Distributed System Study Notes (from Udemy)

Load balancing

Distributed Message Broker

Distributed Storage

Load balancing

Load balancer

- Allows in front of servers
- Can scale up and down the service without exposing implementation details to the clients
- Allows server’s health monitoring, which keeps the system reliable and highly available
- Types: can use hardware or software load balancers (HAProxy, Nginx)

Load balancing strategy

- Round Robin: Uniformed/Weighted; sent to server one by one and start over when one round finishes
- Source IP Hash (a user will connect to one backend server through one session)

Example scenarios: open session, maintaining online shopping cart state;

Local server caching, improving performance by

preloading data, caching data locally

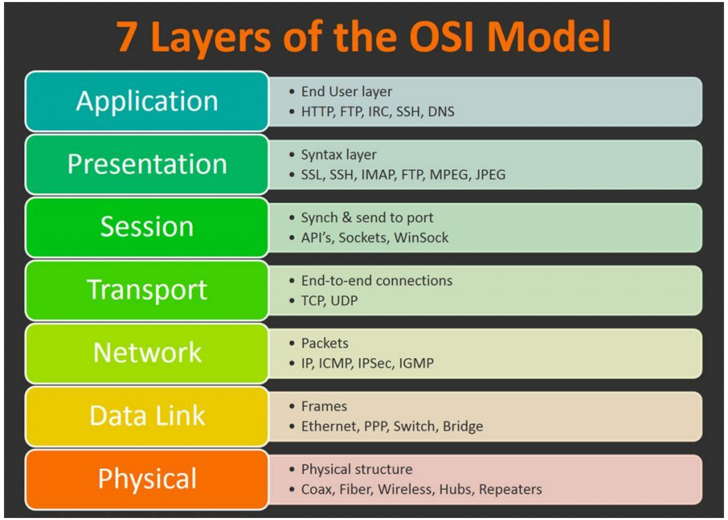
Hash(source.IP) = server #3

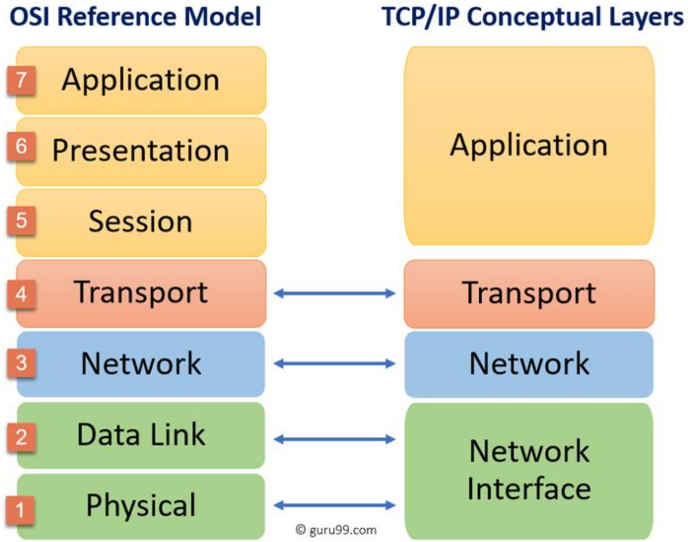
Load balancing strategy

- Least Connection
  - Servers has fewer connections will be assigned more tasks
- Weighted Response Time
  - Server has fewer responses time will be assigned more tasks
- Usage pattern
  - Server will send more complex data for metrics, such as
    memory usage, disk operations, inbound or outbound network traffic (bytes)

2 networking layer modes for Load Balancing

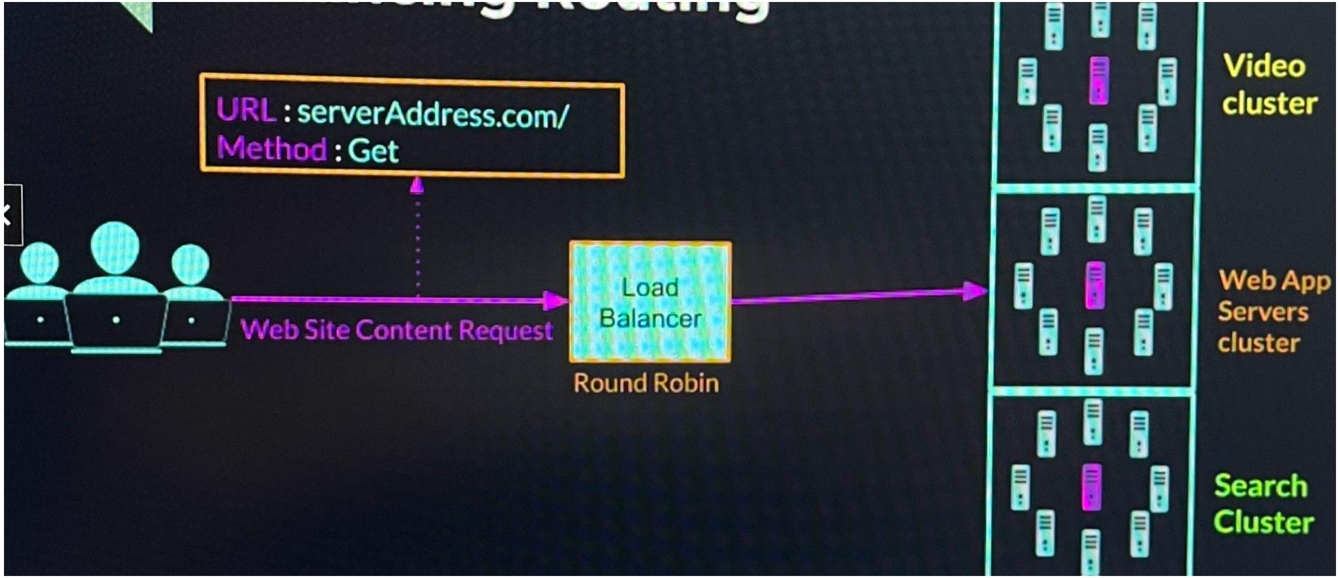
- Layer 4 (Transport) load balancing is best for:
  - Simple Load Balancing, Lowest load balancing overhead
- Layer 7 (Application) load balancing is best for: (def: routing based on http header)
  - Most system for better control over incoming traffic routing to our system





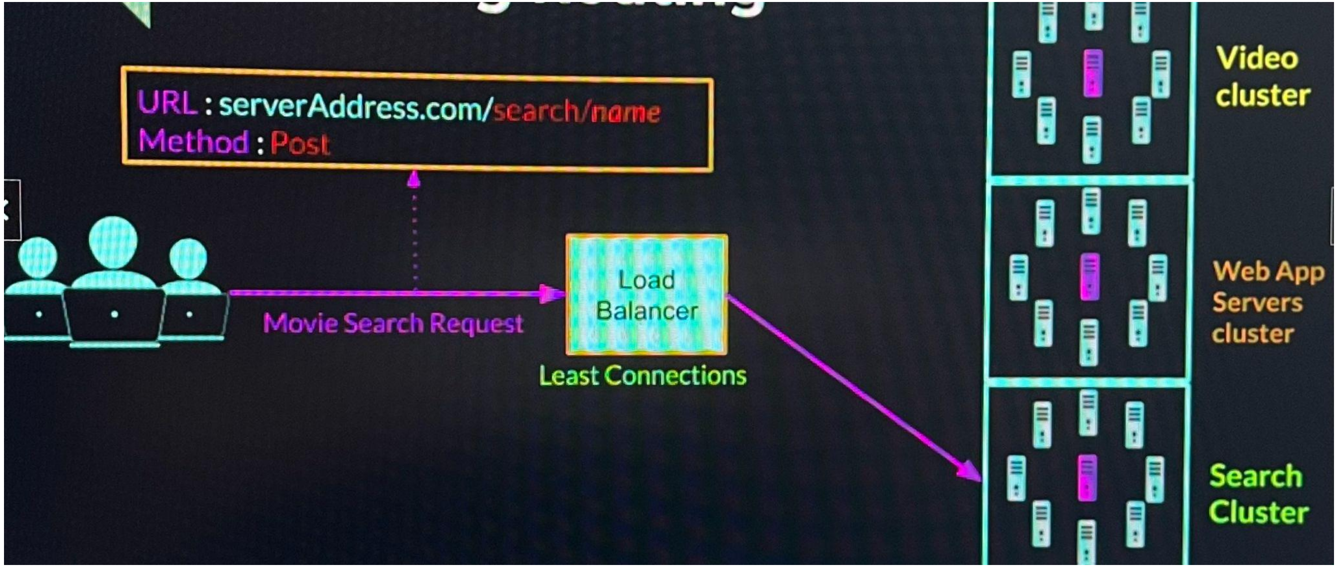
Layer 7 example (video streaming service)

General web requests



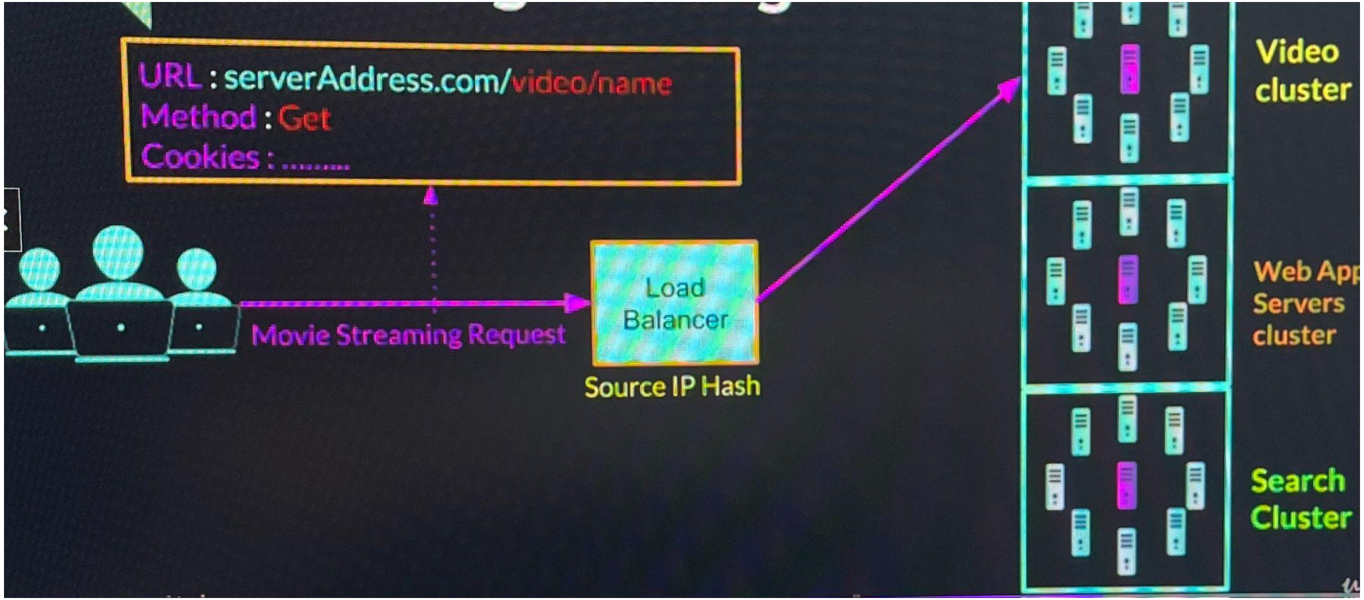
Layer 7 example (video streaming service)

Search requests



Layer 7 example (video streaming service)

Video streaming requests



Sample HAProxy (similar as nginx) config file

```
global
        maxconn 508
defaults
        mode http
        timeout connect 10s
        timeout client 50s
        timeout server 50s
frontend http-in
        bind *: 80
        acl even path_end -1 /even
        acl odd path end -1 /odd
        use_ backend even_application_nodes if even
        use_backend odd_application_nodes if odd
```

Advanced Routing (ACLs)

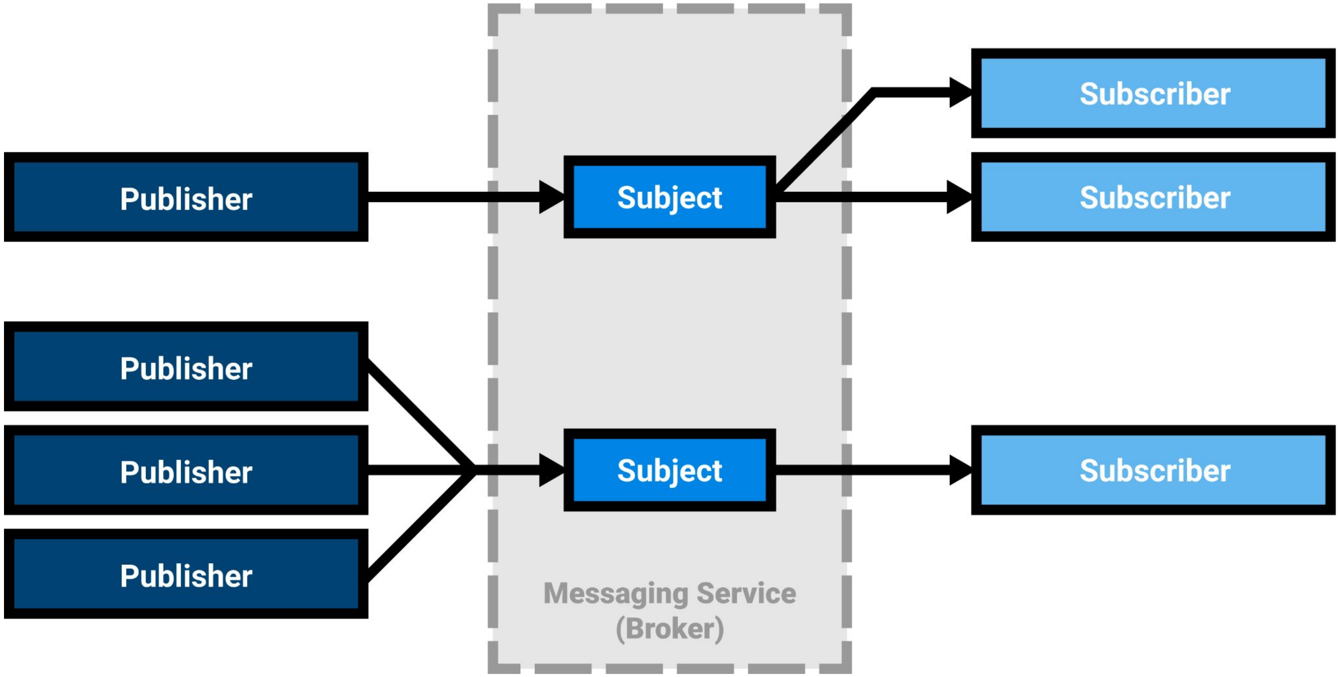
```
backend odd_application_nodes
        balance roundrobin
        option httpchk GET /status
        server server01 localhost:9081 check inter 5s
        server servers3 localhost:9683 check inter 5s
backend ever_application_nodes
        balance roundrobin
        option httpchk GET /status
        server server02 localhost:9002 check inter 5s
listen stats_page
        bind *83
        stats enable
        stats uri /
```

Health check every 5 s

Distributed Message Broker

- Intermediary software (**middleware**) that passes messages between senders and receivers
- May provide additional capabilities such as data transformation, validation, queueing and routing
  - Msg brokers allows event driven, **asynchronous** network communication (vs direct synchronous communication)
- Full decoupling between senders and receivers
  - Msg brokers have queuing abilities that receivers do not have to present when msg are sent
  - They have powerful queuing such that whenever receivers scale horizontally, msgs will not lost
  - They also serve as **load balancer**
- Open Source: RabbitMQ, ActiveMQ, Kafka

Publish to and Subscribe msgs from Message Broker



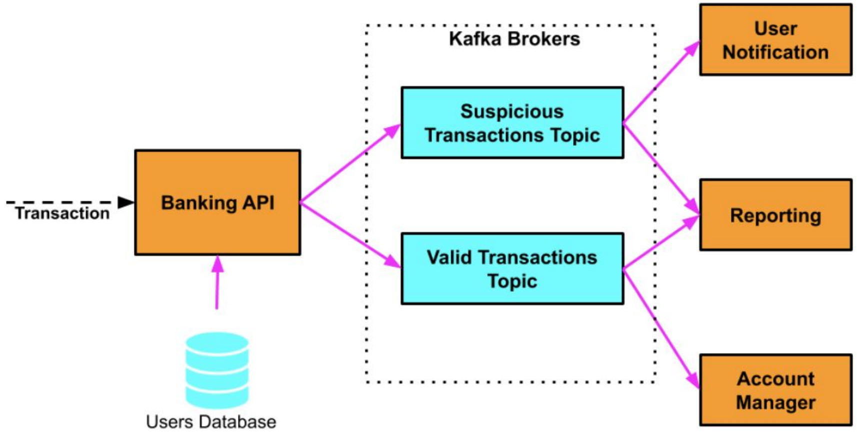
Publisher will need to publish msgs to message broker

Then subscriber will need to pull msgs from message broker

Example Distributed Banking system

Once a transaction request comes in, the Banking API validate user info from the Users DB and use Kafka message brokers to distribute downstream task.

The message broker has two topics: suspicious/valid. If the message is flagged as suspicious, then it calls 2 api. If the valid, it calls the other 2 api.

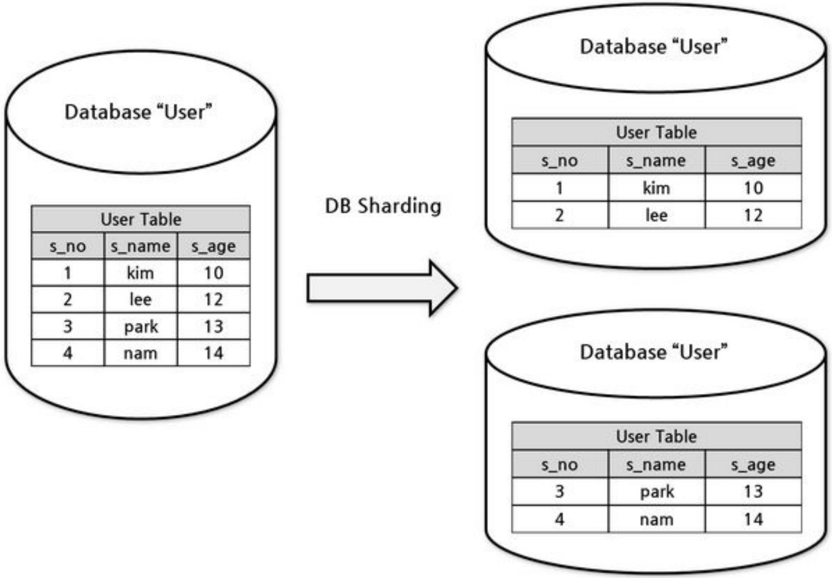


Distributed Storage

- Typically an application can store data to 1) file system or 2) DB
- Two Types: Relational database vs Non Relational database
- Distributed Database can achieve: 1) Availability 2) Scalability 3) Fault Tolerance
- Most Distributed Database use both **sharding and replication** to achieve those three above (1) Availability 2) Scalability 3) Fault Tolerance)

Database Sharding

- Data Partition to different DB
- Sharding Strategy:
    - Hash based sharding
        - Key: userID; N of DB = 3
        - Hash(key) = key mod N
    - Range based sharding (more common)
        - Ordered by name
        - DB 1: name[A-H] …
- Concurrency control becomes complexed



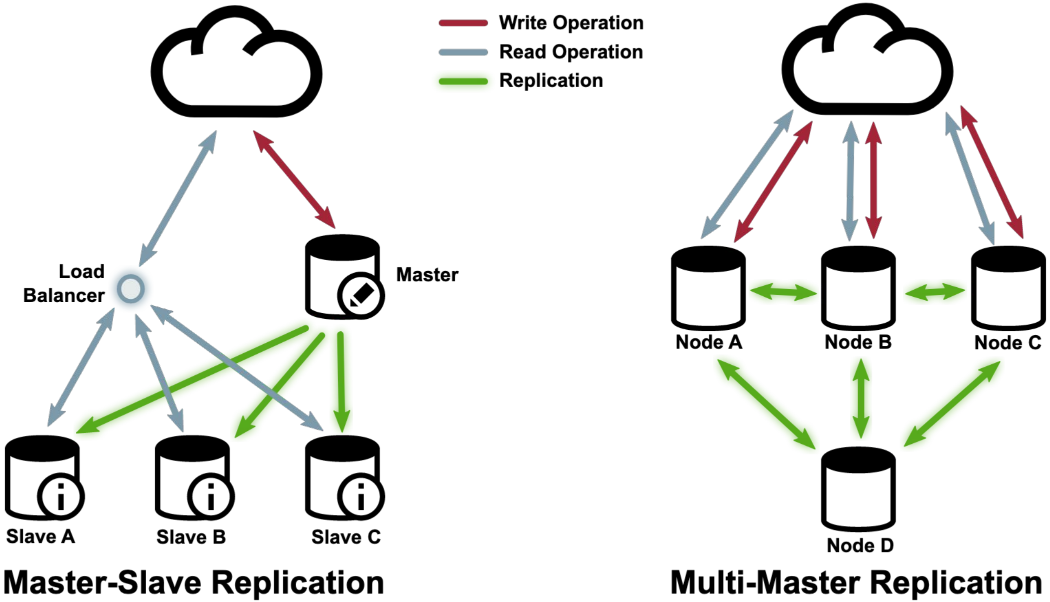
Database Replication

- Creating identical copy of data into different machines
- Motivation
    - High availability
    - Fault tolerance
    - Scalability and performance (High throughput and more concurrent R/W)
- Design choice: Eventual Consistency vs Strict Consistency
    - Strict Consistency: user account, numbers of items in a story inventory, seats in theater
    - Eventual Consistency: posts/update to social media; analytics for product ratings and numbers of views

Master-Slave & Master-Master

master - slave: write operations go to master, read operations go to slave

master - master: each node takes write and read operation

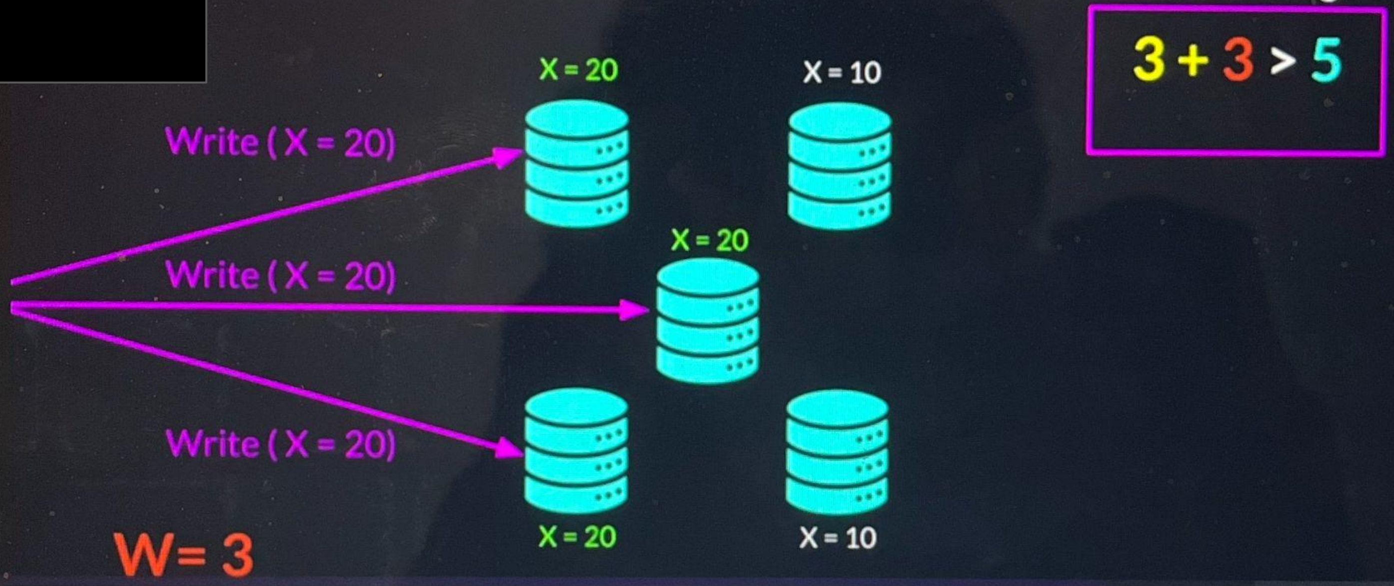


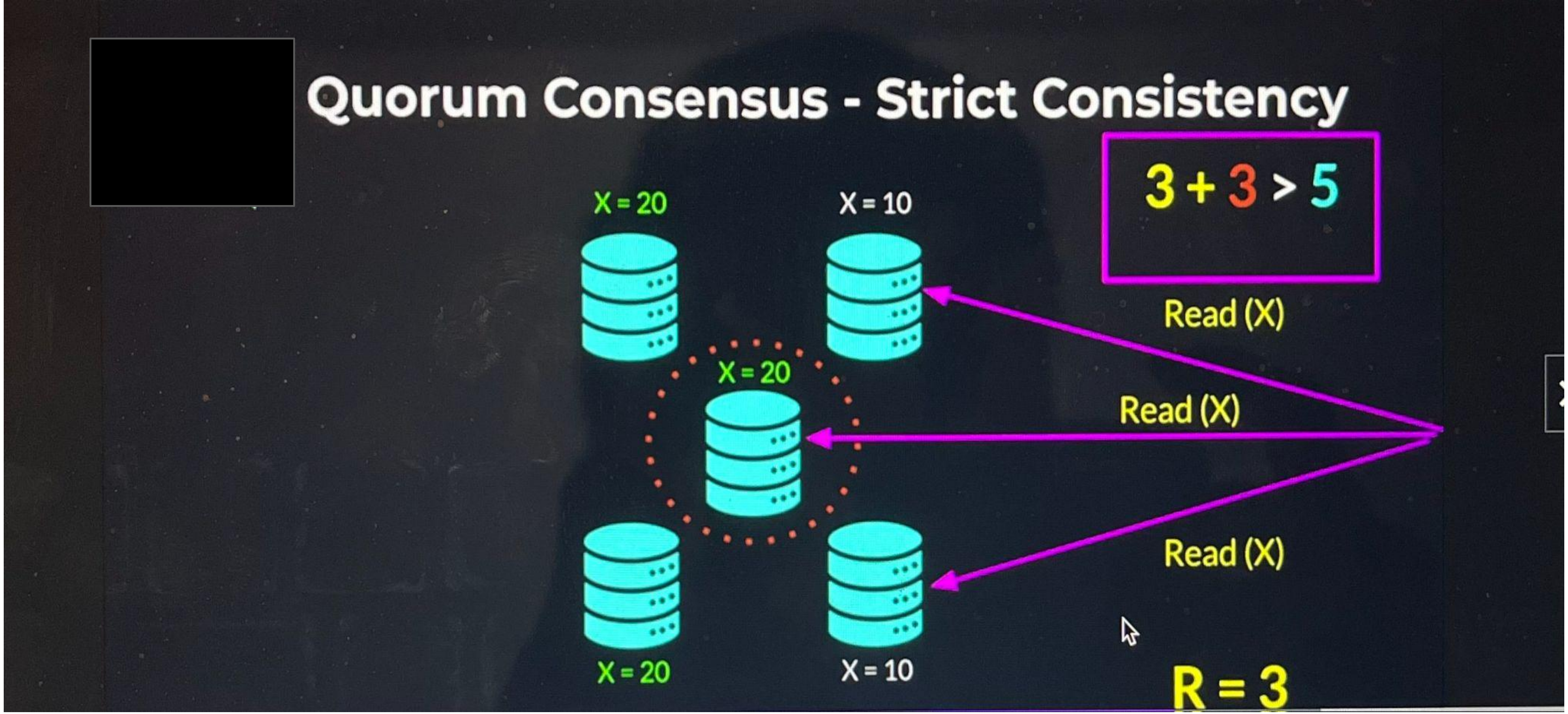
Quorum Consensus - Record Version

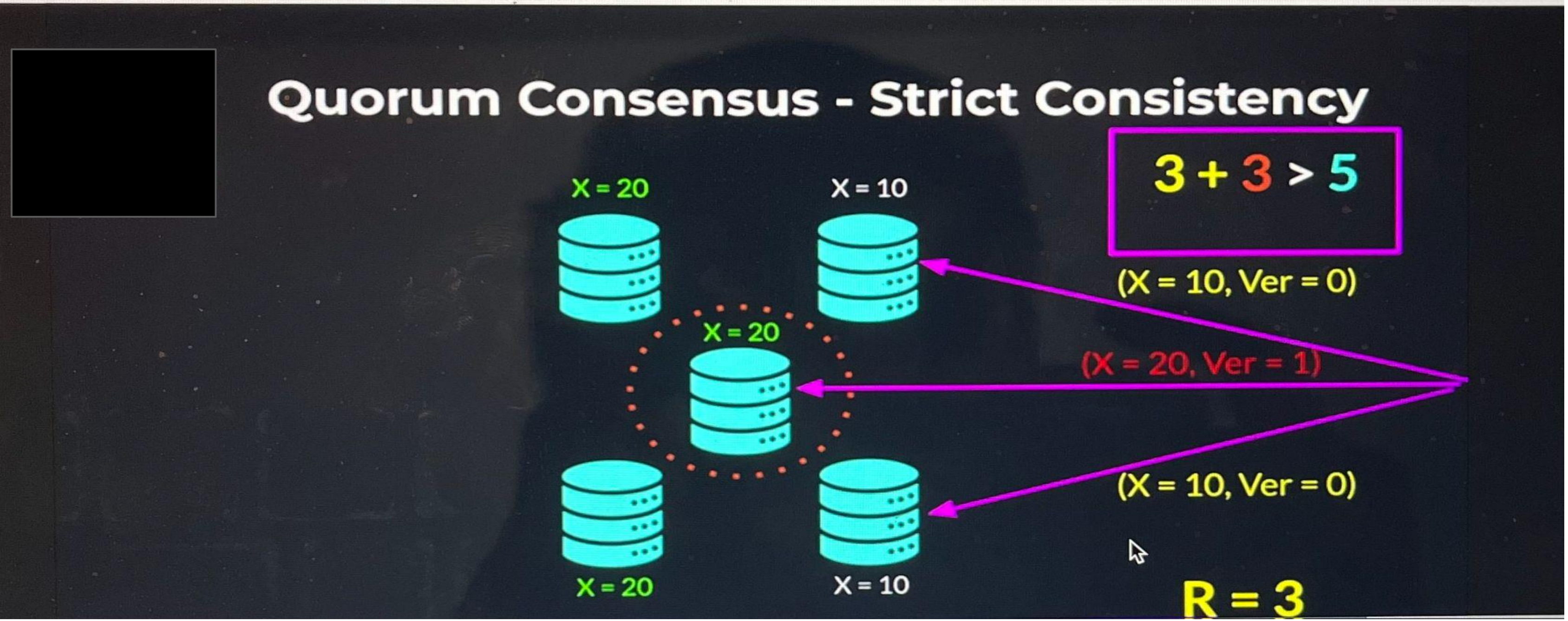
- The strategy used in strict consistency: set min numbers of read and write to guarantee strict consistency
- Every update to a record increments the version number
  - Old record: key1, data1, 1; new record: key1, data2, 2
- min Read + min Write > Numbers of node, guarantee to have strict consistency;

  Else will have eventual consistency

Quorum Consensus - Strict Consistency




Quorum Consensus - Strict Consistency
3 + 3 > 5
X = 20
X = 10
X = 20
Read (X)
Read (X)
Read (X)
X = 20
X = 10
R = 3


Quorum Consensus - Strict Consistency
3 + 3 > 5
X = 20
X = 10
(X = 10, Ver = 0)
X = 20
(X = 20, Ver = 1)
(X = 10, Ver = 0)
X = 20
X = 10
R = 3