a shared resource like a printer, database, or a web server. It used to have multiple clients, such as user behind the computers that decided to use the shared resource, how to use and display it, change data and send it back to the server.
- Code repositories like git is a good example where the intelligence is placed on the developers committing the change to the code.

Three-tier

- In this type of architecture, the information about the client is stored in a middle tier rather than on the client to simplify application deployment.
- This middle tier can be called as agent that receives requests from clients, that could be stateless, processes the data and then forwards it on to the servers. This architecture model is most common for web application.

Multi-tier

- This was first created by enterprise web services for servers which contain business logic and interacts with both the data tiers and presentation tiers.

Peer-to-peer

- There are no special servers which need to the intelligent work in this architecture. The decision making and responsibilities of all the servers are split among the machines involved and each could take on client or server roles. Example of such a system is Block chain

---

Common application of distributed system

Microservices

- It is an architectural pattern where an application is structured as a collection of small autonomous services modeled around a business domain and each of those services comprises of a specific business capability. The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications.

```
For instance, designing a streaming platform such as Spotifymay involve creating
separate web services for different sub-domains of the business:
1.user-identity management
2.user-playback history
3.audio content management
4.and so forth.
```

Note:Monolithic architecture suits simple light-weight applications where all the business logic can be bundled together. On the other hand, microservices architecture is better suited for complex and evolving applications.

Distributed Caching

- With the growth of the internet, it was not possible to cache the information for quick look-up on a single host.
- The usage of distributed systems such as Distributed Hash Tables(DHT) helped us solve this problem by using efficient hashing(e.g. Consisent Hashing Webmaster: Include link to Consistent Hashing) mechanisms and communication protocols(Chord).

Distributed Data Storage

- The growth of NoSQL over the past decade led the innovation in the field of distributed data storage systems, which can run over large clusters of commodity hardware. These systems can be catered towards a range of query patterns: simple key-value data stores

like Amazon DynamoDB and graph-based stores like Neo4j to time-series database such as InFluxDB.

Exercise

> **There is a contest to vote for your favorite TV series. Ten popular TV series (Friends, GOT, The Big Bang Theory, Breaking Bad and so forth) are part of this contest. The contest will last for a few hours. We need to build an application to support this voting contest**

```
The major requirements which we need to support in this application are:
1.Users should be able to use mobile and desktop platforms to cast their vote.
2.Administrators of the voting application should be able to access the current votes count.
3.Since the voting will last for only a couple of hours, we expect the frequency of votes(TPS) to be significantly high (i.e., tens of thousands).
```

Ans:
Brute Force/Naive Approach:

```
- Build an application where each server/host maintains the count of votes.
- Followed by, the total votes count is obtained by fetching the votes count from individual servers and summing them up.
- However, a major issue with this approach is that in case a host goes down, then the votes count stored on that particular node will be lost as well.
```

Optimal Approach

- The voting application can be broken down into two separate components:
    1. Online component (aka user plane, data plane)
    2. The Offline component (aka control plane).
- The online part helps with interfacing with voters on real-time. For example, users can cast their vote and get a confirmation when the vote is persisted onto the database securely.
- On contrary, control plane supervises the background operations for the system (e.g. tallying up the votes).

Online Component/Data Plane:

```
the design of the data plane of this voting application is shown. The voters can use the mobile and desktop platforms to cast their vote by placing HTTP requests. Those requests will be forwarded to load balancers using DNS lookups.
```

```
The load balancer will use scheduling algorithms to direct the traffic to
application servers, which will save the votes in the data store.
```

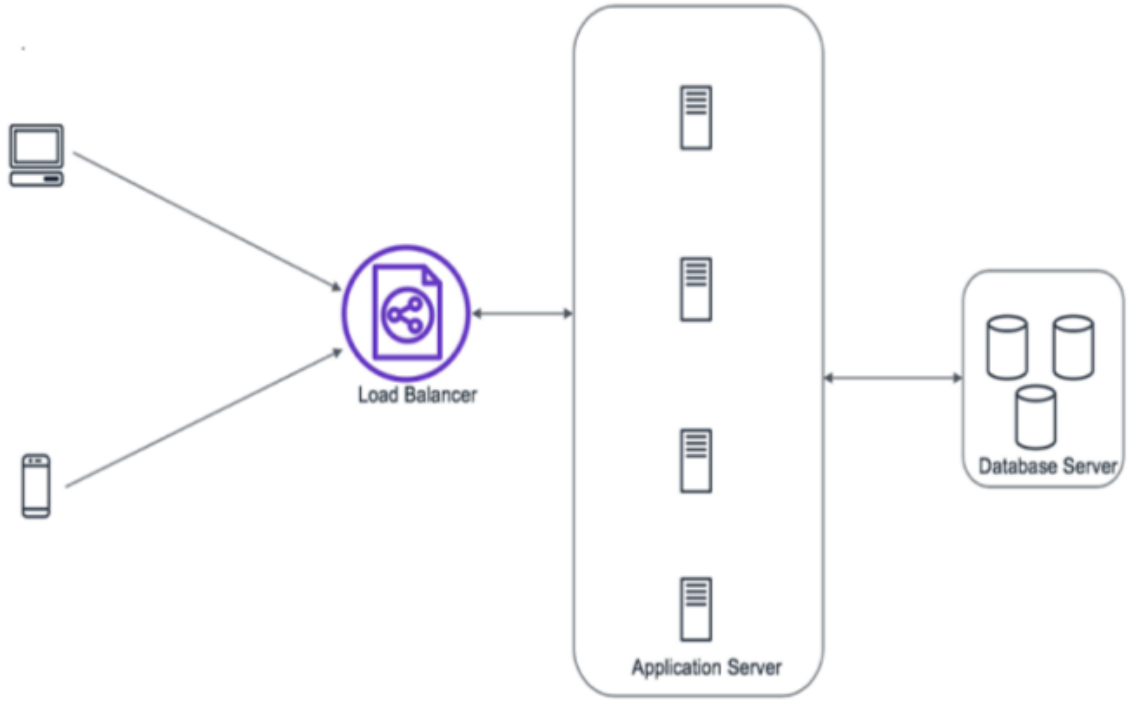


Figure 1.3: Voting Application: Online component

```
Data Model
1.You may use any aggregate-oriented NoSQL data store (key-value store such as
Riak, Amazon DynamoDB) as they are cheaper and scalable compared to traditional
relational databases.
2.One way of modeling the data can be to have separate vote counts for different
TV series (using TV series name as key), which gets updated whenever a vote gets
cast.
3.However, a skewed distribution of votes (as shown in diagram) can lead to a
scenario where the partition on the disk storing the TV series having higher
votes can get overwhelmed by the writes.
```

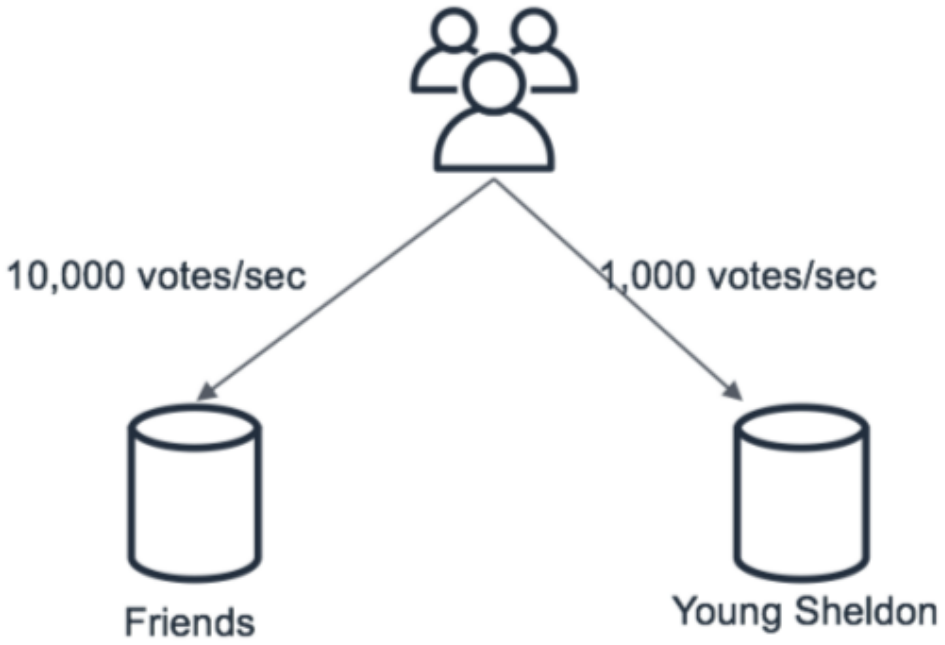


Figure 1.4: Skewed distribution of casted votes

The issue related to skewed distribution can be solved using write-sharding by attaching a random number to the TV series name, as shown in Fig below. Using this approach, the writes for a particular TV series name gets distributed across multiple shards (more details in upcoming chapters)

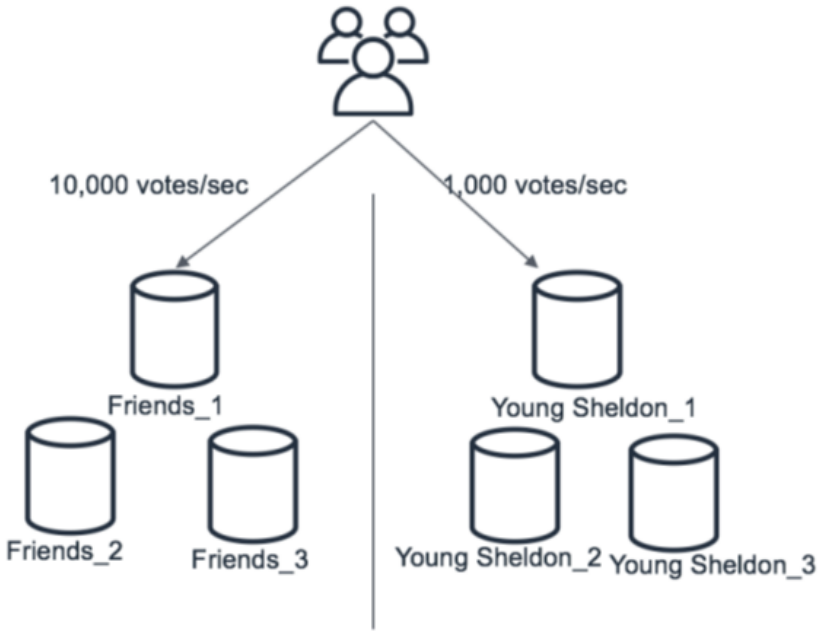


Figure 1.5: Write sharding to solve skewed distribution problem

Offline components

```
- In the offline component, you can use a cron job which triggers a map-reduce operation that scans the sharded records, counts votes of all the TV series and persists the final votes count in a separate data store as shown in Fig 1.6
- We will discuss more details about map-reduce and its applications in future. After that, the administrator's request to fetch the current votes count is served by the database table which gets populated by the offline operation (Votes Count table in Fig 1.6).
```

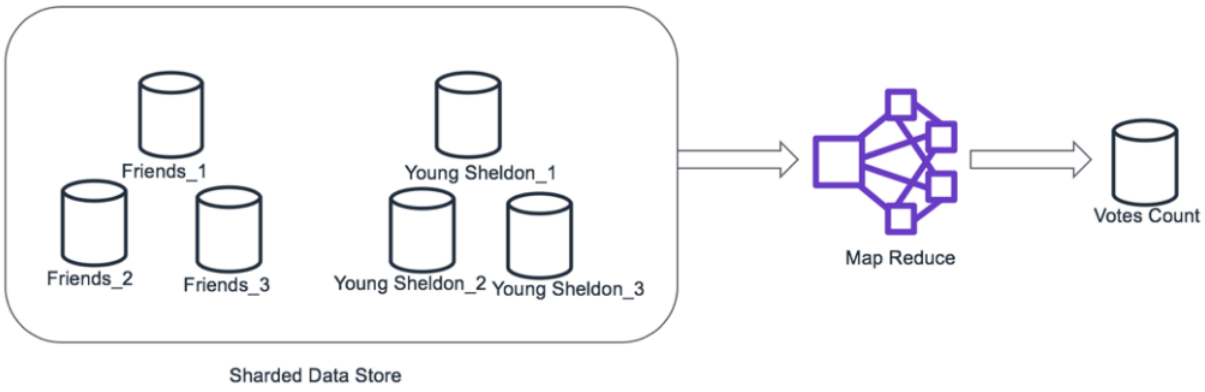


Figure 1.6: Offline operation to compute votes count

Important Things to take care while building a Dsystem...

Caveats uncovered

- You can use the voting application to gain in-depth insights into some of the caveats, which can help software development teams in designing more reliable, robust, and scalable systems.
- **Business Requirements**
- System Design depends a lot on the business domain using which we can make the tradeoff between reliability(consistency) and performance(availability).
- For instance, building a financial system requires data to be consistent even though it comes at the cost of reduced performance.
- However, in the voting application, we may prefer availability over consistency as it's important that users can cast votes, even if the administrators get the exact vote counts after a minimal delay.

Data Modeling

- The choice of the data store and the data schema depends a lot on users requirements and query patterns.
- For social media applications, we can use graph-based data stores like Neo4j, applications similar to Fitbit, which are related to time-series data may use databases like InfluxDB.
- In the voting app above, the data is stored in a key-value data-store by using the write-sharding mechanism to tackle the problem of skewed votes distribution.

Node Failure

- Failure of nodes is an unavoidable phenomenon, and developers should take this into account while building a robust system.
- Under the purview of distributed systems, we employ techniques such as sharding and replication to minimize the impact of such node failures.
- In the voting application, we account for the fact that the hosts used in the application servers can go down, and so the data is persisted in a separate data store rather than on individual servers(naïve approach).

Network Failure

- Similar to node failure, it's challenging to prevent network failure, so developers should account for error-handling on networking errors.
- One way of handling such error-scenarios is the retry policy, which varies a lot depending on business requirements (e.g. latency and availability SLA).
- For instance, we can have a retry mechanism in the voting application to persist the information in the data store to account for issues related to network failure.

Network Security

- Often, complacency related to network security leads to the system being vulnerable to malicious users and programs that keeps on adapting to the existing security measures.
- Using secure protocols such as HTTPS in the voting application will ensure that malicious users don't cast the votes.

Infrastructure Cost

- We should be mindful of the infrastructure cost required for running the system. It's quite common in the industry to see developers being blind-sided about the cost factor in their designs.
- The usage of NoSQL data store in the voting application is an approach to minimize the infrastructure cost as the NoSQL data stores are efficient at running on large clusters of commodity hardware

Communication Interface

- It's imperative to use best practices while designing APIs for the web-services using lightweight communication protocols.
- This allows the creation of secure and extensible integration point for those services. I
- n the voting application, using light-weight REST APIs in the data plane provide ease of access from different platforms such as desktop and mobile.