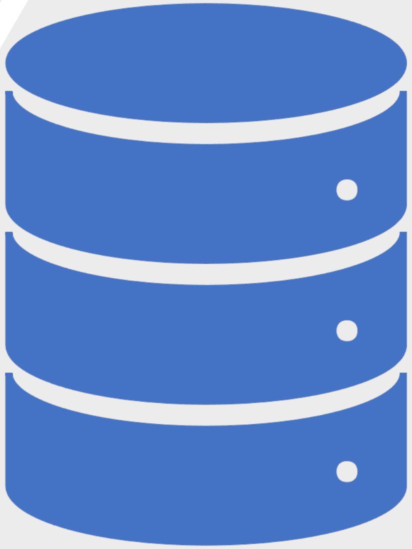
Database Scalability



What we will learn today ?

1. Recap

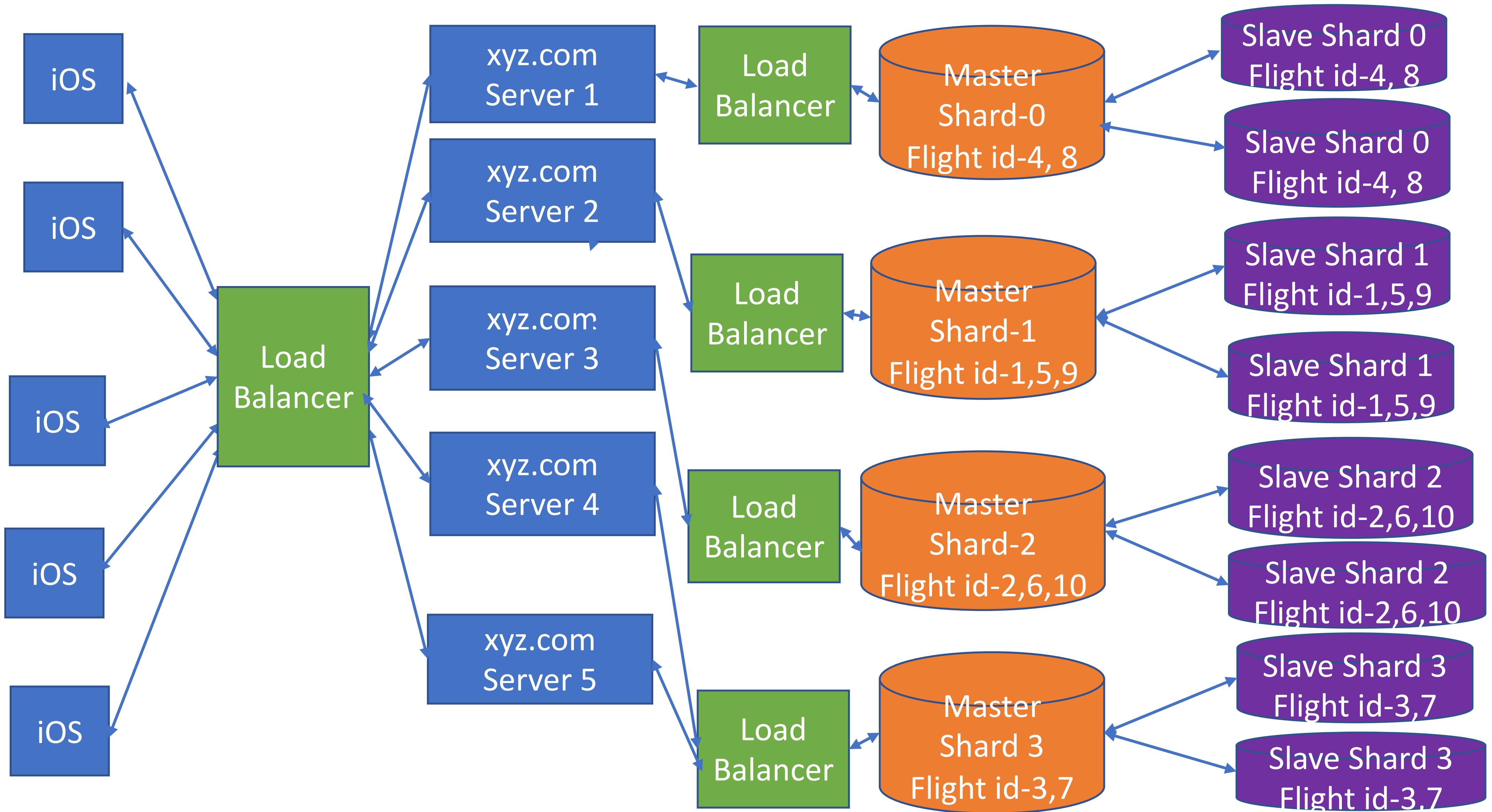
2. Database Replication Models

3. 2 Phase Commit

4. CAP

1. Recap

1. Recap: Database Replication



1. Recap: Database Replication

1. Now that we have **3 dB replicas for each Shard** what happens if we update one shard Replica ?

2. How do we make sure that **updates are consistent across** shard replicas ?

3. How do we make sure **read/write operations are in sync** across all the 3 shard Replicas ?

4. How do the **dB replicas communicate** among themselves ?

5. We have few **dB replication models(Master-Slave, Master-Master)** to to fix this problem.

2. Database Replication Models

2. Database Replication Models

1. Single Primary Replication Model(Master-Slave)
2. Multi Primary Replication Model(Master-Master)

2.1 Single Primary Replication Model

1. This model helps in **distributing the read** request loads across dBs.

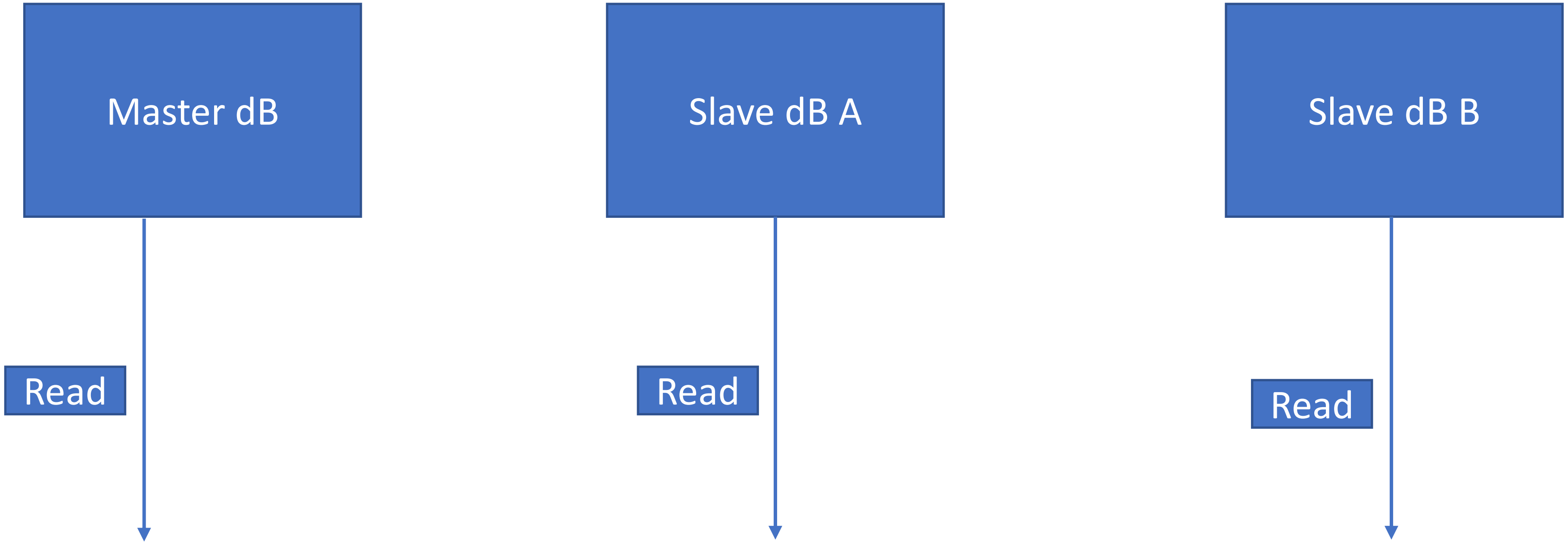
2. In this model we have **two types of dBs**- A **Primary dB(Master)** and **Secondary dBs(Slave).**

3.**Primary dB(Master):**We can perform **read/write** operation on ths dB.

4. **Secondary dBs(Slaves):** We can perform **ONLY read** operation on this dB.

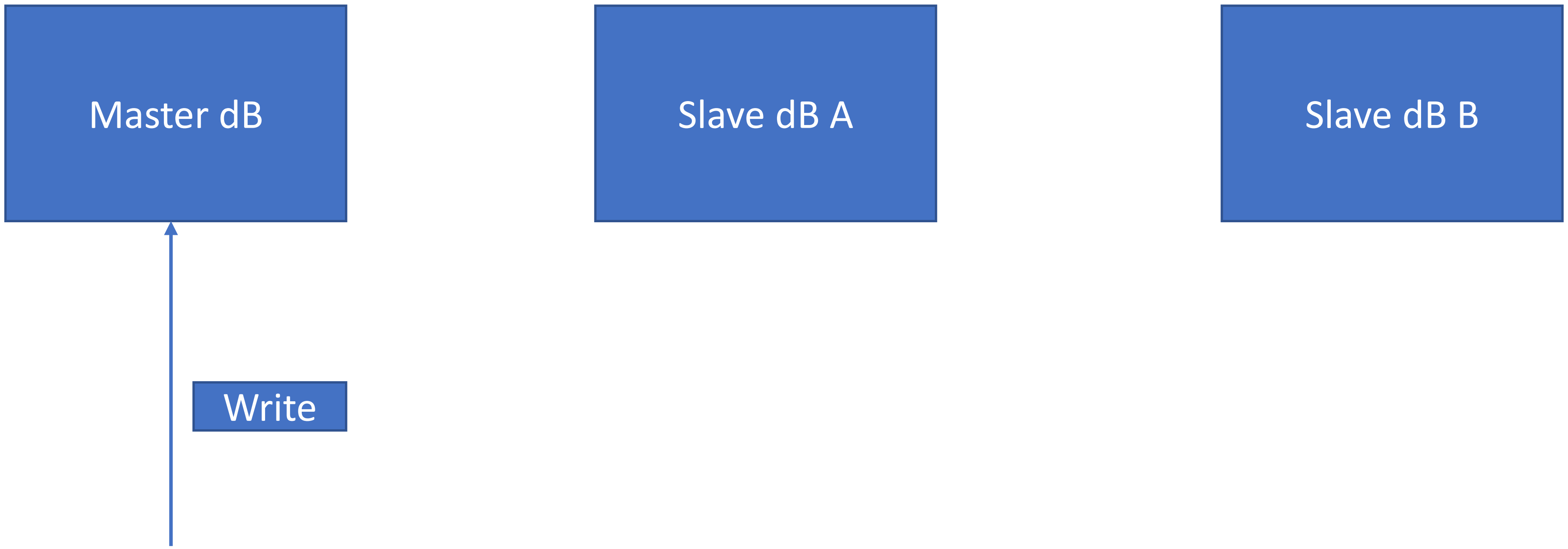
2.1 Single Primary Replication Model

1. Every **Read request** can be redirected to any of the 3 dB Replica.
2. We use a load balancer to distribute read request load evenly.



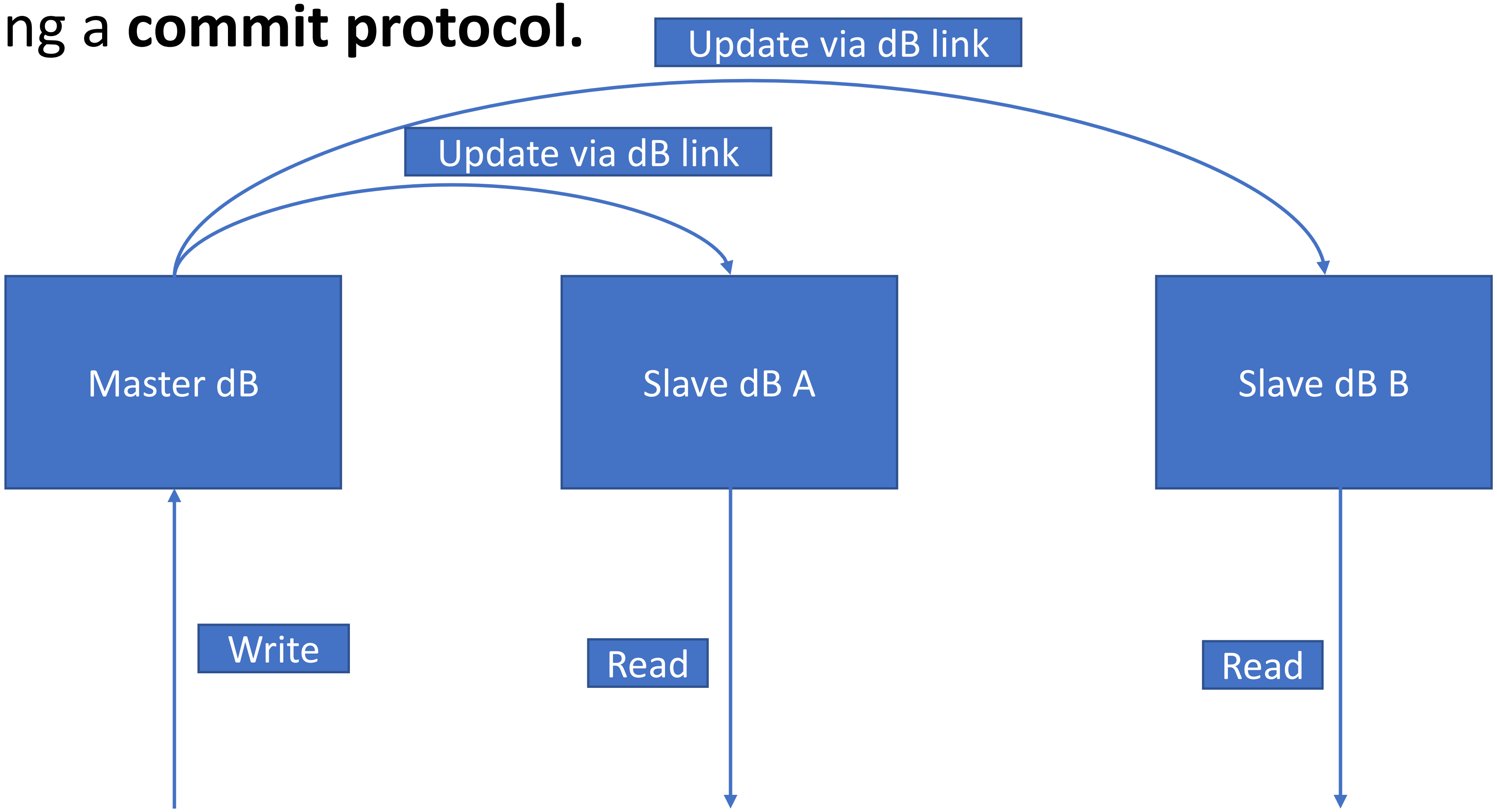
2.1 Single Primary Replication Model

What happens if a write to master(Primary) dB is made ?



2.1 Single Primary Replication Model

Master(Primary) dB performs a sync/async update to its slaves via dB links using a **commit protocol.**



2.1 Concerns with Single Primary Replication Model

1. We might run into **race condition:** If we **write** some data into **primary(master)** dB and then try to **read** data from a **secondary(slave)** dB before the **primary(master)** dB updates its changes to the **secondary(slave)** dB.

2. This is a good model for a **news website** where **Reads > Writes.**

3. If the **primary(master) dB fails**, we cant write to the dB. SPOF.

4. How do we **distribute the write requests** across dBs ? (write heavy)

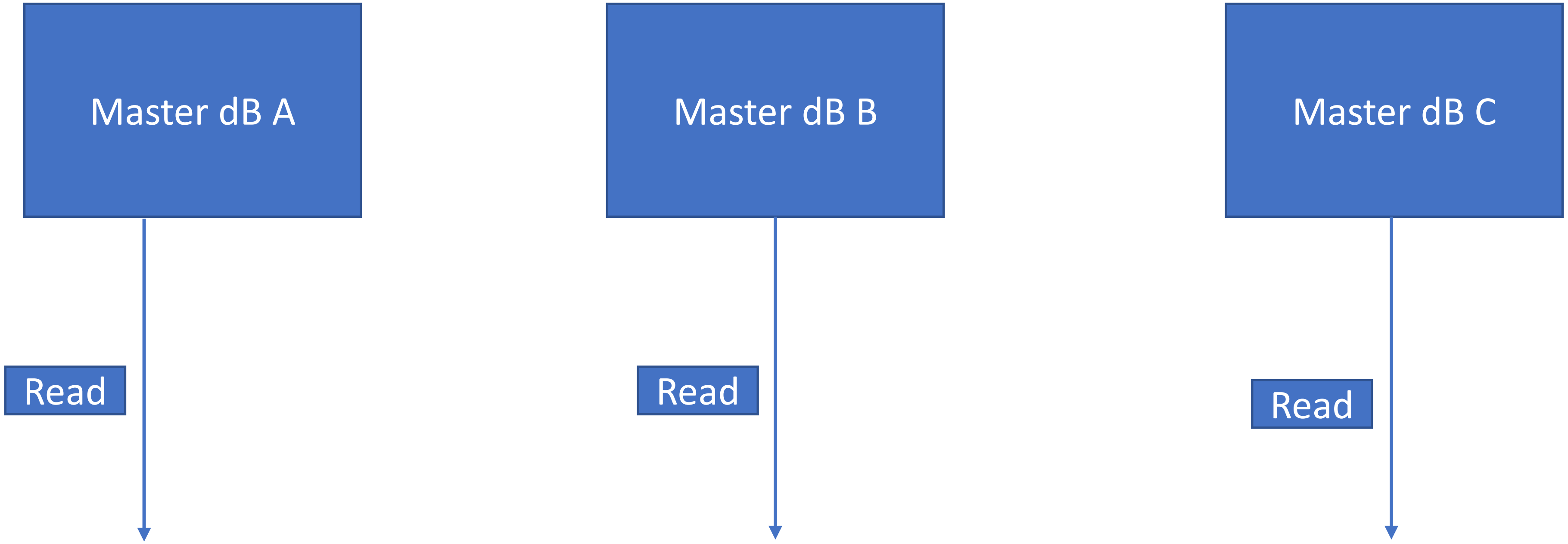
2.2 Multi Primary Replication Model

1. This model helps in **distributing the read+write** request loads across dB servers.

2. We have **multiple primary(master)** dB to read from and write to the dBs.

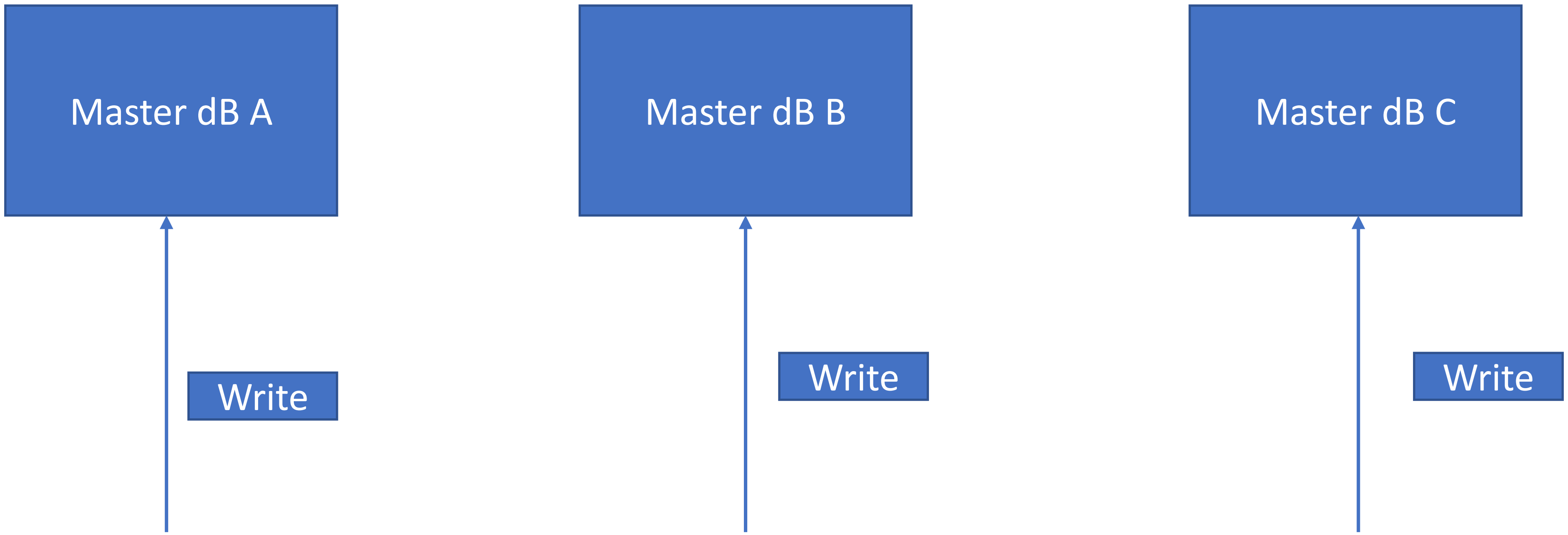
2.2 Multi Primary Replication Model

1. Every **Read request** can be redirected to any of the 3 dB Replica.
2. We use a load balancer to distribute read request load evenly.



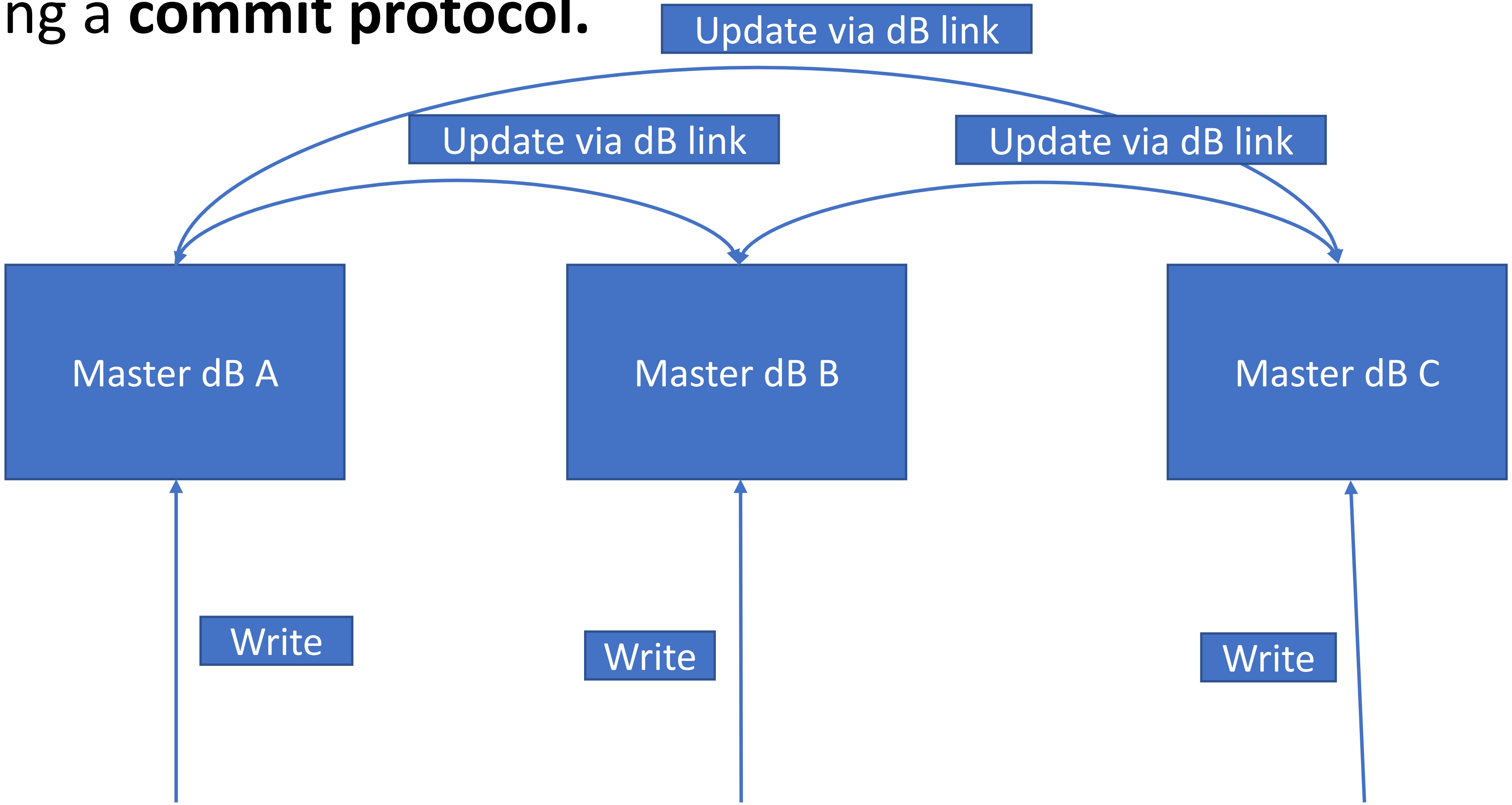
2.2 Multi Primary Replication Model

1. Every **Write request** can be redirected to any of the 3 dB Replica.
2. We use a load balancer to **distribute write request** load evenly.



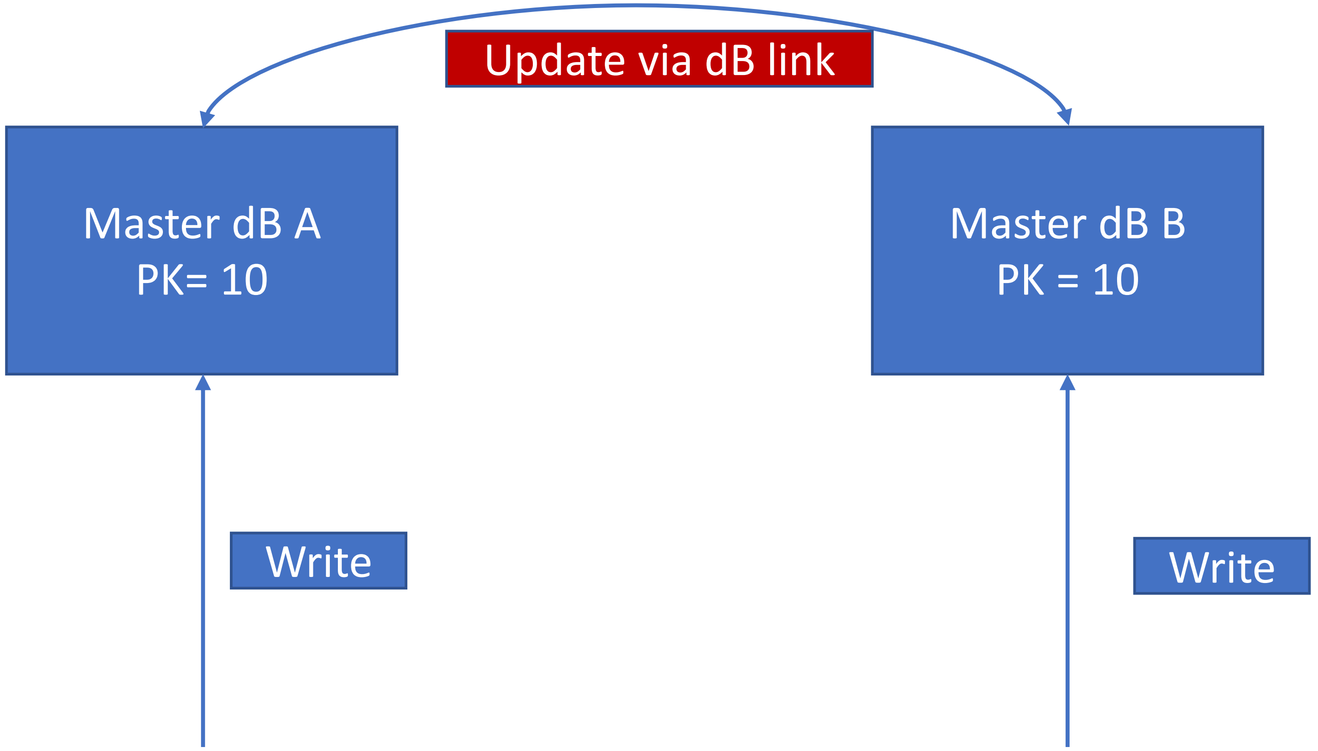
2.2 Multi Primary Replication Model

Master(Primary) dB performs a sync/async update other Master via dB links using a **commit protocol.**



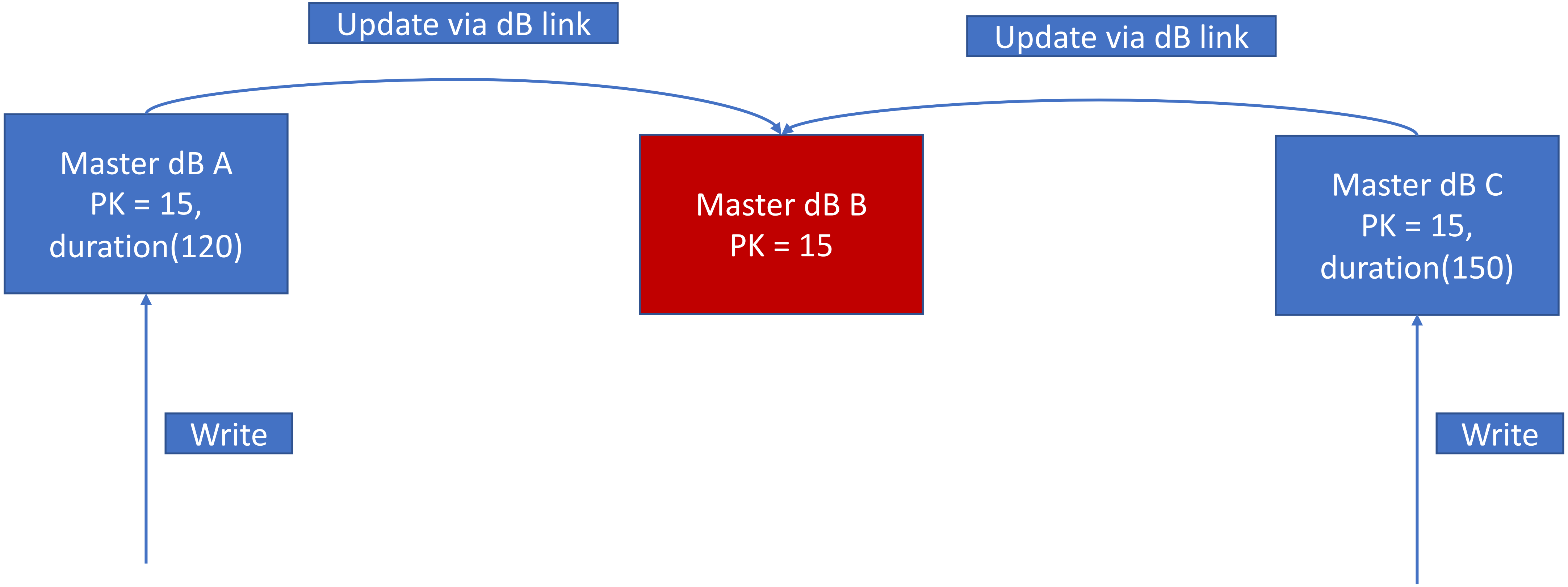
2.2 Concerns with Multi Primary Replication Model

1. We might run into **Primary Key Conflict:** If we **create** a new row into each of two **primary(master)** dBs A and B having the **same primary key**, and then an update happens between A and B to sync data.



2.2 Concerns with Multi Primary Replication Model

1. What happens when two Master dBs A and C try to update the same row in Master dB B ? Conflict



3. 2-Phase Commit Protocol

3. 2-Phase Commit Protocol

1. Generally a **dB transaction on a single dB server** follows either of two semantics to maintain its data Consistency:

ACID = **A**tomicity + **C**onsistency + **I**solation + **D**urability(SQL dBs)

BASE = **B**asically Available + **S**oft State + **E**ventual Consistency(NoSql dbs like Key-Value store, document store, wide-Column store, graph dB)

2. How do we **maintain ACID/BASE property** in case of **distributed dB servers** ?

3. 2-Phase Commit Protocol

1. This is a **distributed atomic(all-or-nothing) commit protocol** to achieve **agreement(consensus) across dB replicas** during a commit process.
2. It has **two phases**:

A. Commit Request Phase(Voting phase)
B. Commit Phase

3. Lets see how it works with a running example of **a banking dB**

3. 2-Phase Commit Protocol: Defining roles and assumptions

1. We define a **Transaction Coordinator TC**(either a separate dB node or an existing one)**:** Master dB

2. We define **participants** of the commit process: 1 Master, 2 Slaves.

3. We assume there exists a **Write-Ahead-log** at each dB server.

4. Also we assume **dB server don't crash forever** and during a dB server crash the **Write-Ahead-log** is **NOT** lost/corrupted.

3. 2-Phase Commit Protocol

1. Let’s say we want to perform a **withdraw transaction** onto a **banking dB(SQL) Customer Table** which has a **Master-Slave Replication Model.**

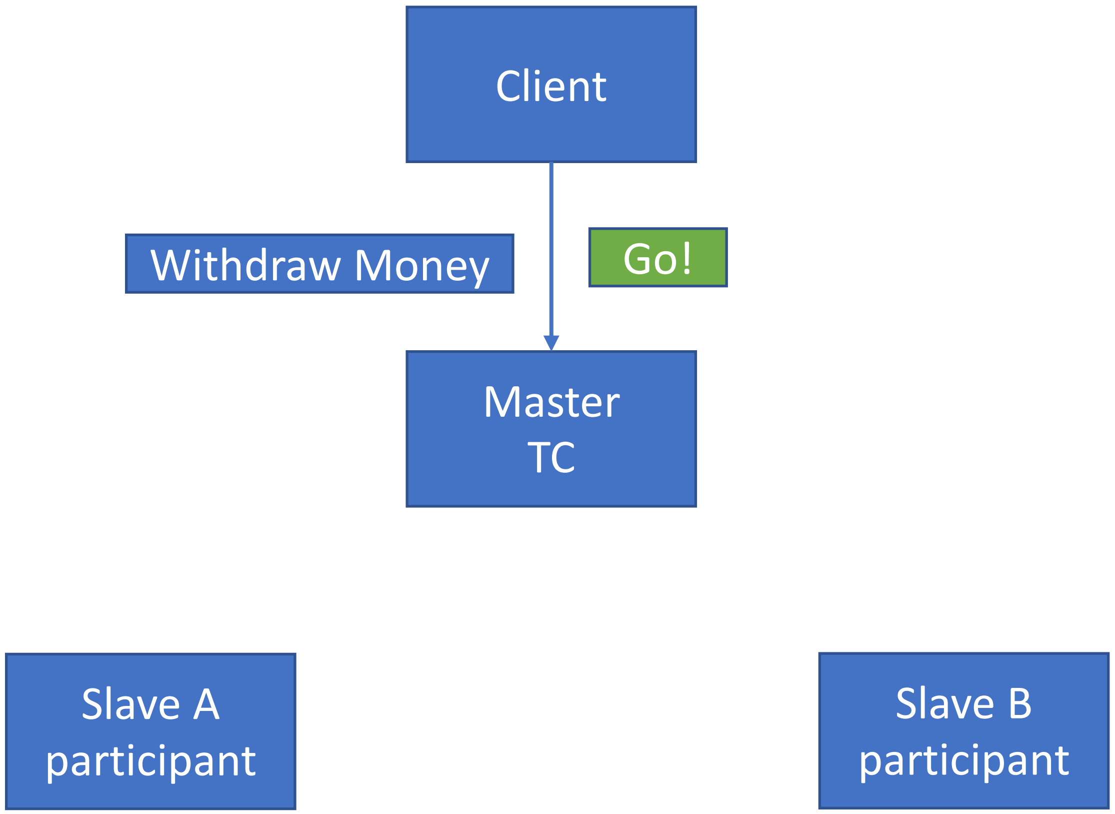
2. We have the following **withdraw transaction T**:

```
withdraw_money(Customer, amount) {
    Begin_Transaction();
    if (Customer.balance – amount >= 0) {
        Customer.balance = Customer.balance – amount;
        Commit_Transaction();
    } else {
        Abort_Transaction();
    }
```

3. 2-Phase Commit Protocol

1. Client(App Server) sends a **“withdraw_money“** request to Master dB(Transaction Coordinator)

C → TC: Go!

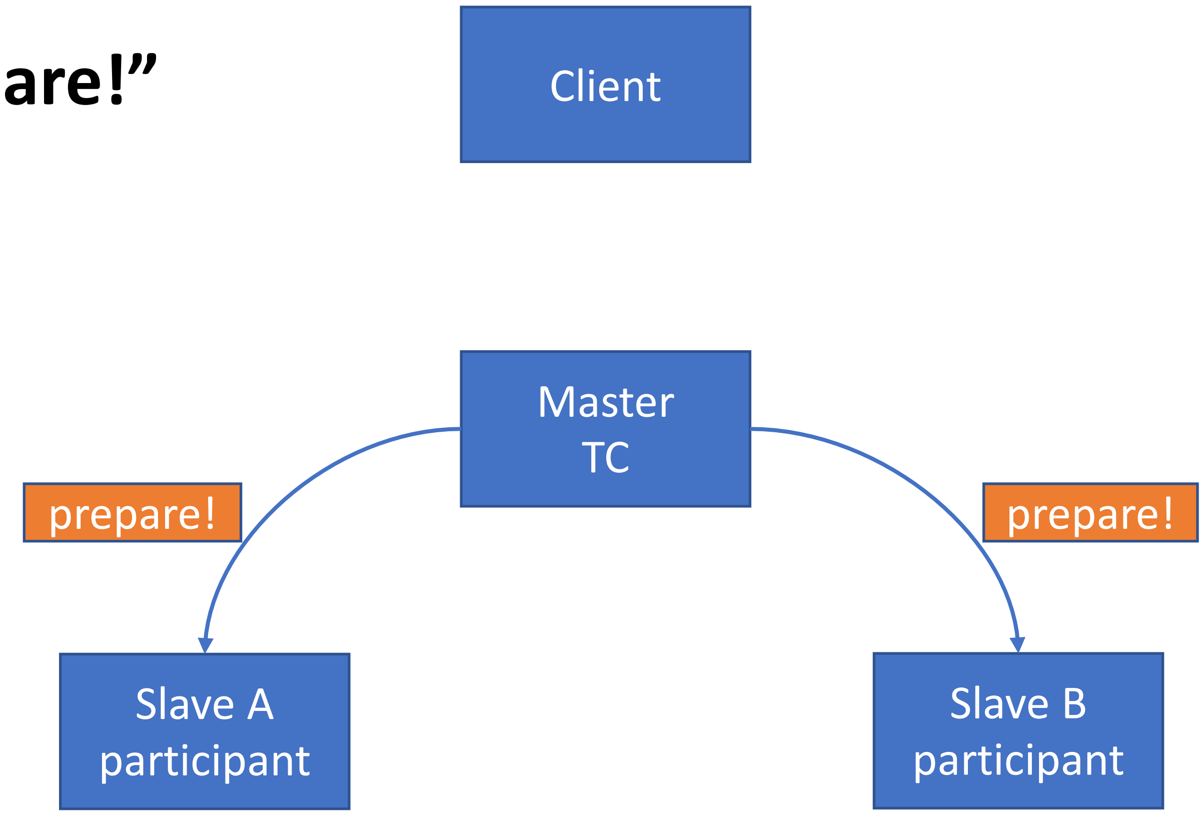


3. 2-Phase Commit: Voting Phase

2. Master(TC) sends a **"prepare!"** message to Slave A and Slave B.

TC→A: "prepare!"

TC→B: "prepare!"

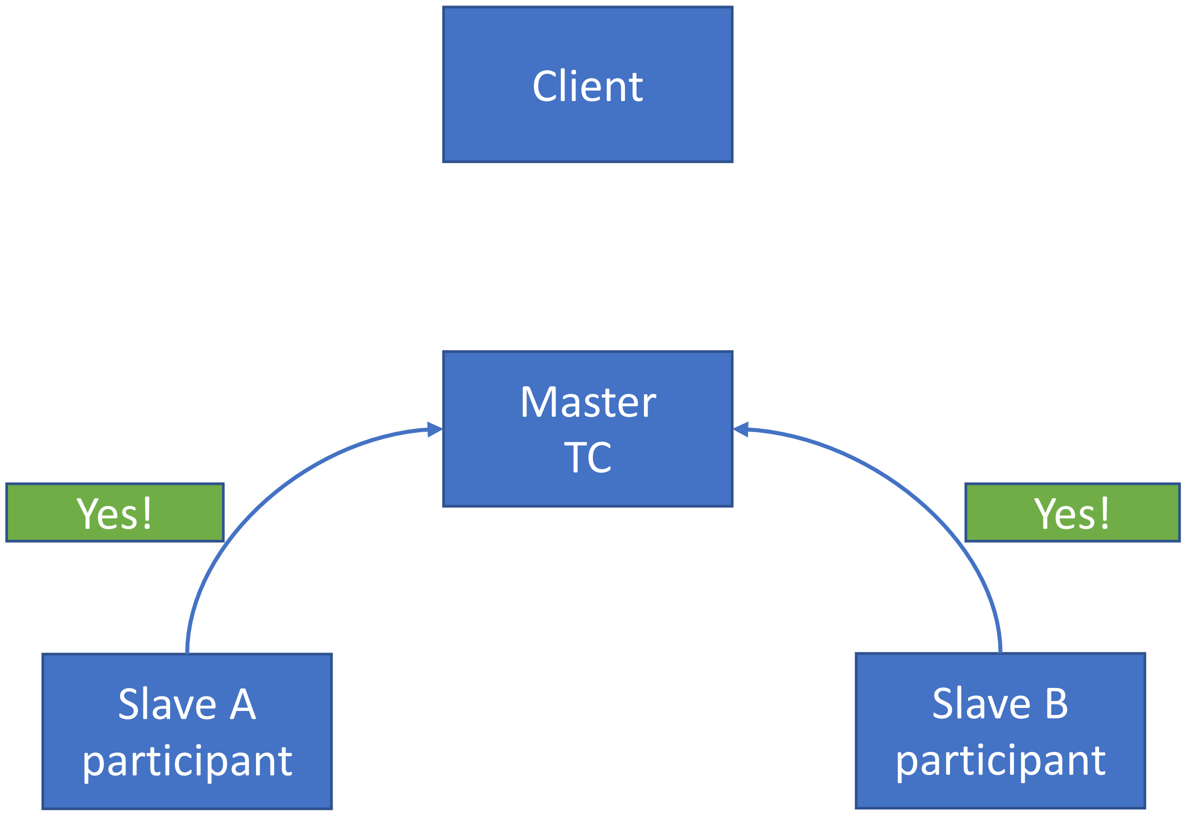


3. 2-Phase Commit: Voting Phase

3. Slave A and Slave B **executes** the transaction T upto before **commit_transaction()** and sends **"yes"** or a **"No"** to TC.

A → TC: yes!

B → TC: yes!

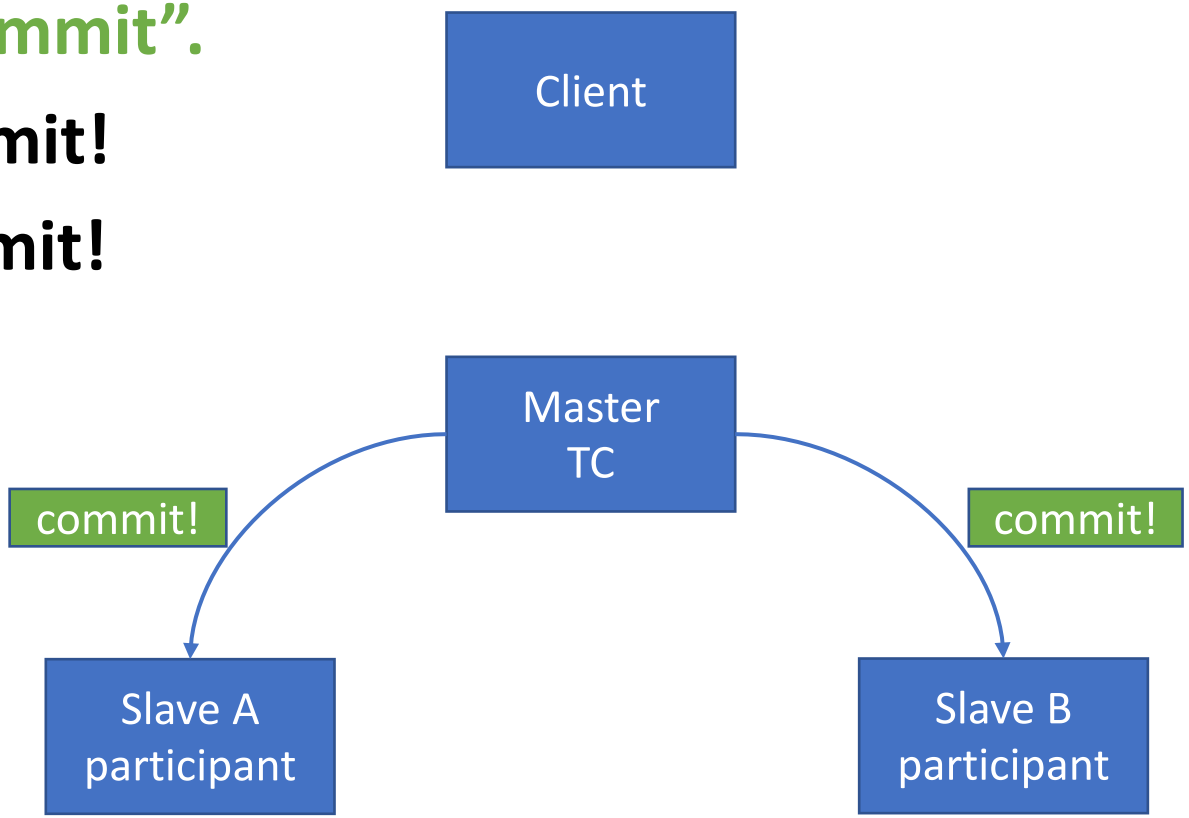


3. 2-Phase Commit: Commit Phase

4. TC sends a **commit** message if both A and B say **"yes".** TC sends an **abort** message if either says **"No"**. Slave A and Slave B **commit** on receipt of **"commit".**

TC → A: commit!

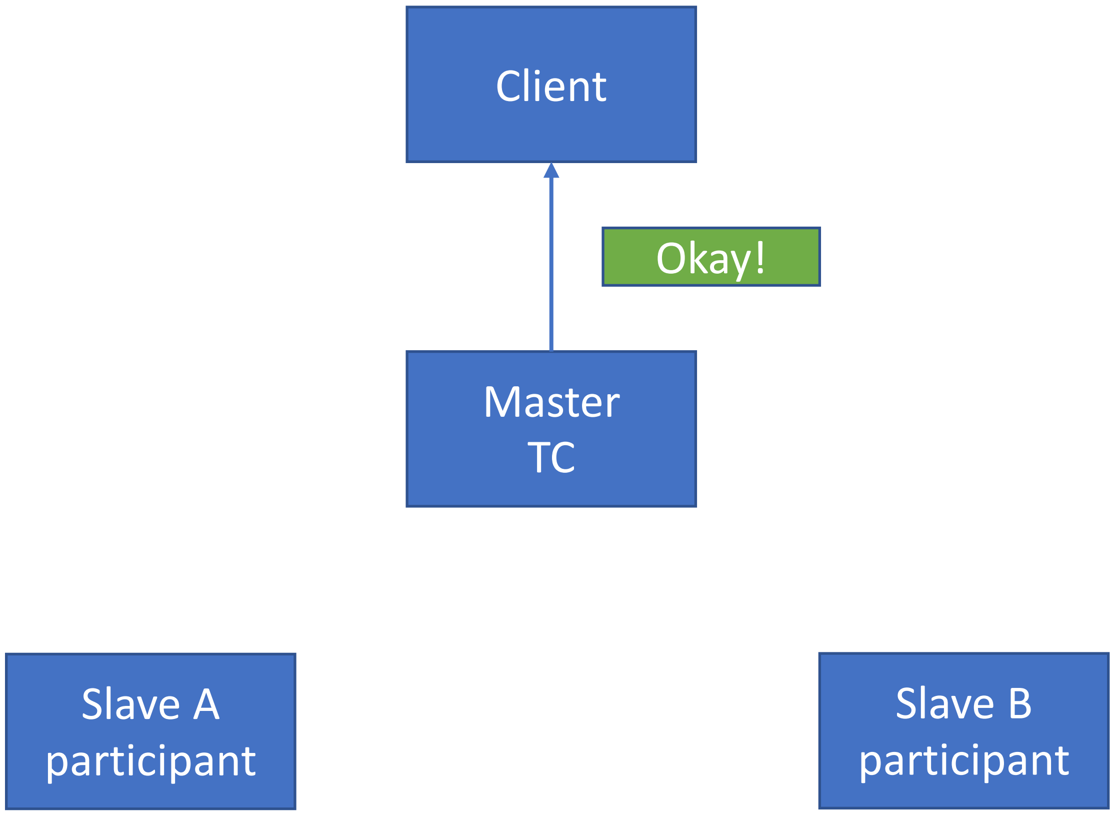
TC → B: commit!



3. 2-Phase Commit: Commit Phase

5. TC **commits** and sends **“Okay”** or **“Failed”** to the client.

TC → C: Okay!



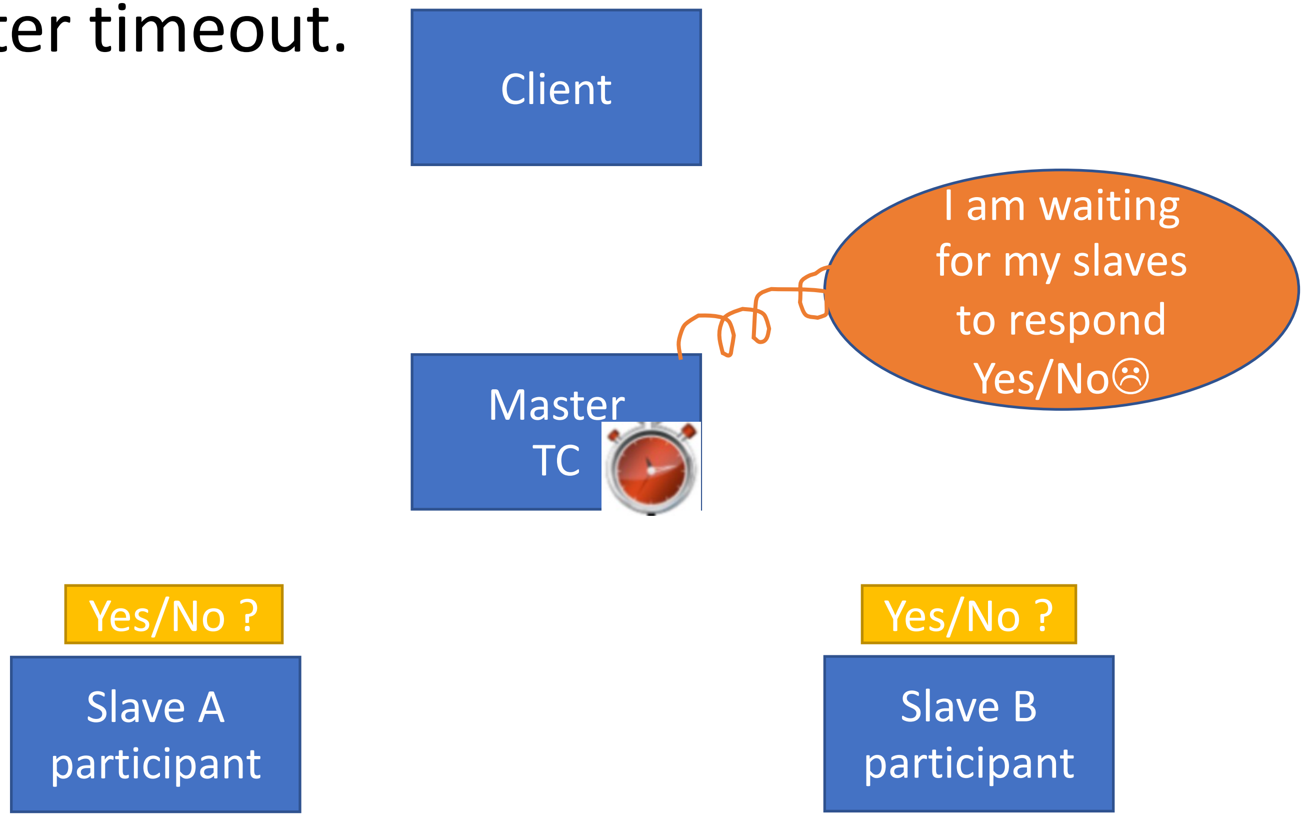
3. 2-Phase Commit: Atomic Commit Rule

1. If one dB **commits**, no dB **aborts**.
2. If one dB **aborts**, no dB **commits**.

So no dB can commit until **all the dBs agree together** to commit.

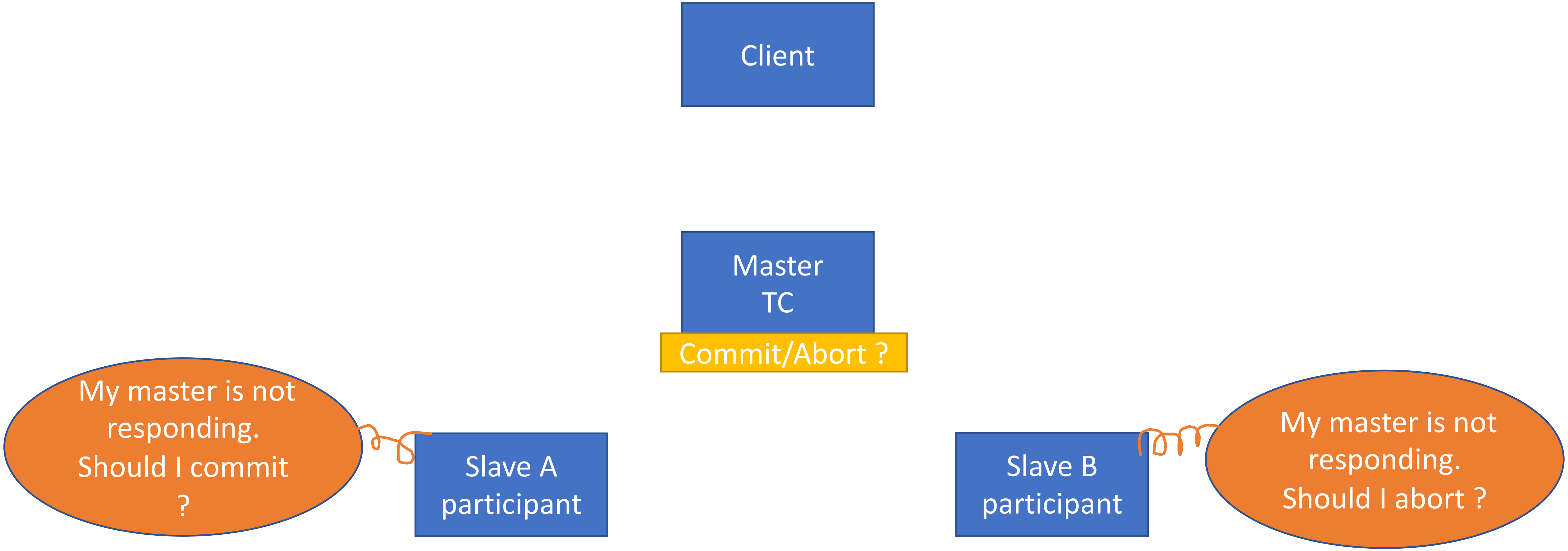
3. Timeouts in Atomic Commit

1. Let's say TC(Master) is waiting for a "yes" or "no" from A and B during **Voting Phase** ? TC hasn't sent any commit message yet so can safely **abort** after timeout.



3. Timeouts in Atomic Commit

2. Let’s say in **Commit Phase**, A and B is waiting for “**commit**” or “**Abort**” message from TC. **How long** should A and B wait ?



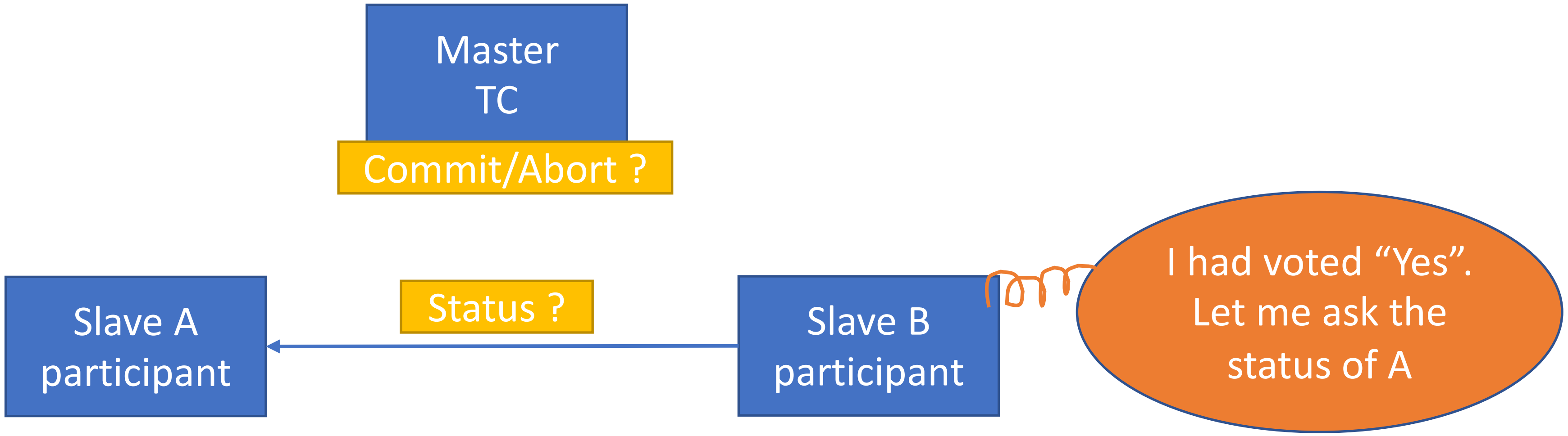
3. 2-Phase Commit: Termination Protocol

1. Let’s say Slave A and Slave B are waiting for “**Commit**” or ”**Abort**” message from TC(Master) in Commit Phase.

2. Let’s assume Slave B had voted “Yes” in Voting Phase.

3. Slave B sends a message to Slave A asking for its status:

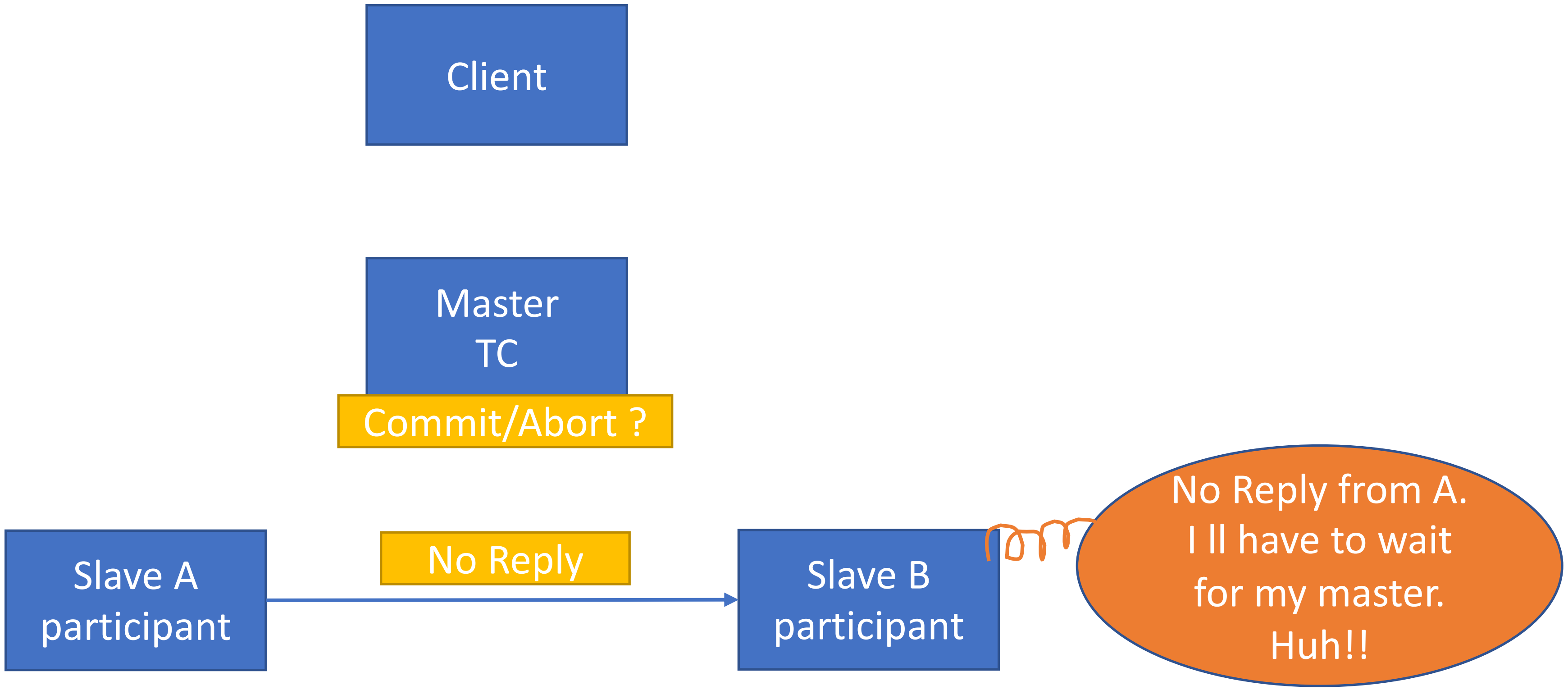
B→A: Status?



3. 2-Phase Commit: Termination Protocol

1. Four Cases arise, **Case 1**:

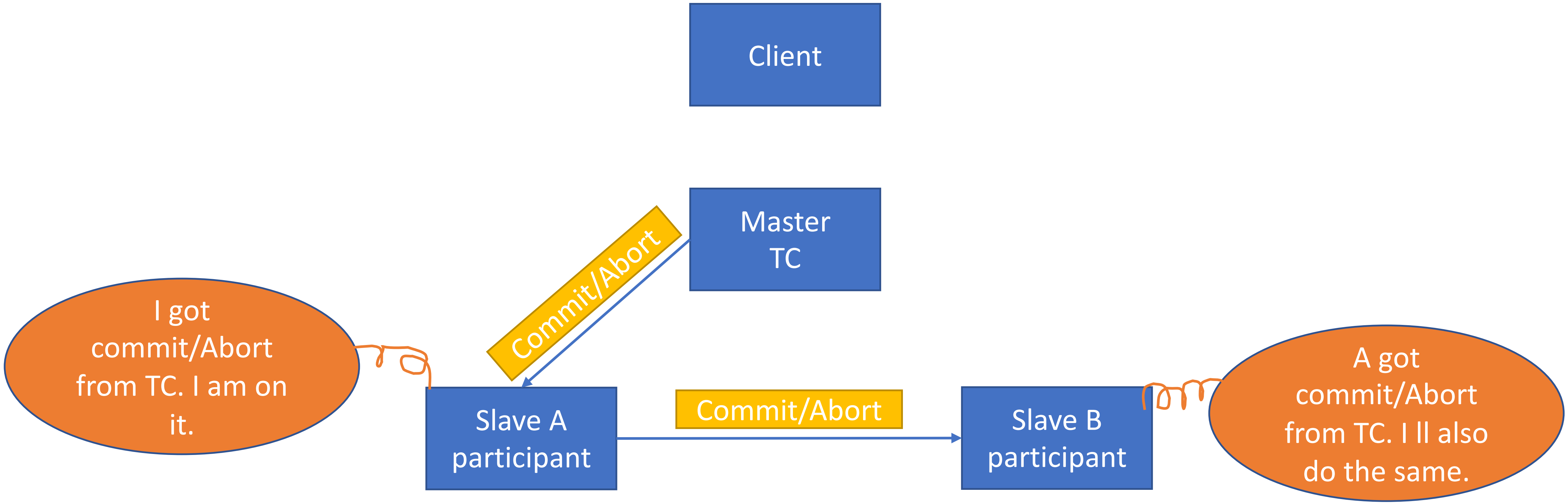
A→ B: No Reply. B continues to wait for TC



3. 2-Phase Commit: Termination Protocol

Case 2: A→ B : I got “commit”/”Abort” message from Master(TC).

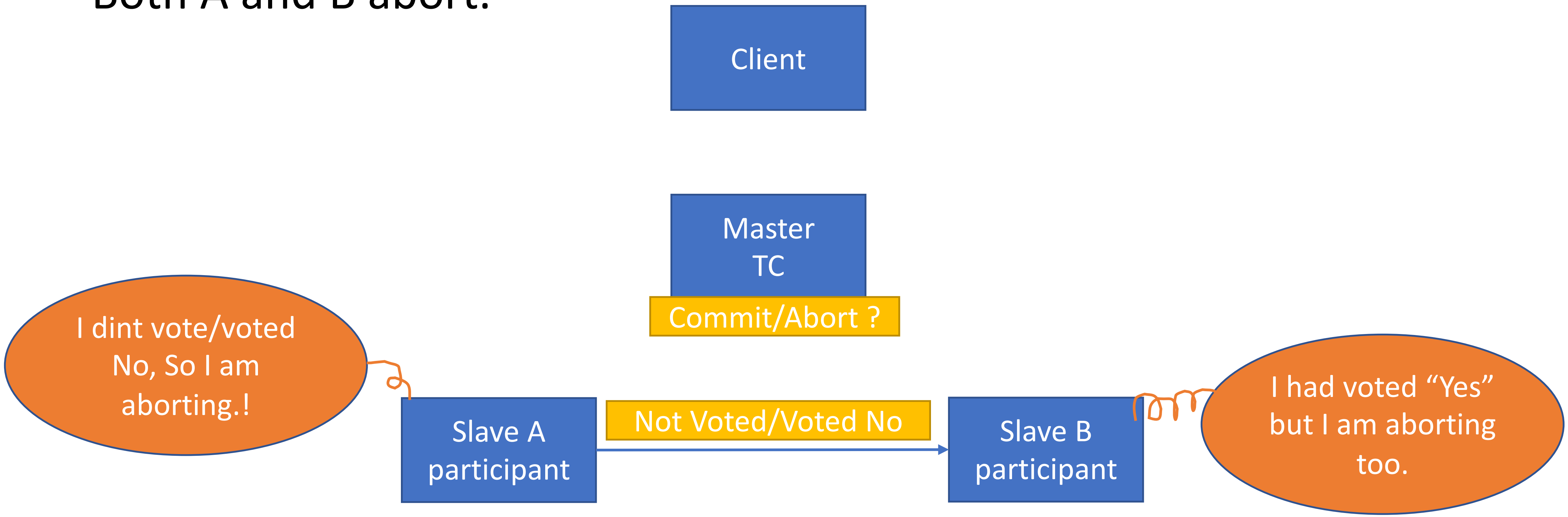
B agrees with the Master’s(TC) decision.



3. 2-Phase Commit: Termination Protocol

Case 3: A→ B : I haven’t voted yet or voted a “**No**”. So TC can’t have decided to “**Commit**”.

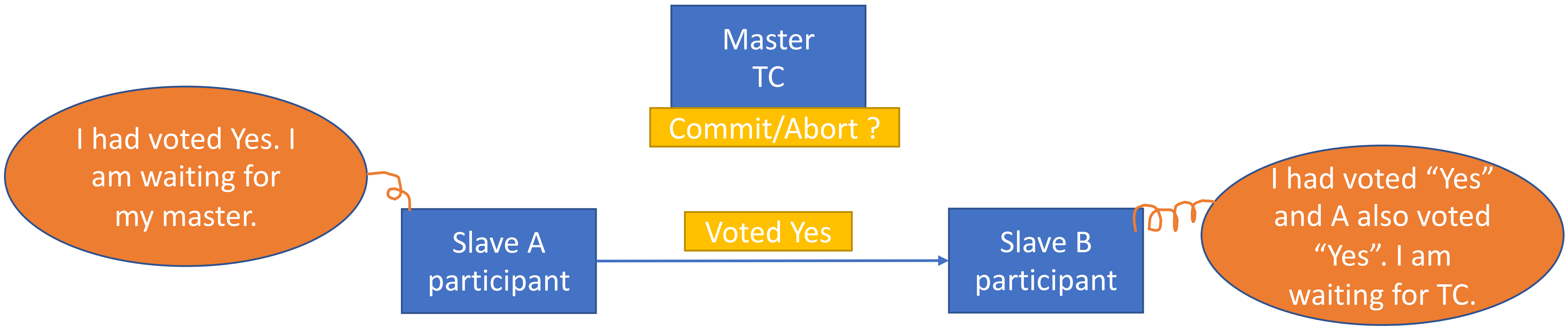
Both A and B abort.



3. 2-Phase Commit: Termination Protocol

Case 4: A→ B : I have voted "**Yes**".

Both A and B must wait for TC. TC decides to "**Commit**" if both "**Yes**" received. TC "**aborts**" if timed-out.



3. 2-Phase Commit: Handling Crash/Reboot

1. What happens if **Master(TC) crashes** just after sending "**Commit**" in Commit Phase ?

2. What happens if Slave A or Slave B **crashes** just after sending "yes" during Voting Phase ?

3. All the dB nodes use their **"Write-Ahead-log"(on disk)** to record their state before crash/Reboot.

4. When every node **reboots** and is reachable, it follows **recovery Protocol** to perform commit/Abort.

3. 2-Phase Commit: Recovery Protocol

1. Master(TC) reboots and finds no "**Commit**" record on disk. It aborts.

TC: I did not send any commit message before so I am aborting.

2. A and B reboots and doesn't find "**Yes**" record on disk, It aborts.

A/B: I dint vote "Yes" so TC could'nt have committed.

3. A and B reboots and and finds "**Yes**" record on disk, it executes termination protocol.

3. Network Failure between dB Nodes

How does a dB system behave in case of network failure between their nodes ?

4. CAP Theorem

4. CAP Theorem

It says that it is **IMPOSSSIBLE** for a distributed dB system to achieve all the three **C**onsistency, **A**vailability and **P**artition Tolerance. We can pick **only two** of them.

1. **Consistency:** Every **dB read** request receives the value of **most recent write** or an **error**.
2. **Availability:** Every dB request receives **a non-error response**. There is no guarantee that the response contains the most recent writes(uptime = 99.99% of total time of service).
3. **Partition Tolerance:** The dB system continues to operate **even if any number of messages are dropped/delayed** by network between dB nodes.

4. Consistency Patterns

1. **Weak Consistency:** After a write, reads may or may or may not see it. E.g – App Engine: memcache, VoIP, live online vid, Realtime multiplayer game
2. **Eventual Consistency**: After a write, reads will eventually see it, mail, Search Engine, Indexing, DNS, Amazon S3
3. **Strong Consistency**: After a write, reads WILL see it, App Engine: Data Store, File System, RDBMS, Azure Tables.

4. Availability Numbers

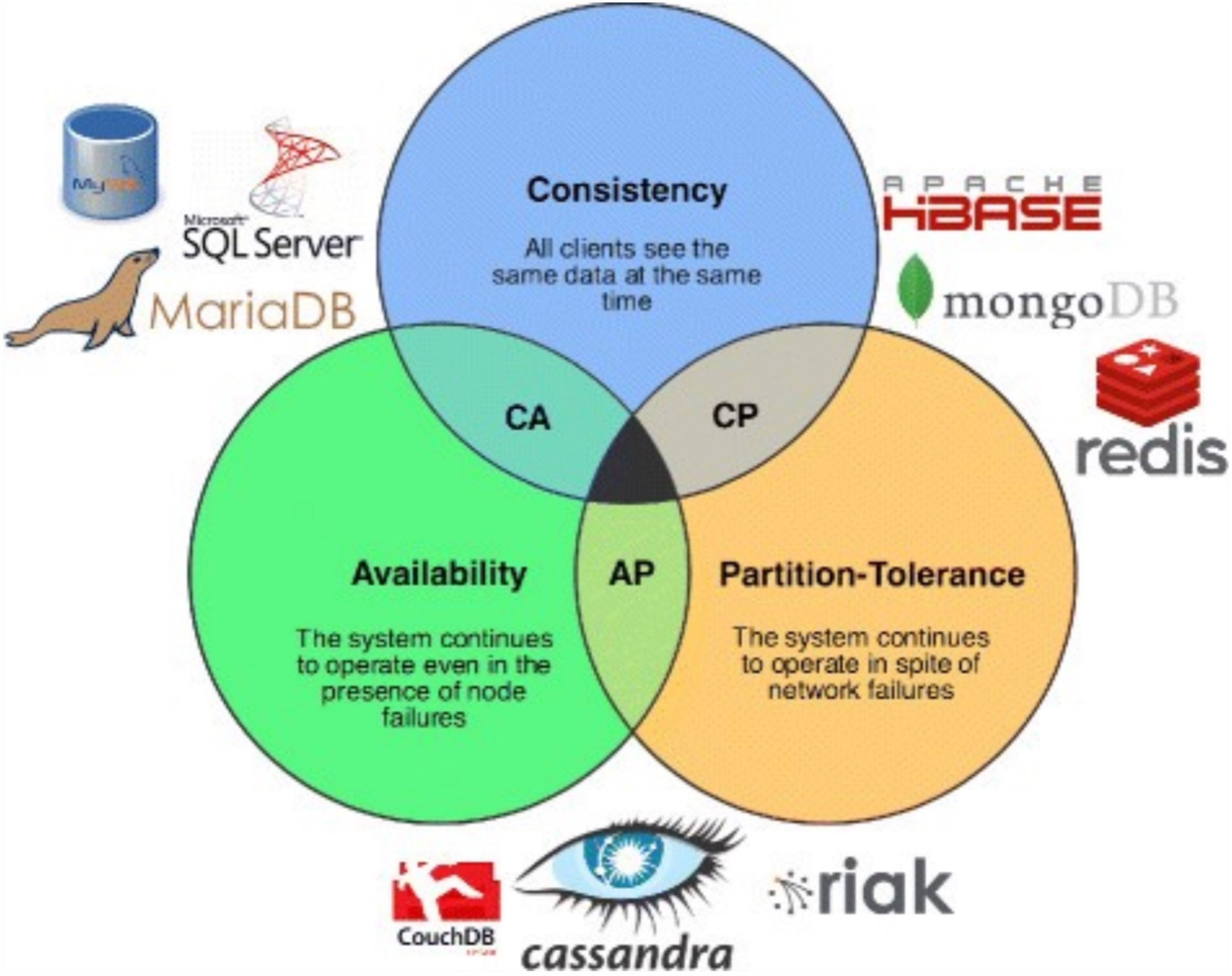
Availability is the **Uptime** of a system as a percentage of **total time of service.** It is measured in number of 9s.

99.9% availability - three 9s

Duration	Acceptable downtime
Downtime per year	8h 45min 57s
Downtime per month	43m 49.7s
Downtime per week	10m 4.8s
Downtime per day	1m 26.4s

99.99% availability - four 9s

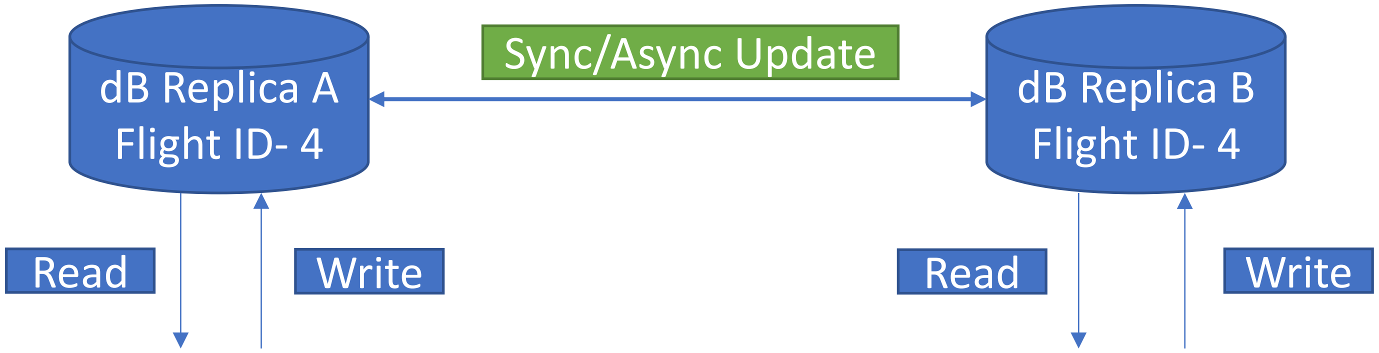
Duration	Acceptable downtime
Downtime per year	52min 35.7s
Downtime per month	4m 23s
Downtime per week	1m 5s
Downtime per day	8.6s


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
Microsoft SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

CAP Theorem

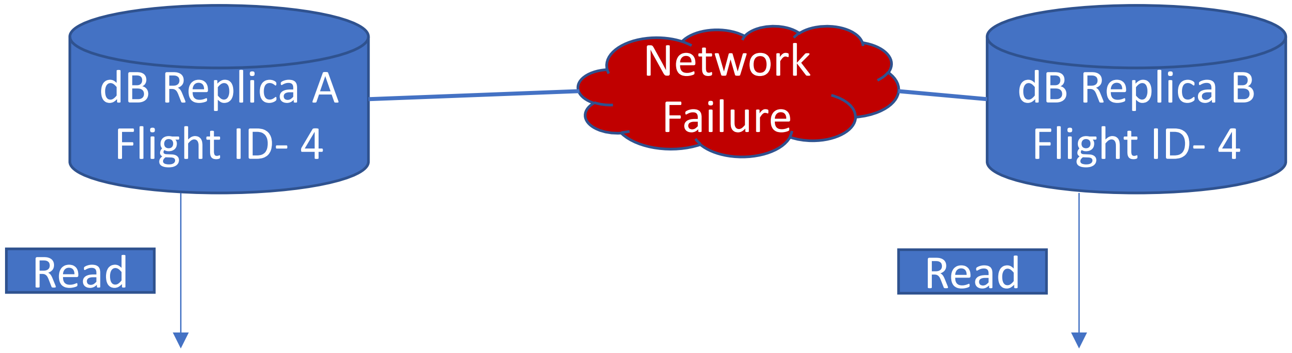
4. CAP Theorem: CA

1. A **write transaction** happening at dB Replica A will be updated to dB Replica B and vice versa. Both replica **maintain consistent copies** of data at any point of time. So every read receives most recent write.
2. Also dB A and B is **up(available)** for read/Write 99.99% of total time. So this provides CA.



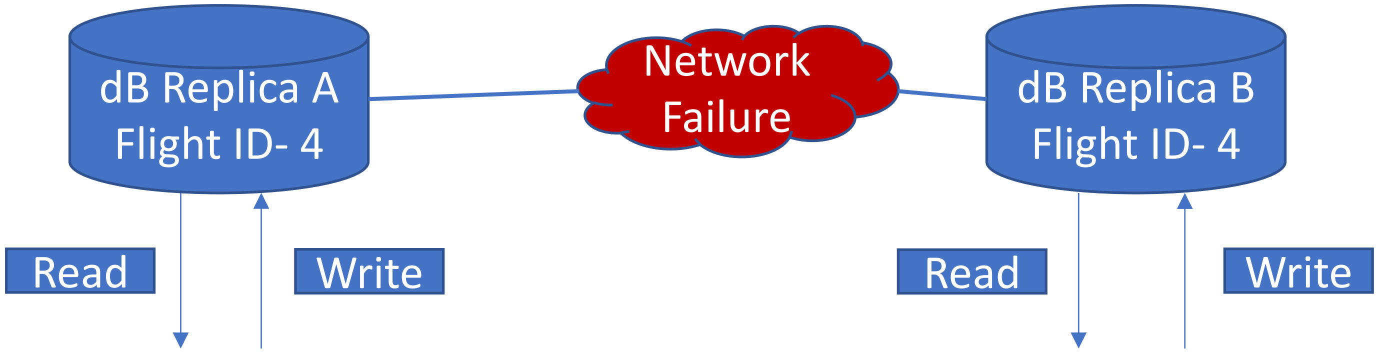
4. CAP Theorem: CP[SQL dBs]

1. Lets say the **network** connecting Replica A and B goes **down** and dBs get partitioned. Now both dBs stop serving Write requests as network failure is detected. The system is **not available** but maintains **consistent copies** of data in Replica A and B.



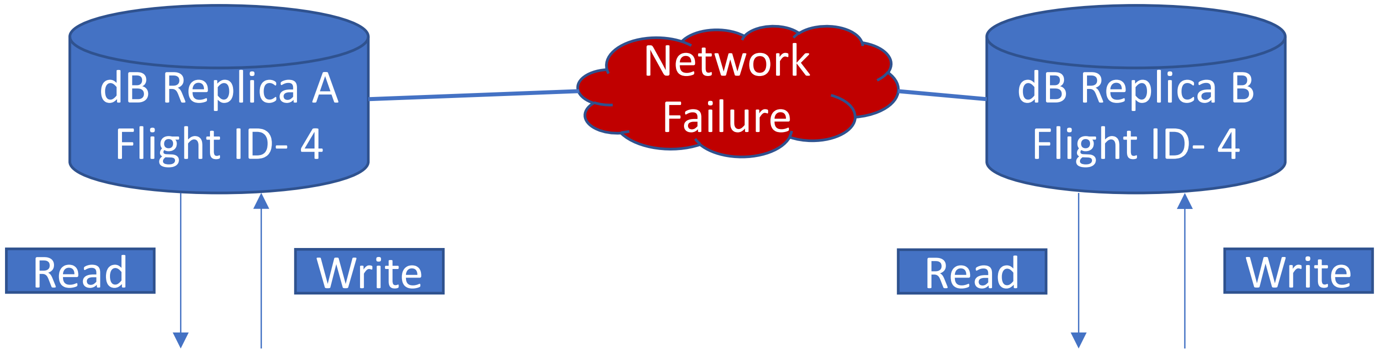
4. CAP Theorem: AP[NoSQL dBs]

1. Lets say the **network** connecting Replica A and B goes **down** and dBs get partitioned. Both dB Replicas are running fine.
2. Now a write transaction happening at A/B **can no longer** reach dB B/A resepectively. So both dB A and B end up maintaining **inconsistent copies** of data. But it continues to **serve(Available)** in a disconnected env.



4. AP and Eventual Consistency

1. The idea is the dB system **will become consistent over time** after **network failure is fixed.**
2. We **allow dB read/Write** requests on both dB Replicas even after network failure.

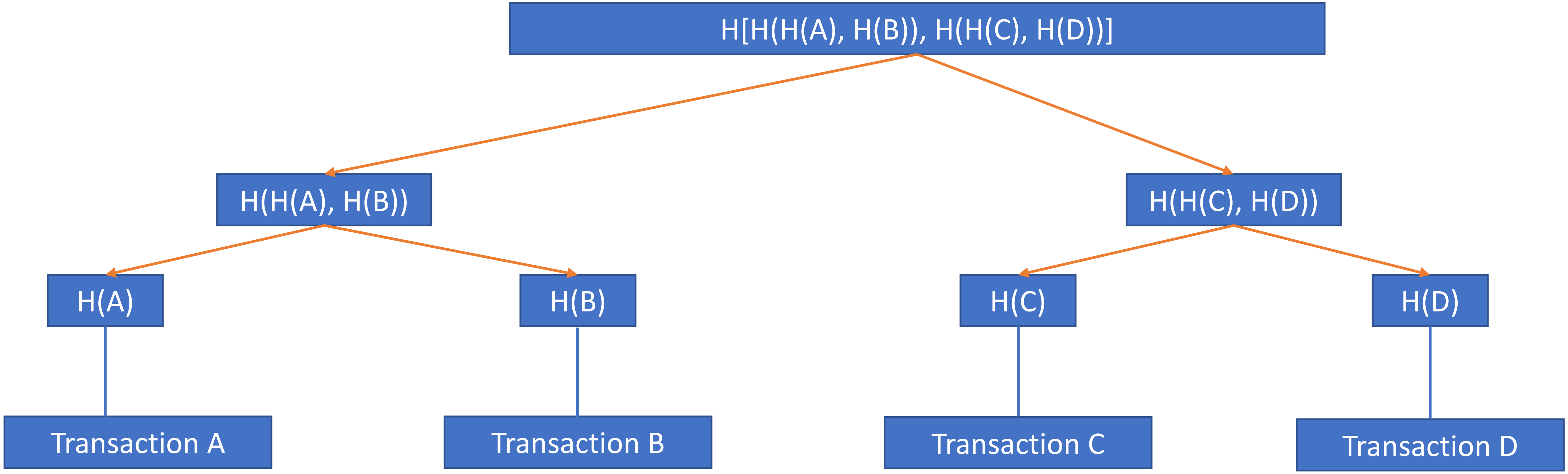


4. AP and Eventual Consistency

1. Both dB Replica A and B maintains a **Merkle Tree** which has the **latest Write Transactions.**
2. Lets say 4 write transactions A, B, C, D occurred on dB A during network Failure.
3. Each dB Replica maintains a **Merkle Tree of write transactions.**

4. Merkle Tree

1. We hash the transaction from leaf up to the root of tree **at each dB.**
2. Now we perform **Tree Traversal(O(N))** and do **data sync between dBs** as network is up.



Stay Tuned for Next session on:

Caching and CDN

Questions