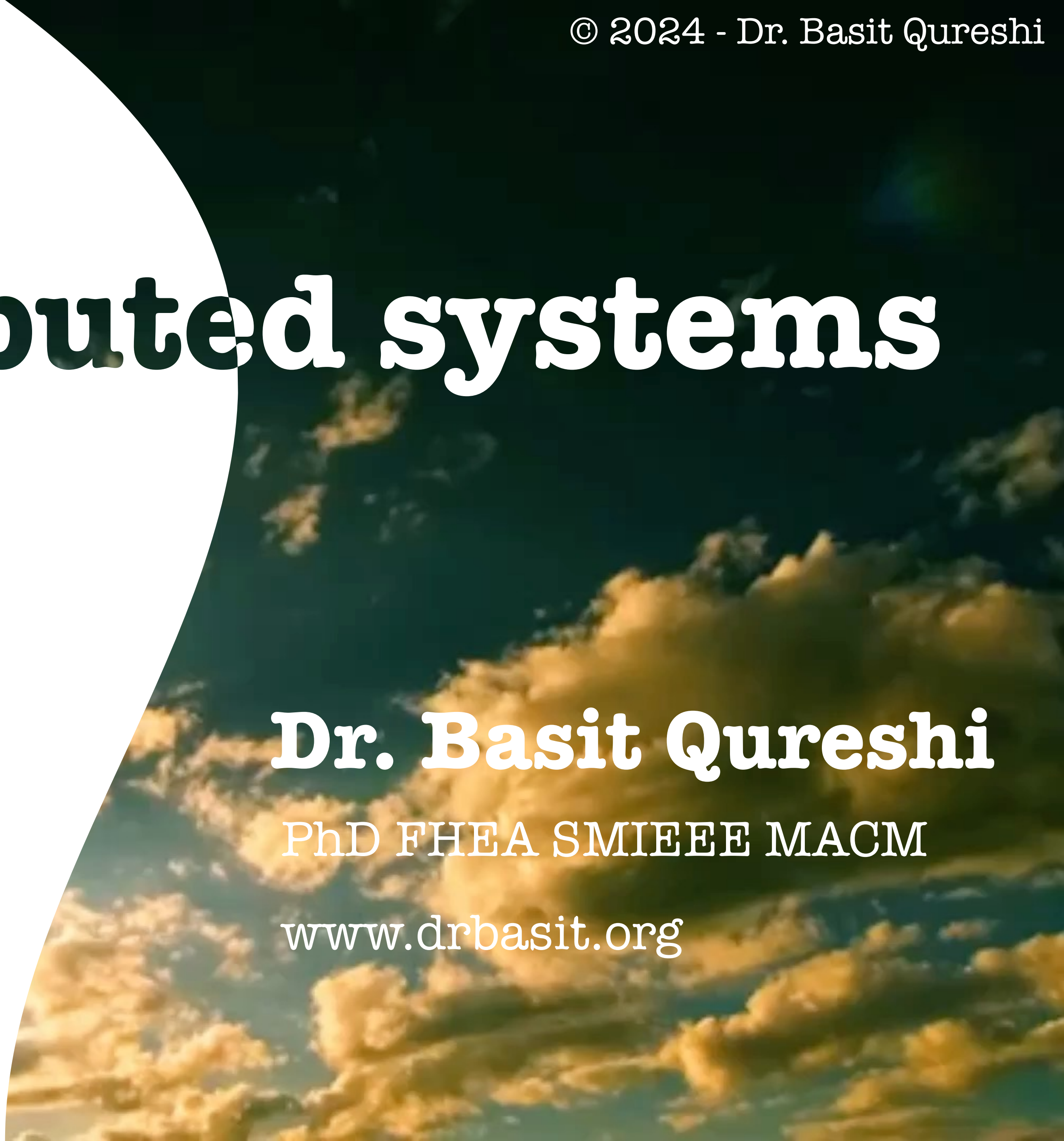
جامعة الامير سلطان
PRINCE SULTAN
UNIVERSITY

CS435 Distributed systems

CONSISTENCY, DIST TRANSACTIONS & CAP THEOREM



Dr. Basit Qureshi

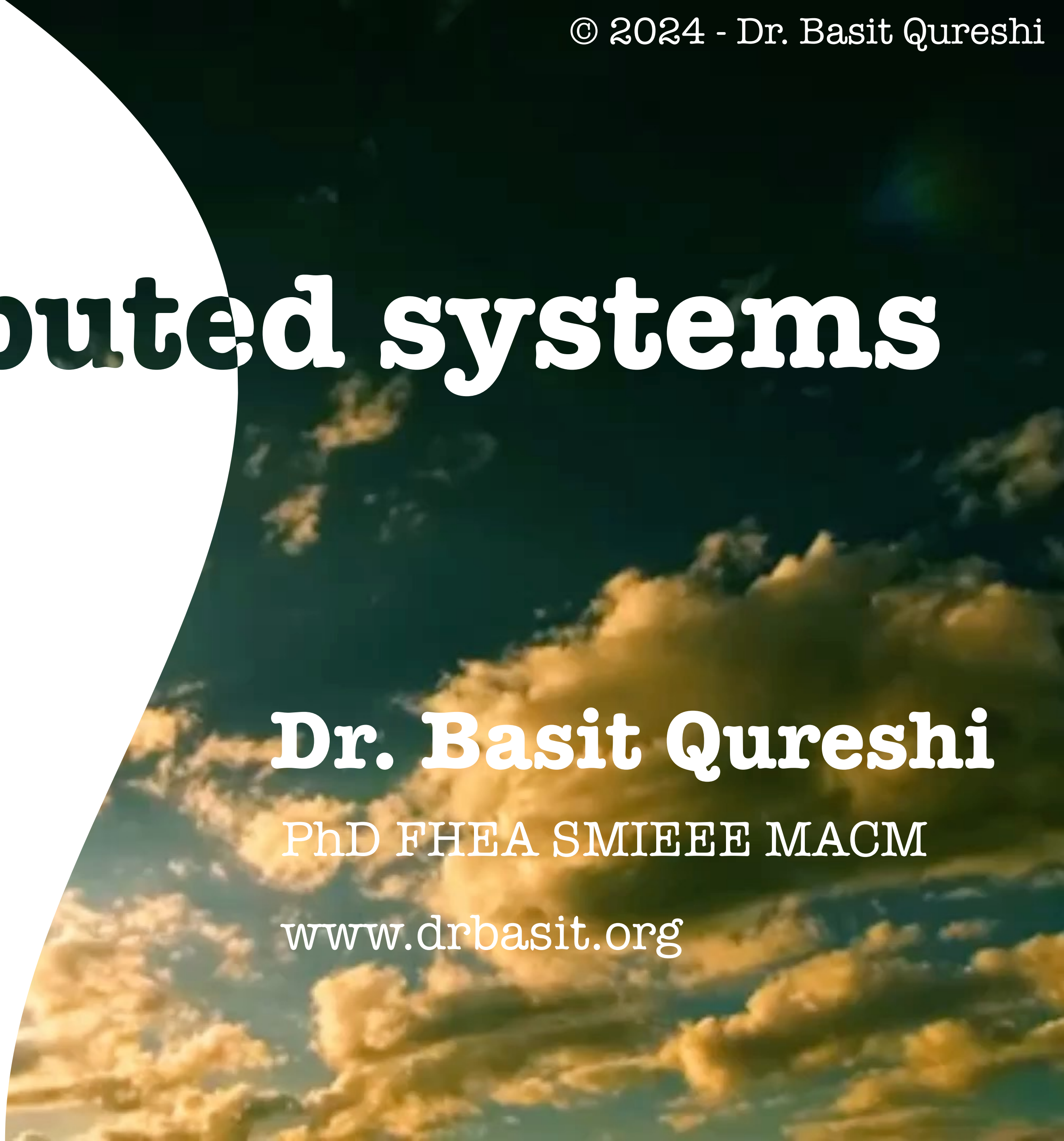
PhD FHEA SMIEEE MACM

www.drbasit.org

TOPICS

- Distributed Transactions
- 2Phase Commit
- 3Phase Commit
- Consensus based Commit
- Linearizability
- CAP Theorem

CS435 Distributed systems



DIST TRANSACTIONS

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

TRANSACTIONS

“Consistency” means different things in different contexts”

“there is no one true definition of consistency"

In Distributed Systems, we have **transactions**, where certain operations describe a property of a state; e.g., a database is in a **consistent** or **inconsistent** state, i.e., the state satisfies or violates certain invariants defined by the application

TRANSACTIONS

“A Transaction is an operation that is composed of a sequence of discrete steps. All the steps must be completed (committed) before the results are made permanent.”

Otherwise, the transaction is **aborted** and the state of the system reverts(**rollback**) to undo any changes.

TRANSACTIONS

Example:

Book a flight from Riyadh to Columbus Ohio. No non-stop flights are available:

```
Transaction begin
1. Reserve a seat for Riyadh to Newyork
2. Reserve a seat for Newyork to St Louis
3. Reserve a seat for St Louis to Columbus
Transaction end
```

If there are no seats available on the Newyork to St Louis; the entire transaction is **aborted** and reservations for (1) and (2) are **roll-backed**.

TRANSACTIONS

Transaction Basics:

– **Begin**: mark the start of a transaction

Do operations; read/write/compute data, modify files, objects, program state But any changes will have to be restored if the transaction is aborted

– **End**: mark the end of a transaction – no more tasks

– **Commit**: make the results permanent

– **Abort**: kill the transaction, roll-back old values

TRANSACTIONS

ACID

- **Atomic**

– The transaction completes as **a single indivisible action**. *Everything succeeds or else the entire transaction is rolled back*. Others do not see intermediate results.

- **Consistent**

– A transaction cannot leave the resource in an *inconsistent state & all invariants must be preserved*. E.g., total amount of money in all accounts must be the same before and after a transfer funds transaction.

- **Isolated** (Serializable)

– Transactions cannot **interfere** with each other or see **intermediate** results. If transactions run at the same time, the result must be the same *as if they executed in some serial order*.

- **Durable**

– Once a transaction commits, the results are made **permanent**.

TRANSACTIONS

Distributed Transactions

- A transaction that reads or writes data on multiple nodes
  - Data on these nodes may be **replicas** of the same dataset
  - Or different **parts** of a larger dataset
- **Challenge**
  - Handle machine, software, & network failures while preserving transaction integrity

TRANSACTIONS

Distributed Transactions

- Each computer runs a **transaction manager**
    - Responsible for sub-transactions on that system
    - Performs **PREPARE**, **COMMIT**, and **ABORT** calls for sub-transactions
- Every sub-transaction must **AGREE** to *commit changes before* the overall transaction can complete
- Caveat (**Consensus**)
    - **AGREE** on a value proposed by at least one process
    - BUT we need **unanimous agreement (100%)** to commit change

TRANSACTIONS

Distributed Transactions

- Coordinator
  - Propose to commit a transaction
  - **All participants agree** ⇒ all participants then **commit**
  - **Not all participants agree** ⇒ all participants then **abort**

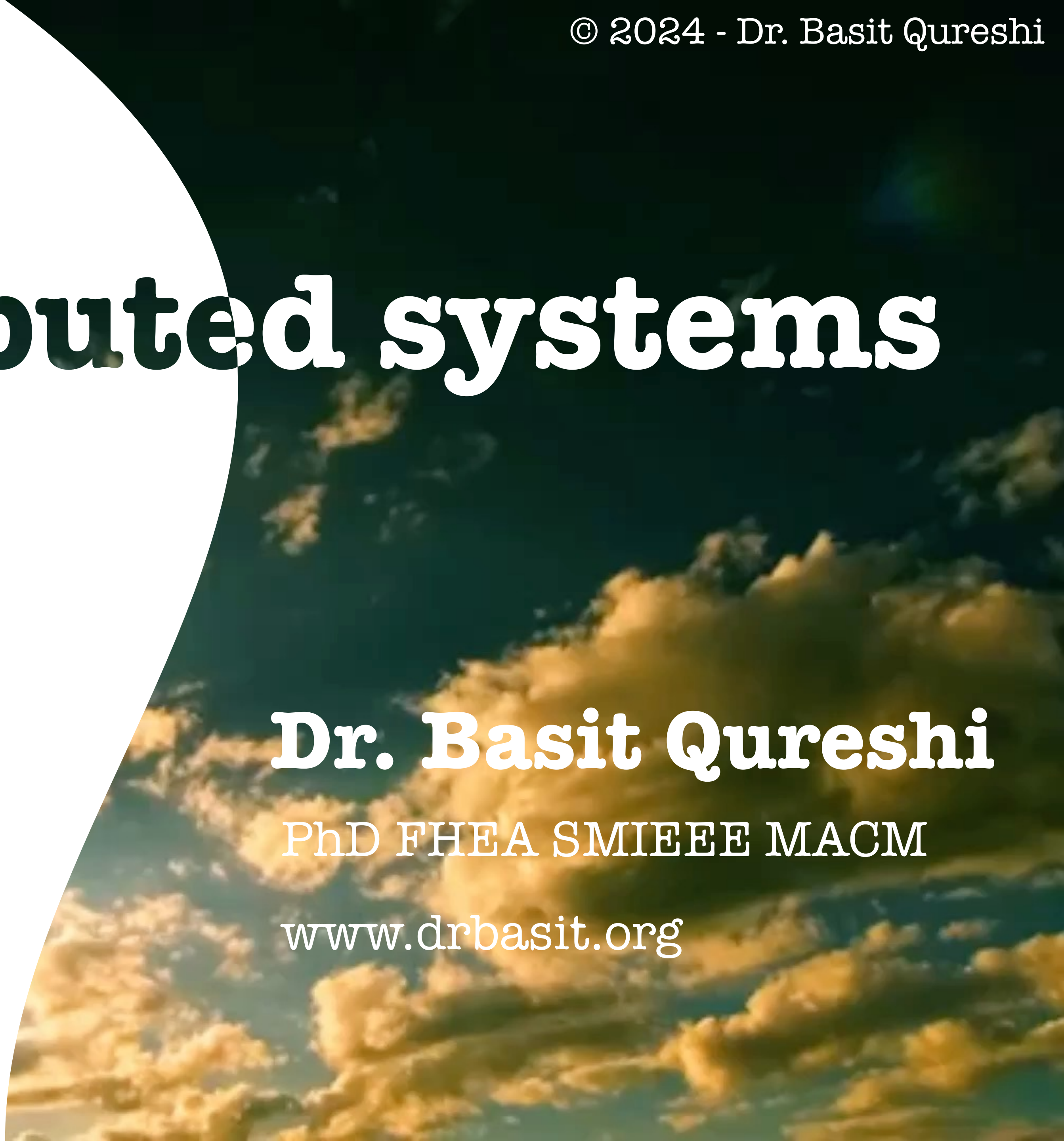
TRANSACTIONS

Distributed Transactions Algorithms must have:

- **Safety** (the algorithm must work correctly)
  - If one sub-transaction commits, no other sub-transaction will **abort**
  - If one sub-transaction needs to abort, no sub-transactions will **commit**
- **Liveness** (the algorithm must make progress & reach its goal)
  - If no sub-transactions fail, the transaction will **commit**
  - If any sub-transactions fail, the algorithm will *reach a conclusion* to **abort**

CS435 Distributed systems

2PC



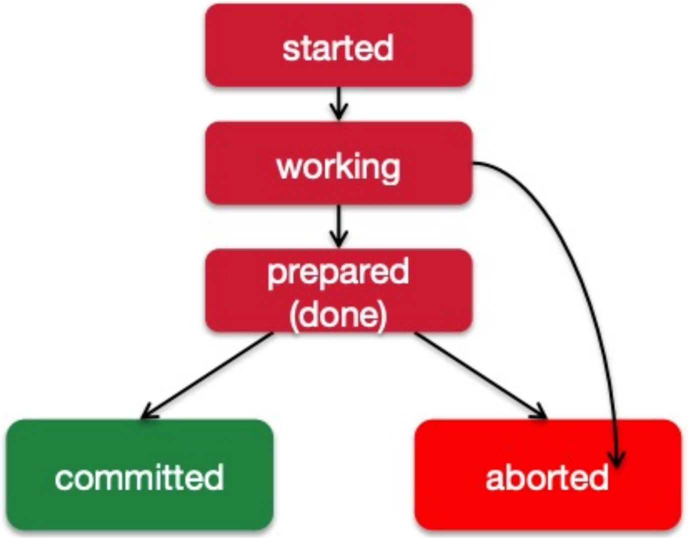
Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

TWO PHASE COMMIT PROTOCOL (2PC)

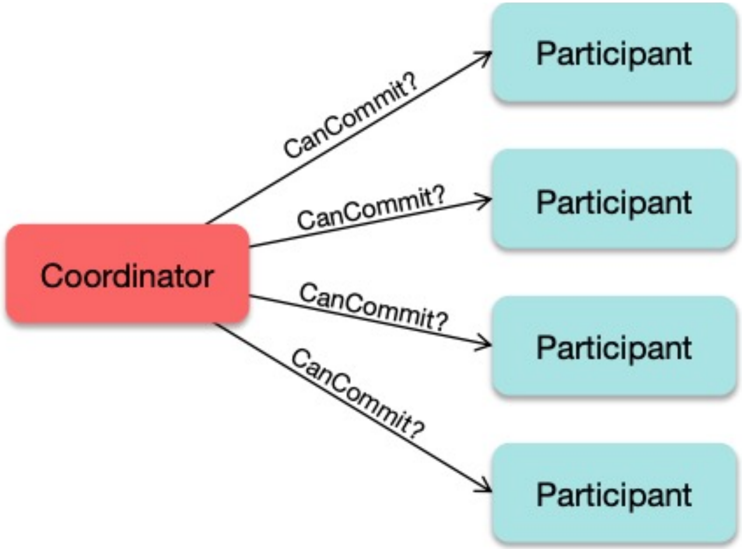
- Two Phase Commit (2PC) ensures atomic commitment.
  - All processes in the transaction will **agree** to
    - **commit** or **abort**
  - One transaction manager is elected as a ***coordinator*** – the rest are ***participants***
  - When a participant enters the **prepared** state:
    - It contacts the *coordinator* to start the *commit* protocol to commit the *entire transaction*

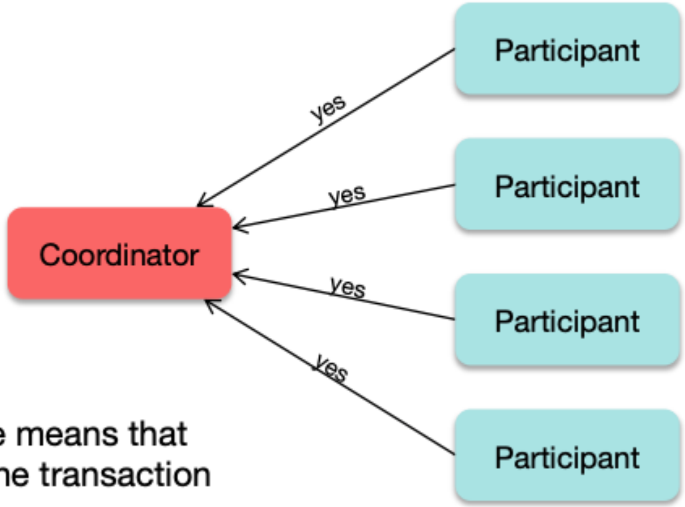


TWO PHASE COMMIT PROTOCOL (2PC)

Phase 1: Voting

- Get commit agreement from every participant



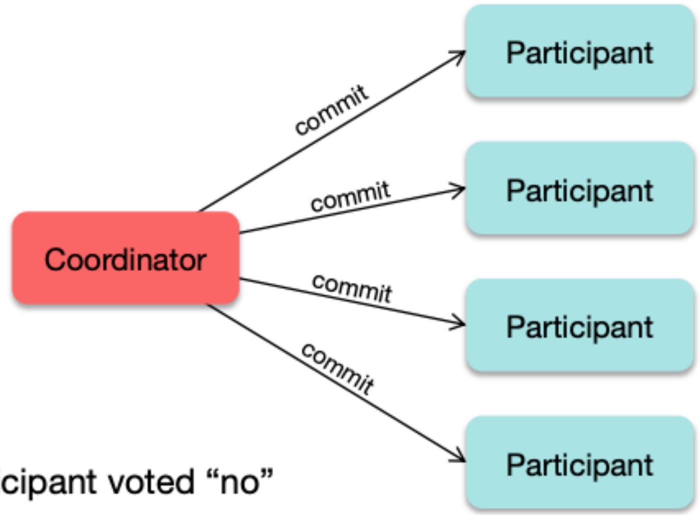


A single “no” response means that we will have to abort the transaction

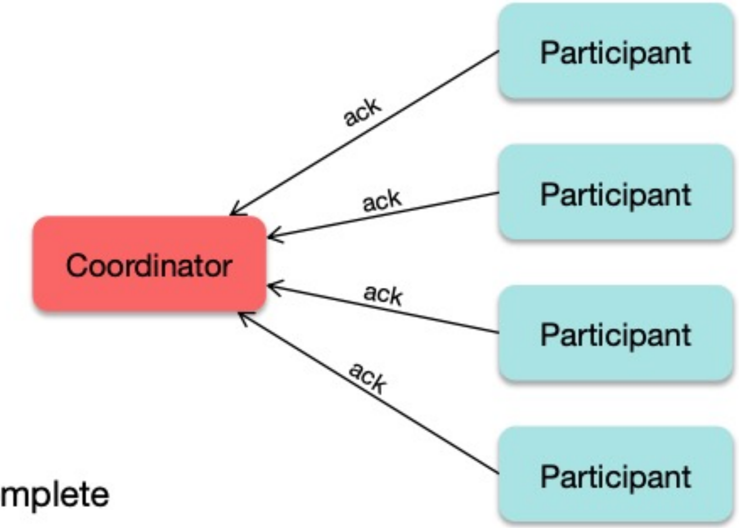
TWO PHASE COMMIT PROTOCOL (2PC)

Phase 2: Commit

- Send the results of the vote to every participant.



Send *abort* if any participant voted “no”



The transaction is complete

TWO PHASE COMMIT PROTOCOL (2PC)

Dealing with failure (Voting)

Voting

Coordinator dies

- Some participants may have responded; others have no clue

⇒ **Coordinator restarts voting**: *checks log*; sees that voting was in progress

Participant dies

- The participant may have died *before or after sending its vote* to the coordinator

⇒ If coordinator received the vote: *wait for other votes* and then **goes to Phase 2**

⇒ Otherwise: **wait for the participant** to *recover and respond* (keep querying it)

TWO PHASE COMMIT PROTOCOL (2PC)

Dealing with failure (Commit)

Commit

Coordinator dies

- Some participants may have been given commit/abort instructions

⇒ **Coordinator restarts**; *checks log*; informs **all** participants of chosen action

Participant dies

- The participant may have died *before or after getting the commit/abort* request

⇒ Coordinator **keeps trying to contact the participant** with the request

⇒ **Participant recovers**; checks log; gets request from coordinator

  - If it committed/aborted, acknowledge the request
  - Otherwise, process the commit/abort request and send back the acknowledgement

TWO PHASE COMMIT PROTOCOL (2PC)

Dealing with failure

Recovery Coordinator?

- Another Participant can take over as coordinator
- Contact ALL participants to see how they voted
- If we get voting results from all participants
  - We know that Phase 1 has completed
  - If *all participants voted to commit* ⇒ send **commit** request
  - Otherwise send **abort** request
- If ANY participant states that it has *not voted*
  - We know that Phase 1 has **not completed**
  - ⇒ Restart the protocol
- **2PC ensures Strong consistency…but … if any participant node also crashes, we’re stuck**

CS435 Distributed systems

3PC

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

THREE PHASE COMMIT PROTOCOL (3PC)

Problem with 2PC

- A **blocking** protocol with failure modes that require all systems to recover eventually
- If the coordinator crashes, participants have **no idea** whether to **commit** or **abort**
  - A **recovery coordinator** helps
- If a **coordinator AND a participant** crashes
  - The system has no way of knowing the result of the transaction
  - It might have committed at the crashed participant
  - Hence all others must block

THREE PHASE COMMIT PROTOCOL (3PC)

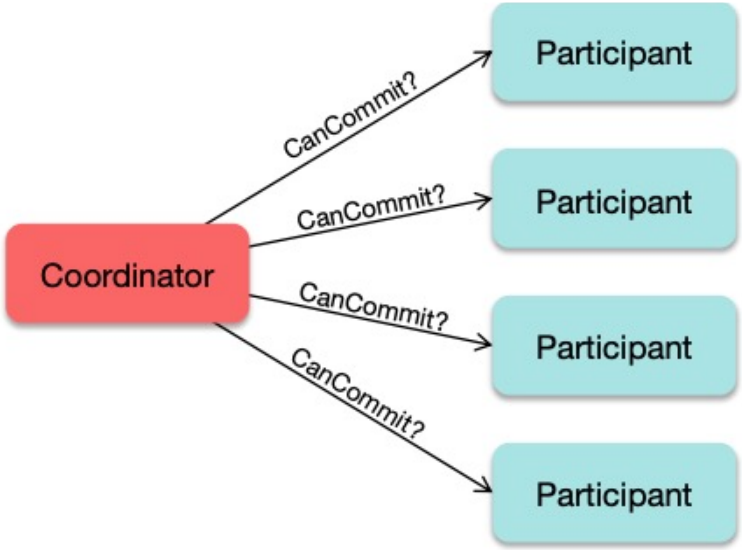
3PC

- Same as 2PC but ADD **timeouts** to each phase that result in an **abort**
- Propagate the result of the **commit/abort** vote to each participant *before* telling them to act on it; this will allow us to recover the state of the transaction from any participant and give more options for aborting

THREE PHASE COMMIT PROTOCOL (3PC)

Phase 1: Voting

- Coordinator sends **CanCommit?** queries to participants & gets responses
- If the coordinator gets a **timeout** from any participant or any "No" replies are received
  - Send an **abort** to all participants
- If a participant **times out waiting** for a request from the coordinator
  - It **aborts** itself (assume coordinator crashed)
- Else **continue** to phase 2



THREE PHASE COMMIT PROTOCOL (3PC)

Phase 2: Prepare to commit

- Send a **prepare** message to all participants
- Get **OK** messages from all participants
  - We need to hear from **all** before proceeding so we can be sure the state of the protocol can be properly recovered if the coordinator dies i.e. let all participants know the decision to commit
- If a participant **times out**: assume it crashed; send abort to all participants

THREE PHASE COMMIT PROTOCOL (3PC)

Phase 3: Finalize

- Send **commit** messages to participants and get responses from all
- If participant times out: contact any other participant and move to that state (**commit** or **abort**)
- If coordinator times out: **abort**

THREE PHASE COMMIT PROTOCOL (3PC)

3PC Recovery

- If the coordinator crashes; a **recovery node** can query the state from any available participant
- Possible states that the participant may report:

Already committed

- That means that **every** other participant has received a **Prepare to Commit**
- Some participants may have **committed** ⇒ Send **Commit** message to all participants (just in case they didn’t get it)

Not committed but received a Prepare message

- That means that all participants agreed to commit; *some may have committed*
- Send **Prepare to Commit** message to all participants (just in case they didn’t get it)
- Wait for everyone to acknowledge; then commit

Not yet received a Prepare message

- This means no participant has committed; some may have agreed
- Transaction can be **aborted** or the commit protocol can be **restarted**

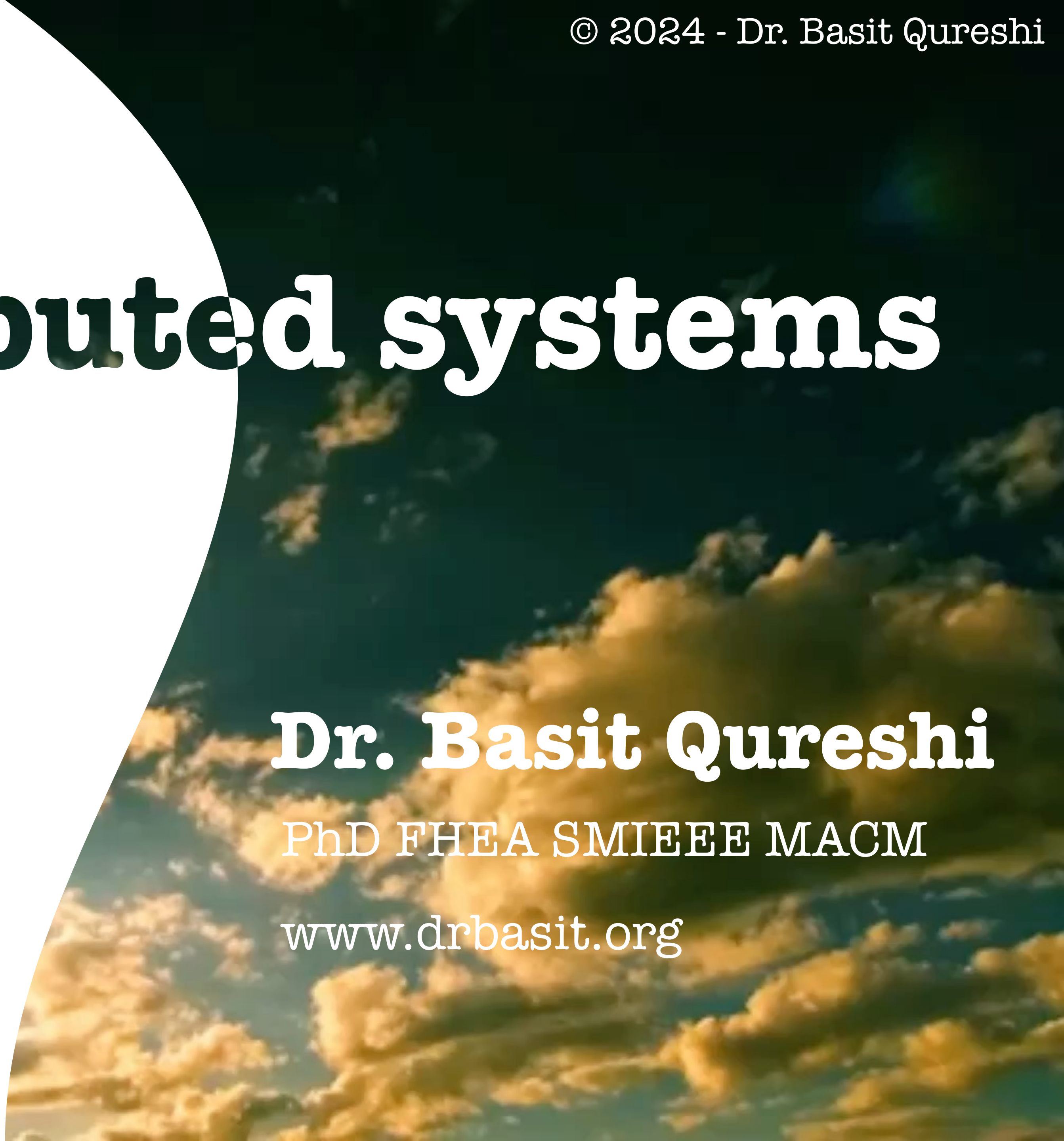
THREE PHASE COMMIT PROTOCOL (3PC)

3PC Weakness

- Suppose a coordinator sent a Prepare message to all participants
  - All participants acknowledged the message
  - BUT the coordinator **died** before it got all acknowledgements
- A recovery coordinator queries a participant.
  - It continues with the **commit**: Sends **Prepare**, gets **ACKs**, sends **Commit**
- Around the same time…the original coordinator recovers
  - Realizes it is still missing some replies from the Prepare
  - Gets timeouts from some and decides to send an Abort to all participants
- **Some processes may commit while others abort!**
- 3PC works well when servers crash (fail-stop model) BUT it is not resilient to **network partitions, fail-recovery** and **extra latency**

CS435 Distributed systems

CONSENSUS BASED COMMIT



Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

3PC

Problem with 3PC

- Suppose a coordinator sent a Prepare message to all participants
  - All participants acknowledged the message BUT the coordinator died before it got all acknowledgements
  - A recovery coordinator queries a participant; It continues with the commit: Sends Prepare, gets ACKs, sends Commit
  - Around the same time…**the original coordinator recovers;**
    - Realizes it is still missing some replies from the Prepare;
    - Gets timeouts from some and decides to send an Abort to all participants
  - Some processes may commit while others abort!
  - **3PC works well when servers crash (fail-stop model) But …**
    - 3PC is not resilient against **fail-recover environments**
    - 3PC is not resilient against **network partitions**
    - Also, 3PC involves an extra round of messages vs. 2PC → **extra latency!**

CONSENSUS BASED COMMIT

- Consensus-based protocols (Raft, Paxos) are designed to be resilient against network partitions
- But consensus protocols are designed to solve a different problem!
  - Majority agreement makes sense in replicated state machines, not in distributed transactions, where each sub-transaction has different responsibilities
- Can we make 2PC use a consensus algorithm?
  - Turn the coordinator into a fault-tolerant replicated state machine
  - Use replicated nodes to avoid blocking if the coordinator fails
  - Run a consensus algorithm on the commit/abort decision of EACH participant

CONSENSUS BASED COMMIT

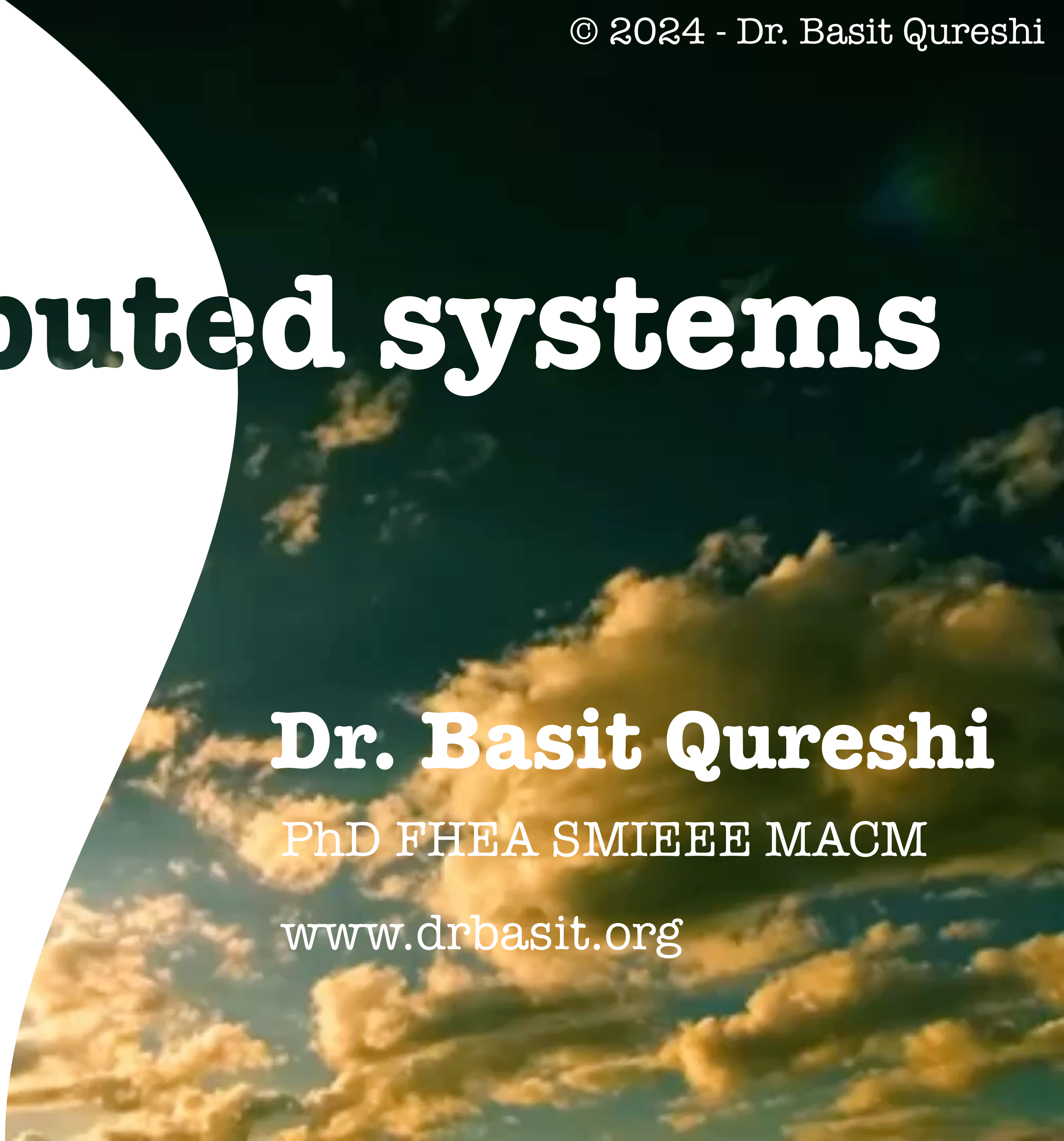
- Each participant must get its chosen value
  - **can_commit** or **must_abort**
  - accepted by the majority of replicated nodes
  - Transaction Leader
    - Chosen via an election algorithm
    - Coordinates the commit algorithm
    - Not a single point of failure – we can elect a new one; Raft nodes store state

CONSENSUS BASED COMMIT

- Some *participant* decides to begin to commit
  - Sends a message to the Transaction Leader
- **Transaction Leader:** Sends a prepare message to each participant
- Each *participant* now sends a **can_commit** or **must_abort** message to its instance of the consensus protocol
  - All participants share the elected Transaction Leader
  - "**Can_commit**" or "**Must_abort**" is sent to majority of followers
  - Result is sent to the leader
- **Transaction Leader** tracks all instances of the commit protocol
  - Commit **iff every** participant's instance of the consensus protocol chooses "can_commit"
  - Tell each participant to **commit** or **abort**

CS435 Distributed systems

LINEARIZABILITY



Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

LINEARIZABILITY

Linearizability, aka, Atomic consistency or strong consistency, is a consistency model for distributed systems that guarantees that each operation appears to have occurred *instantaneously* at a single point in time, known as its **linearization point**, and that operations from different nodes appear to have executed in a sequential order

LINEARIZABILITY

Linearizability ≠ Serializability

- **Serializability** means that transactions have the same effect as if they had been executed in some serial order, but it does not define what that order should be.
- **Linearizability** defines the values that operations must return, depending on the concurrency and relative ordering of those operations

LINEARIZABILITY

Multiple nodes concurrently accessing replicated data. How do we define "consistency" here?

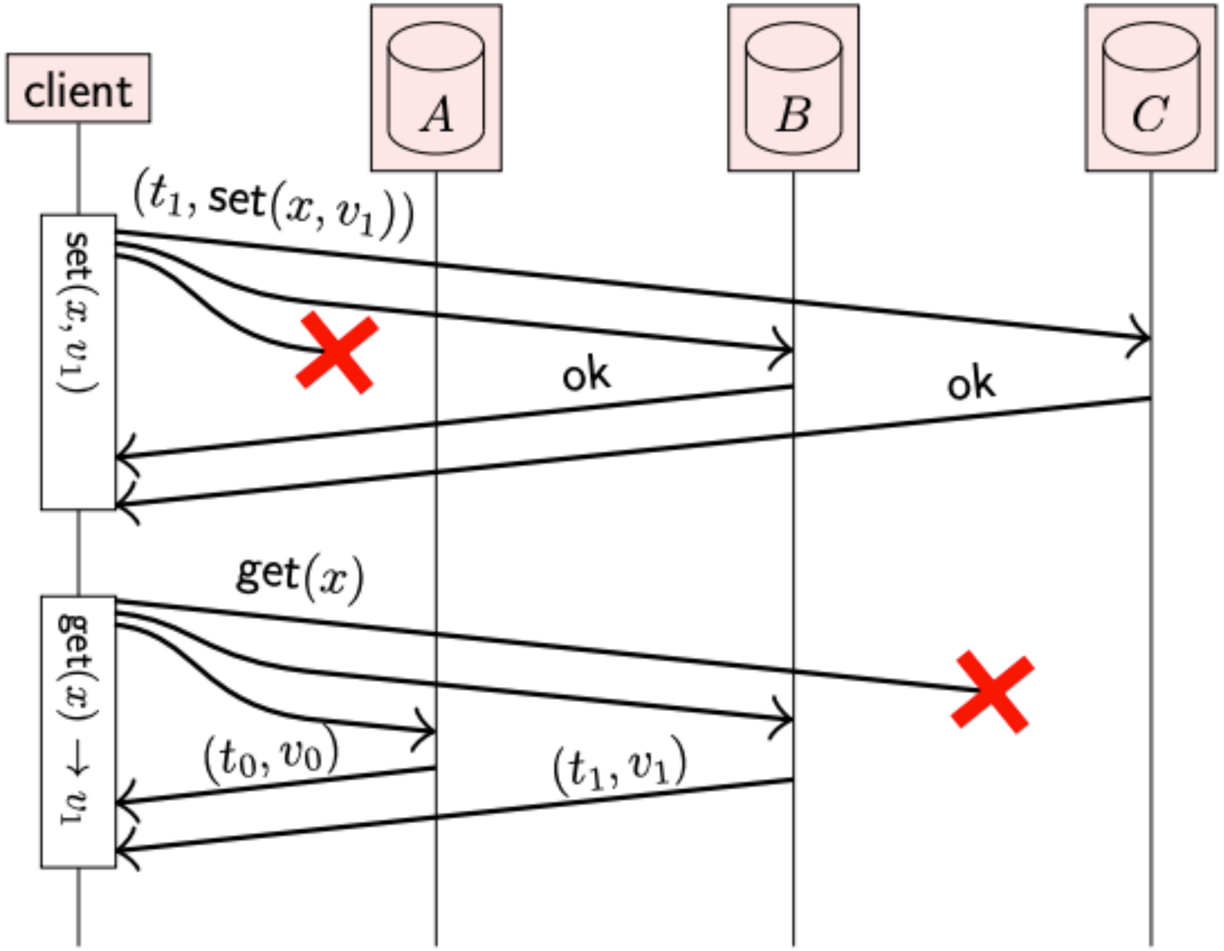
The strongest option: linearizability

- Informally: every operation takes effect atomically sometime after it started and before it finished
- All operations behave as if executed on a single copy of the data (even if there are in fact multiple replicas)
- Consequence: every operation returns an "**up-to-date**" value, a.k.a. "strong consistency"
- Not just in distributed systems, also in shared-memory concurrency (memory on multi-core CPUs is not linearizable by default!)

LINEARIZABILITY

Read-after-Write consistency

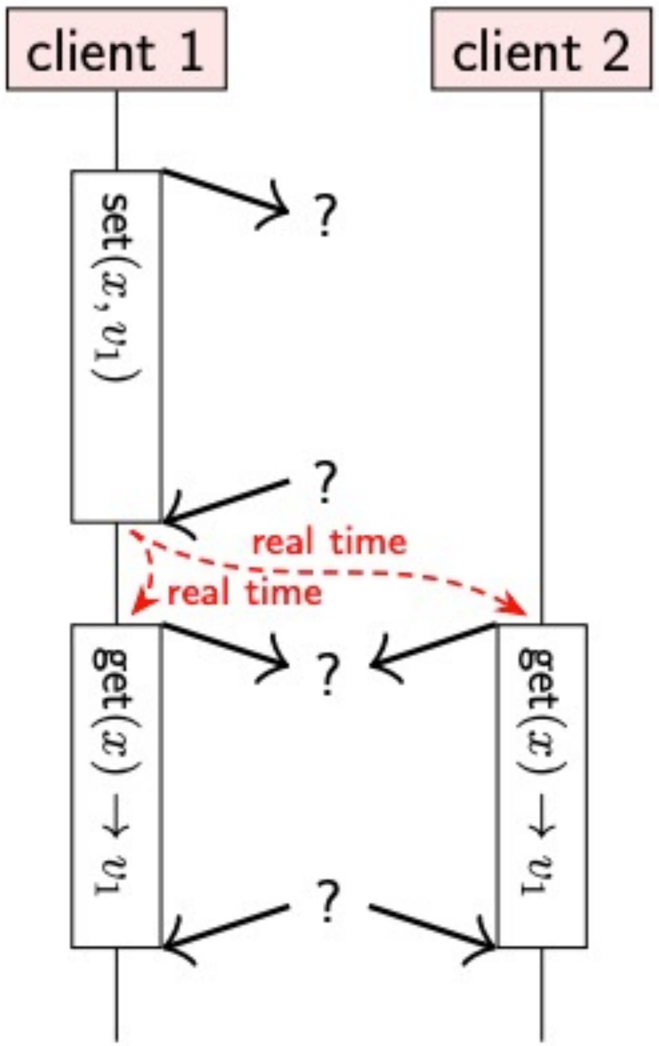
- The client’s view of a get/set operation as a rectangle covering the period of time from the start to finish of an operation.
- Inside the rectangle we write the effect of the operation: **set(x, v)** means updating the data item x to have the value v, and **get(x) → v** means a read of x that returns the value v.



LINEARIZABILITY

Read-after-Write consistency

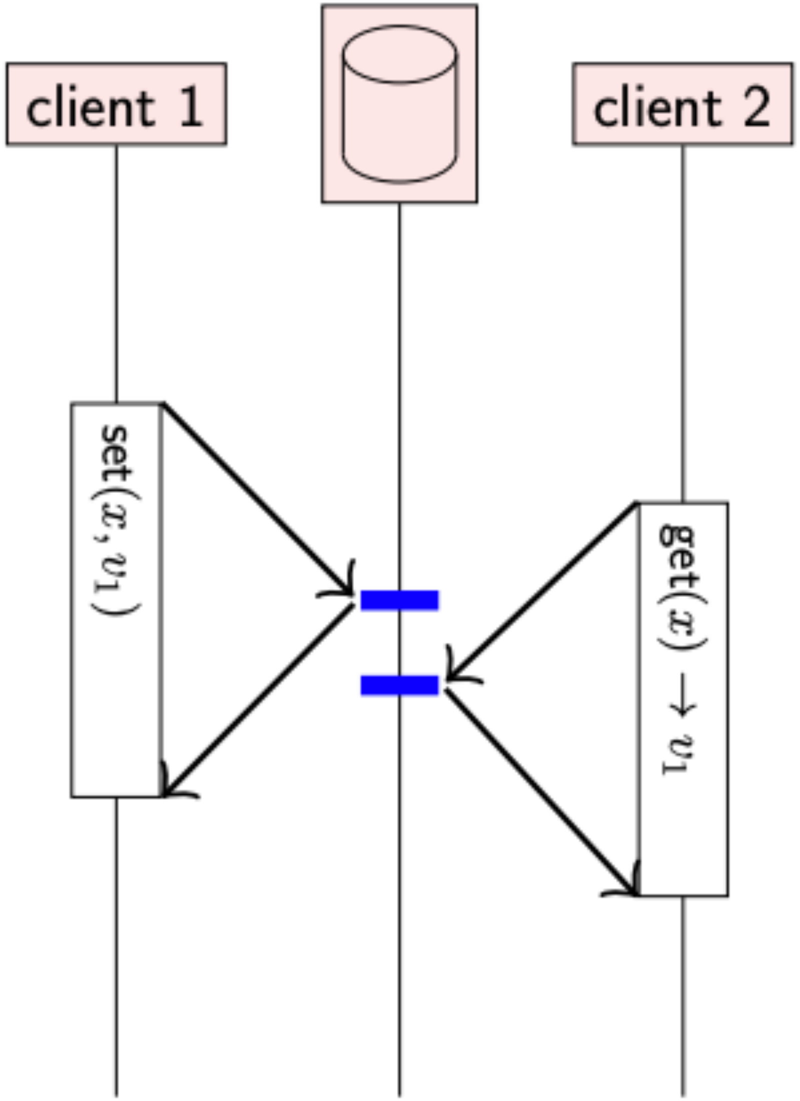
- Focus on client-observable behavior: when and what an operation returns
- Ignore how the replication system is implemented internally
- Did operation A finish before operation B started?
- Even if the operations are on different nodes?
- This is **not happens-before**: we want client 2 to read value written by client 1, even if the clients have not communicated!



LINEARIZABILITY

Read-after-Write consistency

- Client 2’s get operation overlaps in time with client 1’s set operation
- Maybe the set operation takes effect first?
- Maybe the get operation is executed first?
- Either outcome is fine in this case



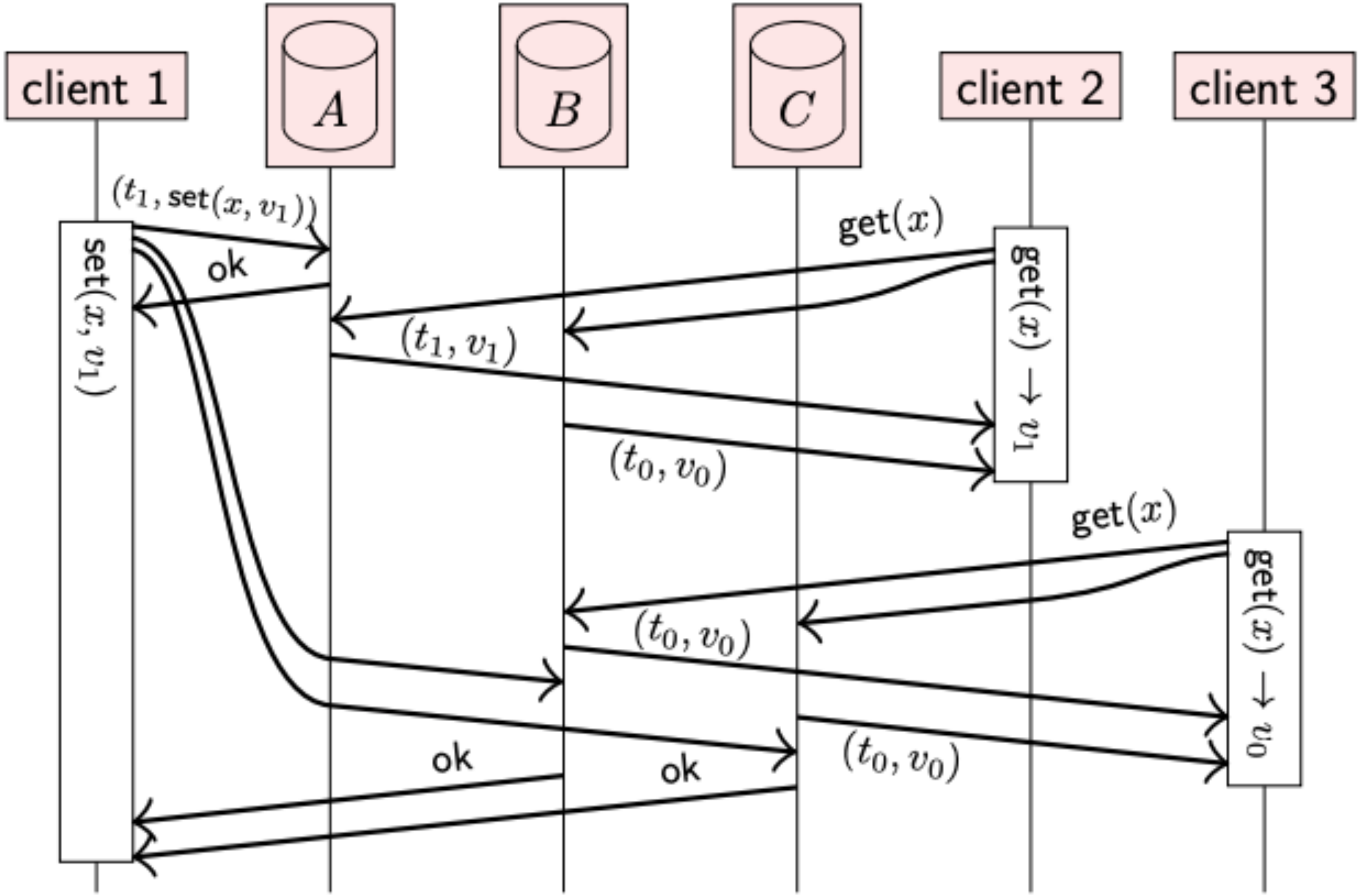
NON-LINEARIZABILITY

Non-Linearizable despite quorum reads/writes

- Linearizability is not only about the relationship of a **get** operation to a prior **set** operation, but it can also *relate one **get** operation to another.*

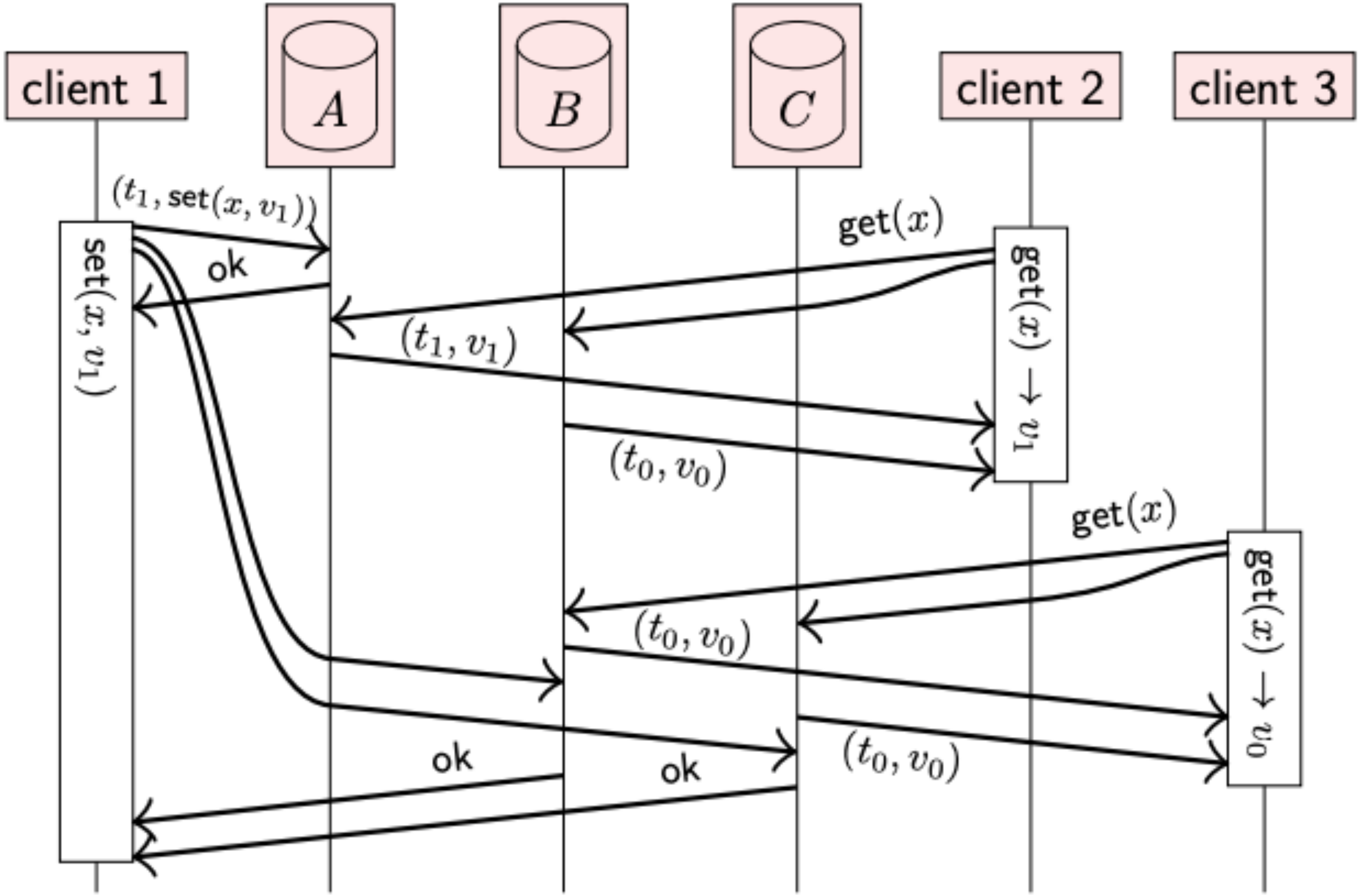
EXAMPLE: Client 1 sets x to v1,

Update to replica A happens quickly, while the updates to replicas B and C are delayed.

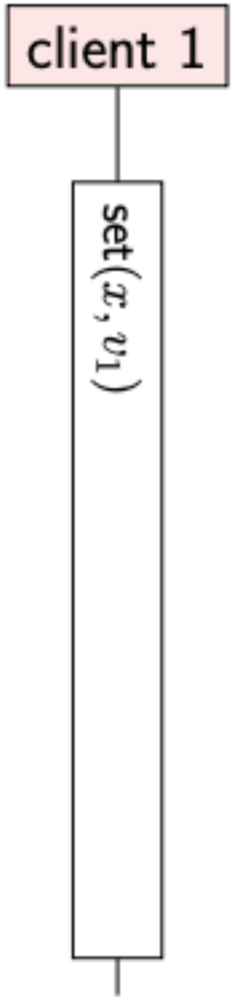


NON-LINEARIZABILITY

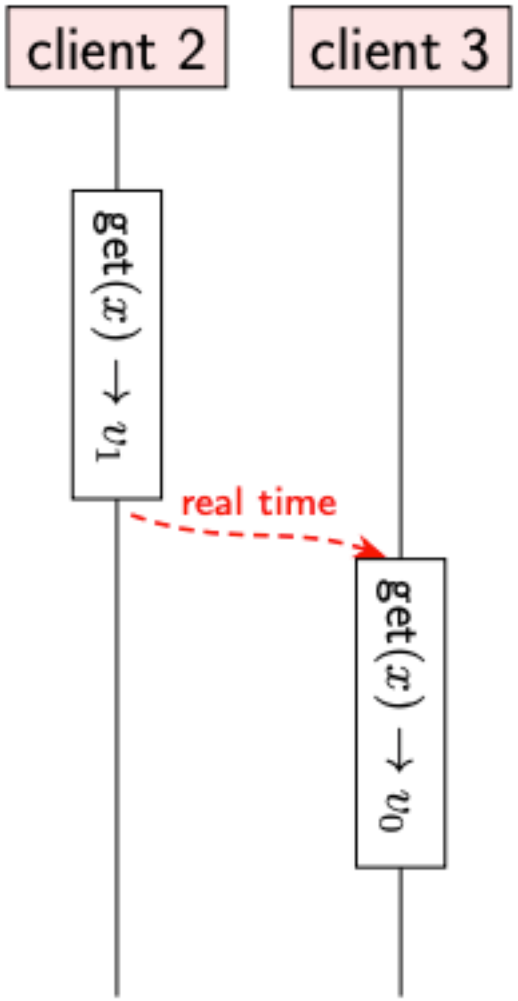
- Client 2 reads from a quorum of {A, B}, receives responses {v0, v1}, and determines v1 to be the newer value based on the attached timestamp.
- After client 2's read has finished, client 3 starts a read from a quorum of {B, C}, receives v0 from both replicas, and returns v0 (since it is not aware of v1).
- Client 3 observes an older value than client 2, even though the real-time order of operations would require client 3's read to return a value that is no older than client 2's result.
- ***This behavior is not allowed in a linearizable system***



NON-LINEARIZABILITY



- Client 2’s operation finishes before client 3’s operation starts
- **Linearizability therefore requires client 3’s operation to observe a state no older than client 2’s operation**
- This example violates linearizability because v0 is older than v1



ATTIYA, BAR-NOY, AND DOLEV (ABD) ALGORITHM

https://www.cs.huji.ac.il/course/2004/dist/p124-attiya.pdf

1. Write Operation:
   - When a process wants to perform a write operation on the shared register, it assigns a timestamp to the write operation and **broadcasts the write request** to all other processes.
   - Upon receiving a write request, **each process compares the timestamp** of the incoming write operation with the timestamp of the last known write operation.
   - If the incoming write operation has a **higher timestamp**, the process updates its local copy of the register with the new value and timestamp.

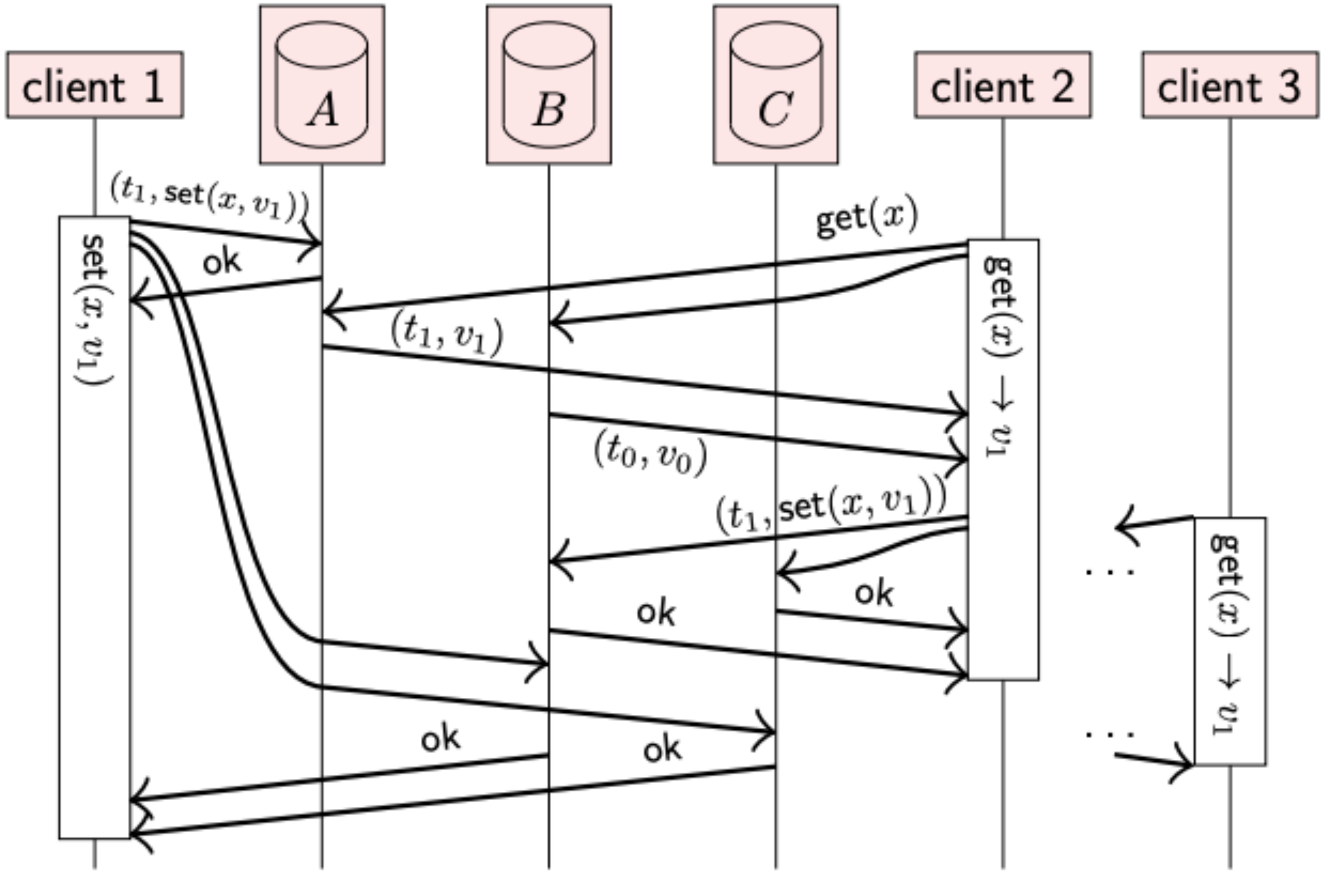
2. Read Operation:
   - When a process wants to perform a read operation on the shared register, it sends a read request to all other processes.
   - Each process responds to the read request by sending back its local copy of the register value along with its associated timestamp.
   - The process collecting the responses chooses the value with the **highest timestamp** and returns it as the result of the read operation.

ATTIYA, BAR-NOY, AND DOLEV (ABD) ALGORITHM

https://groups.csail.mit.edu/tds/papers/Attiya/TM-423.pdf

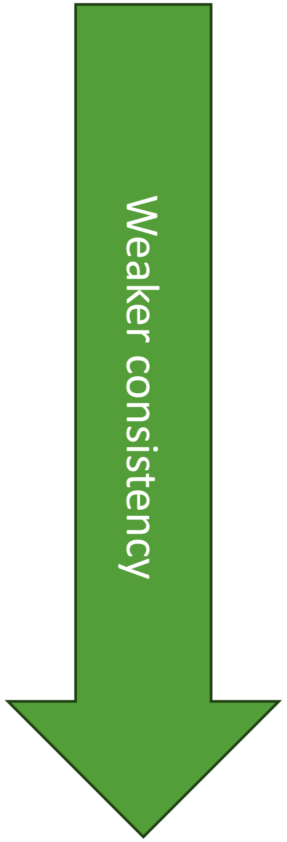
The ABD algorithm ensures consistency by enforcing a **total order** of write operations based on their timestamps. This total order allows processes to determine the latest value written to the shared register and ensures that all processes observe the same order of operations.

This ensures linearizability of get (quorum read) and set (blind write to quorum)



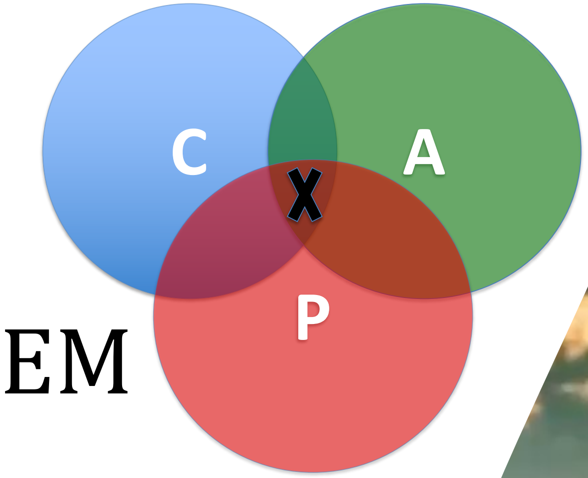
SUMMARY CONSISTENCY MODELS

- **Strong Consistency (Linearizability)**
  - Operations appear as if executed instantaneously and respect **real-time** ordering.
  - Banking systems (real-time, correctness-critical)
- **Sequential Consistency**
  - All operations appear in a **consistent global order** for all clients, but this order may not reflect real-time.
  - Collaborative document editing (global order without real-time constraints).
- **Read-Your-Writes Consistency**
  - Guarantees a client sees its own updates, but not necessarily updates from other clients immediately.
  - E-commerce shopping carts (personal updates visibility).
- **Eventual Consistency**
  - No guarantees about the order of operations; replicas only converge eventually.
  - DNS systems (availability and eventual convergence).



Weaker consistency

CS435 Distributed systems



CAP THEOREM

Dr. Basit Qureshi

PhD FHEA SMIEEE MACM

www.drbasit.org

The following slides are taken from Lecture Notes on Dist Systems by Prof. Douglas Thain @ University of Notre Dame

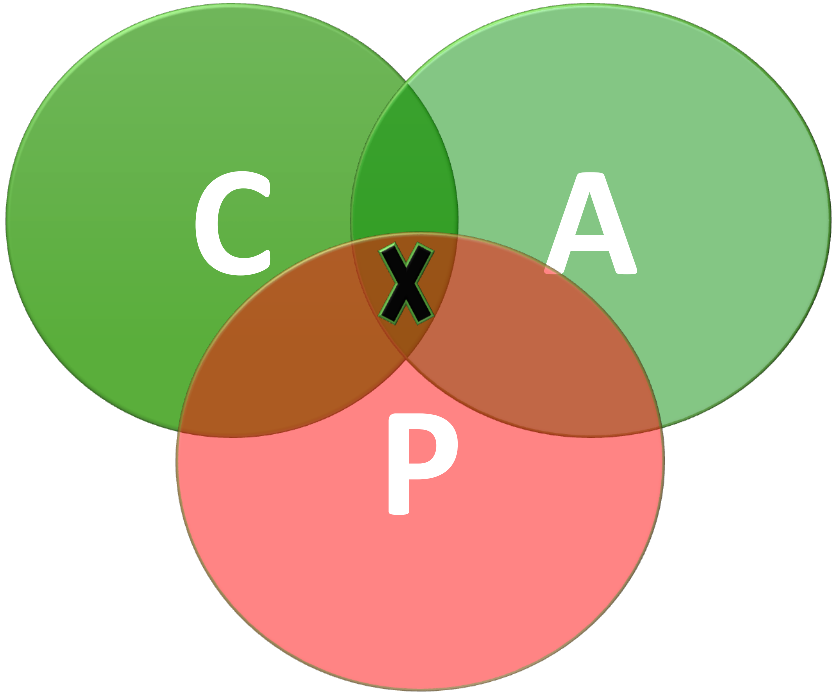
CAP THEOREM

- **Brewer's theorem**, is a fundamental principle in distributed computing that states that it is impossible for a distributed data system to simultaneously provide more than two out of three of the following guarantees:
  1. **Consistency (C)**: Every read receives the most recent write or an error. *In other words, all nodes in the system have the same data at the same time, regardless of which node they communicate with.*
  2. **Availability (A)**: Every request receives a response, even if some nodes are down or unreachable. *In other words, the system remains operational and responsive to client requests under all circumstances.*
  3. **Partition tolerance (P)**: The system continues to operate despite network partitions that may cause some nodes to be unreachable by others. *In other words, the system remains functional and maintains its guarantees even if there are network failures or splits.*

In other words! ***When there is a network partition, you cannot guarantee both availability & consistency***

CONSISTENCY OR AVAILABILITY

- Consistency and Availability is not “binary” decision
- **AP** systems relax consistency in favor of availability – but are not inconsistent
- **CP** systems sacrifice availability for consistency- but are not unavailable
- This suggests both **AP** and **CP** systems can offer a degree of consistency, and availability, as well as partition tolerance



AP: BEST EFFORT CONSISTENCY

- Example:
  - Web Caching
  - DNS
- Trait:
  - Optimistic
  - Expiration/Time-to-live
  - Conflict resolution

CP: BEST EFFORT AVAILABILITY

- Example:
  - Majority protocols
  - Distributed Locking (Google Chubby Lock service)
- Trait:
  - Pessimistic locking
  - Make minority partition unavailable

TYPES OF CONSISTENCY

- Strong Consistency
  - After the update completes, **any subsequent access** will return the **same** updated value.
- Weak Consistency
  - It is **not guaranteed** that subsequent accesses will return the updated value.
- **Eventual Consistency**
  - Specific form of weak consistency
  - It is guaranteed that if **no new updates** are made to object, **eventually** all accesses will return the last updated value (e.g., *propagate updates to replicas in a lazy fashion*)

Consistency

EVENTUAL CONSISTENCY VARIATIONS

- Causal consistency
  - Processes that have causal relationship will see consistent data
- Read-your-write consistency
  - A process always accesses the data item after it’s update operation and never sees an older value
- Session consistency
  - As long as session exists, system guarantees read-your-write consistency
  - Guarantees do not overlap sessions

EVENTUAL CONSISTENCY VARIATIONS

- Monotonic read consistency
  - If a process has seen a particular value of data item, any subsequent processes will never return any previous values
- Monotonic write consistency
  - The system guarantees to serialize the writes by the *same* process
- In practice
  - A number of these properties can be combined
  - Monotonic reads and read-your-writes are most desirable

EVENTUAL CONSISTENCY - A FACEBOOK EXAMPLE

- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but finds:

    - **Nothing is there!**

EVENTUAL CONSISTENCY - A FACEBOOK EXAMPLE

- Bob tells Alice to wait a bit and check out later
- Alice waits for a minute or so and checks back:

  - **She finds the story Bob shared with her!**

Wall
facebook
TIME TO UPDATE

EVENTUAL CONSISTENCY- A FACEBOOK EXAMPLE

- Reason: it is possible because Facebook uses an **eventual consistent model**
- Why Facebook chooses eventual consistent model over the strong consistent one?
  - Facebook has more than 1 billion active users
  - It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
  - Eventual consistent model offers the option to **reduce the load and improve availability**

EVENTUAL CONSISTENCY- A DROPBOX EXAMPLE

- Dropbox enabled immediate consistency via synchronization in many cases.
- However, what happens in case of a network partition?

www.bigstock.com · 30744092

EVENTUAL CONSISTENCY- A DROPBOX EXAMPLE

- Let’s do a simple experiment here:
  - Open a file in your drop box
  - Disable your network connection (e.g., WiFi, 4G)
  - Try to edit the file in the drop box: can you do that?
  - Re-enable your network connection: what happens to your dropbox folder?

EVENTUAL CONSISTENCY- A DROPBOX EXAMPLE

- Dropbox embraces eventual consistency:
  - Immediate consistency is impossible in case of a network partition
  - Users will feel bad if their word documents freeze each time they hit Ctrl+S , simply due to the large latency to update all devices across WAN
  - Dropbox is oriented to **personal syncing**, not on collaboration, so it is not a real limitation.

EVENTUAL CONSISTENCY- AN ATM EXAMPLE

- In design of automated teller machine (ATM):
  - Strong consistency appear to be a nature choice
  - However, in practice, **A beats C**
  - Higher availability means **higher revenue**
  - ATM will allow you to withdraw money *even if the machine is partitioned from the network*
  - However, it puts **a limit** on the amount of withdraw (e.g., $200)
  - The bank might also charge you a fee when a overdraft happens

DYNAMIC TRADEOFF BETWEEN **C** AND **A**

- An airline reservation system:
  - When most of seats are available: it is ok to rely on somewhat out-of-date data, **availability** is more critical
  - When the plane is close to be filled: it needs more accurate data to ensure the plane is not overbooked, **consistency** is more critical
- Neither strong consistency nor guaranteed availability, but it may significantly increase the tolerance of network disruption

DISCUSSION

- In an e-commercial system (e.g., Amazon, e-Bay, etc), what are the trade-offs between consistency and availability you can think of? What is your strategy?
- Things you might want to consider:
  - Different types of data (e.g., shopping cart, billing, inventory, etc.)
    - Shopping cart: User needs to see accurate data.
      - An outdated cart data can frustrate users. Use Eventual consistency
    - Billing: Consistent data to avoid over-charging.
      - Availability is less important than consistency. Strong consistency
    - Inventory: User needs to see products even some of them are unavailable (stale data).
      - Availability is important. Eventual Consistency
  - Different types of operations (e.g., query, purchase, etc.)
    - Querying: Queries should always respond, even if some replicas are temporarily unavailable.
      - Prioritize availability.
    - Checkout: The purchase flow must maintain data integrity
      - It is acceptable to block or retry during network partitions to ensure strong consistency.

DISCUSSION

- Different types of services (e.g., distributed lock, DNS, etc.)
  - Dist. locks on Inventory control:
    - Ensure mutual exclusion for critical sections like stock updates. Availability can be sacrificed during network partitions
  - DNS lookups:
    - Prioritize availability, as downtime disrupts all services. Use eventually consistent DNS caching (Stale data)
- Different groups of users (e.g., users in different geographic areas, etc.)
  - Geographically Distributed Users: Users in distant regions might see slightly stale data
    - Improve availability. Implement caching on the edge.
  - High-value customers:
    - Require stricter consistency guarantees to ensure a premium experience

PARTITIONING EXAMPLES

1. Data Partitioning
2. Operational Partitioning
3. Functional Partitioning
4. User Partitioning
5. Hierarchical Partitioning

Partitioning

Caused by the **possibility of failure** in distributed systems

High availability -> replicate data -> consistency problem

PARTITIONING EXAMPLES

1. Data Partitioning

- Different data may require different consistency and availability
- Example:
  - Shopping cart: high availability, responsive, can sometimes suffer anomalies
  - Product information need to be available, slight variation in inventory is sufferable
  - Checkout, billing, shipping records must be consistent
- YouTube:
  - User subscriptions: Strong Consistency, Medium Availability
  - Video Content: Eventual Consistency, High Availability
  - View Counts: Eventual Consistency, High Availability
  - Payment info: Strong Consistency, Medium Availability

PARTITIONING EXAMPLES

2. Operational Partitioning

- Each operation may require different balance between consistency and availability
- Example:
  - Reads: high availability; e.g.., "query"
  - Writes: high consistency, lock when writing; e.g., "purchase"

  - Online banking, Cloud storage, Messaging apps, E-commerce platforms etc.
  - **WhatsApp**
    - **Sending a message**: Must be eventually consistent across devices
    - **Delivery/read receipts**: Can tolerate delay
    - **Account registration**: Requires strong consistency

PARTITIONING EXAMPLES

3. Functional Partitioning

- System consists of sub-services
- Different sub-services provide different balances
- Example: A comprehensive distributed system
  - Distributed lock service (e.g., Chubby) :
    - Strong consistency
  - DNS service:
    - High availability

- **Facebook (Meta) Sub-services:**
  - **News Feed Service**: Aggregates and ranks content.
  - **Messaging Service**: Handles real-time communication (e.g., Messenger).
  - **Ads Service**: Manages ad targeting and billing.
  - **Notification Service**: Pushes updates and alerts.
  - **Profile Service**: Manages user data and privacy.

PARTITIONING EXAMPLES

4. User Partitioning

- Try to keep related data close together to assure better performance
- Example: Craglist
  - Might want to divide its service into several data centers, e.g., east coast and west coast
    - Users get high performance (e.g., high availability and good consistency) *if they query servers closest to them*
    - Poorer performance if a New York user query Craglist in San Francisco

  - **Gmail**: User mailboxes are partitioned by user, keeping all emails and settings for one user close together.
  - **Spotify**: User playlists, preferences, and playback history are stored together to ensure smooth performance when browsing or playing music.

PARTITIONING EXAMPLES

5. Hierarchical Partitioning

- Large global service with local “extensions”
- Different location in hierarchy may use different consistency
- Example:
  - **Amazon.com (amazon.sa)**
  - Local servers (better connected) guarantee more consistency and availability
  - Global servers has more partition and relax one of the requirement

  - **Netflix**?
  - User playback progress and subscription state need **strong consistency**.
  - Recommendations and viewing stats can tolerate **eventual consistency**

WHAT IF THERE ARE NO PARTITIONS?

- Basic idea:
  - Availability and latency are arguably **the same thing**: unavailable -> extreme high latency
  - Achieving different levels of consistency/availability takes different amount of time

Tradeoff between **Consistency** and **Latency**

CAP -> PACELC

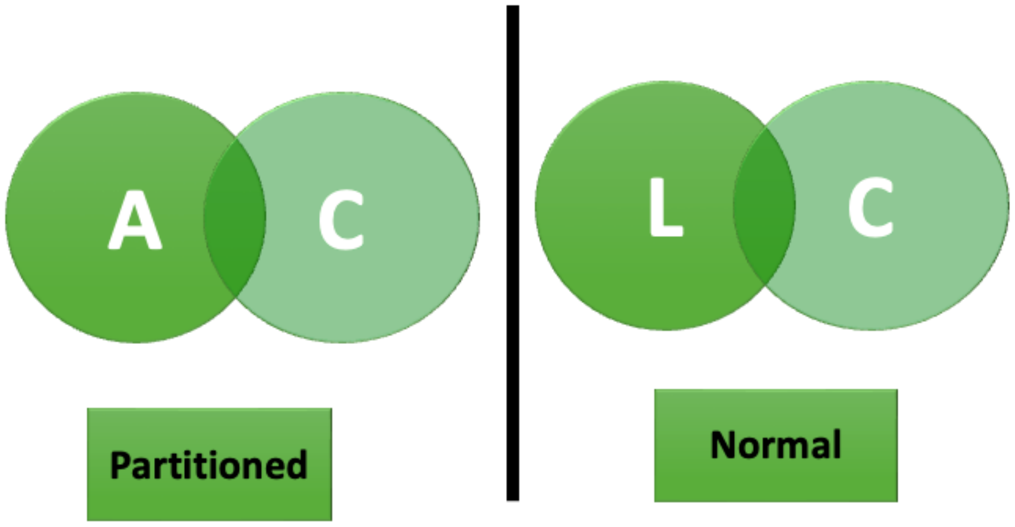
PACELC an extension of CAP Theorem?

Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." Computer-IEEE Computer Magazine 45.2 (2012): 37.

P: Partition Tolerance
A: Availability
C: Consistency
E: Else
L: Latency
C: Consistency

- If there is a **partition (P)**, how does the system trade off **availability and consistency (A and C)**; **else (E)**, when the system is running normally in the absence of partitions, how does the system trade off **latency (L) and consistency (C)**?

PACELC



During a Partition (P):

- A distributed system must choose between:
  - **Availability (A)**: Ensuring the system continues to serve requests.
  - **Consistency (C)**: Ensuring the data remains consistent across replicas.
- This is the same trade-off described in the CAP theorem.

Else (E) (when there is no Partition):

- The system must choose between:
  - **Latency (L)**: Responding quickly to requests, even if it sacrifices some data consistency.
  - **Consistency (C)**: Maintaining strict consistency, which might introduce delays.

EXAMPLES

- **Amazon DynamoDB** (AELC):
  - During a partition, prioritizes **Availability** over **Consistency**.
  - Else, prioritizes **Latency** over strict **Consistency**, allowing eventual consistency for faster performance.
- **Google Spanner** (PCEC):
  - During a partition, prioritizes **Consistency** over **Availability**.
  - Else, prioritizes strict **Consistency** over low **Latency**, ensuring globally consistent reads/writes.
- **Cassandra** (AELC):
  - During a partition, prioritizes **Availability**.
  - Else, prioritizes **Latency**, offering tunable consistency levels.

PACELC

- PACELC gives a **more realistic** view of distributed systems. It's not just about what happens **during a failure**, but also how systems behave under **normal operation**—where **latency and consistency** are still a trade-off.