Fault-Tolerant Distributed Transactions on Blockchain

Toward Scalable Blockchain


Suyash Gupta


Jelle Hellings


Mohammad Sadoghi

UCDAVIS
UNIVERSITY OF CALIFORNIA

ExpoLab
Creativity Unfolded

McMaster University

Scalability versus Fully-Replicated Blockchains

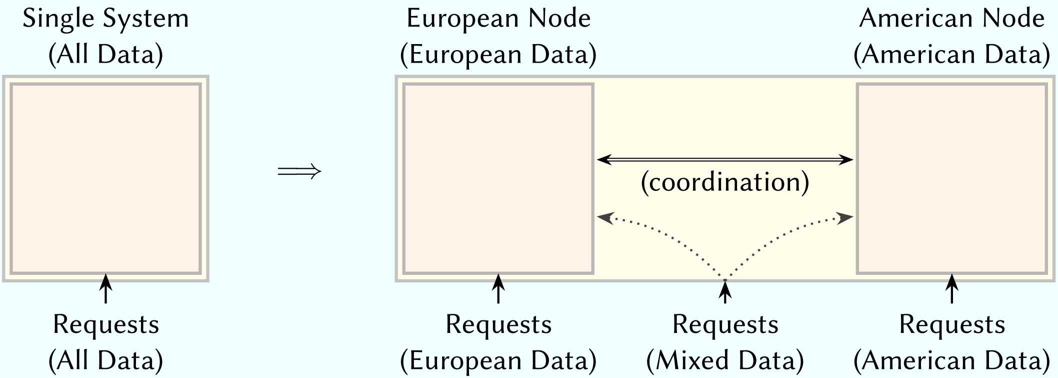
Scalability: adding resources $\Longrightarrow$ *adding performance.*

Scalability versus Fully-Replicated Blockchains

Scalability: adding resources $\Longrightarrow$ *adding performance.*

Full replication: adding resources (replicas) $\Longrightarrow$ less performance!

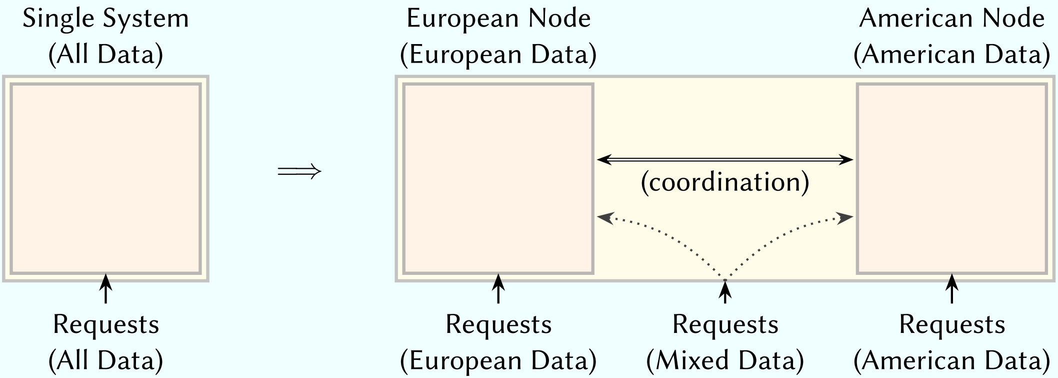
Distributed Systems: Scalability



Partition the system: More storage and *potentially* more performance.

Potentially *lower latencies* if data ends up closer to users.

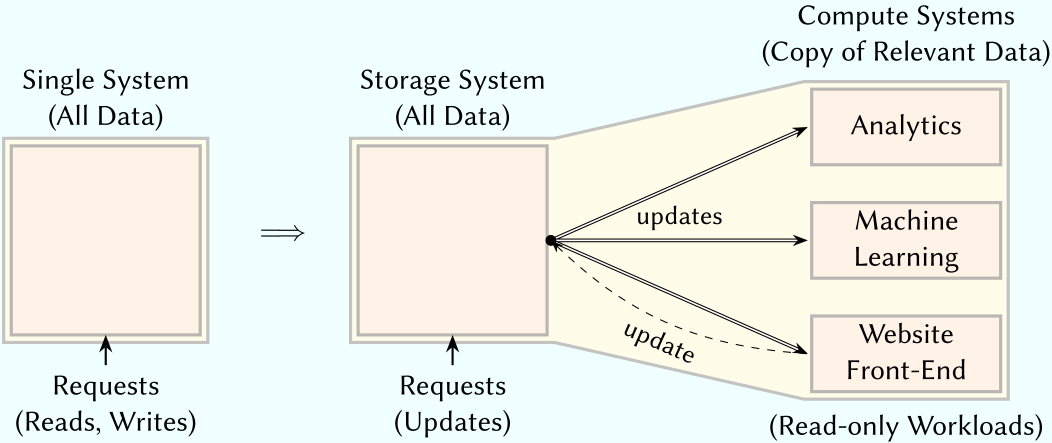
Distributed Systems: Scalability



Partition the system: More storage and *potentially* more performance.
Potentially *lower latencies* if data ends up closer to users.

Adding shards $\Longrightarrow$ adding throughput (parallel processing), adding storage.

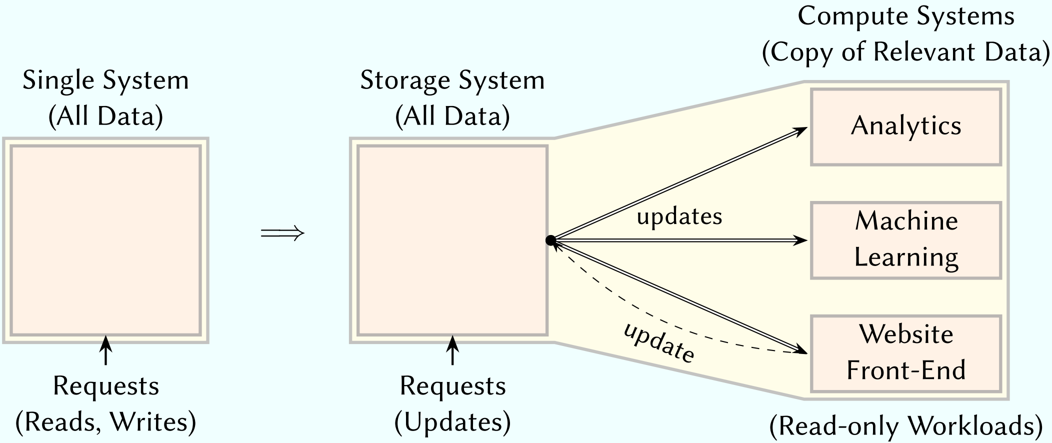
Distributed Systems: Specialization



Specialize the system: Different nodes have distinct tasks.

Specialized hardware and software *per* task.

Distributed Systems: Specialization



Specialize the system: Different nodes have distinct tasks.
Specialized hardware and software *per* task.

Specializing roles $\Longrightarrow$ adding throughput (parallel processing, specialized hardware, ...).

Central Ideas for Improvement

Reminder

We can make a resilient system that manages data: e.g., fully-replicated blockchains.

- **Role Specialization**: make the storage system a blockchain.
  Requires: *reliable read-only updates of the blockchain.*
  Permissionless blockchains: light clients!
- **Sharding**: make each shard an independent blockchain.
  Requires: *reliable communication between blockchains.*
  Permissionless blockchains: relays, atomic swaps!

Central Ideas for Improvement

Reminder

We can make a resilient system that manages data: e.g., fully-replicated blockchains.

- **Role Specialization**: make the storage system a blockchain.
  Requires: *reliable read-only updates of the blockchain.*
  Permissionless blockchains: light clients!
- **Sharding**: make each shard an independent blockchain.
  Requires: *reliable communication between blockchains.*
  Permissionless blockchains: relays, atomic swaps!

Consensus is of no use here if we want efficiency.

Reliable Read-Only Updates of Fault-Tolerant Clusters

Definition

Let $\mathcal{C}$ be a cluster deciding on a sequence of transactions $\mathcal{L}$ and L be a learner.

The *Byzantine learning problem* is the problem of sending $\mathcal{L}$ from $\mathcal{C}$ to L such that:

- the learner L will eventually *receive all* decided transactions;
- the learner L will *only receive* decided transactions.

Reliable Read-Only Updates of Fault-Tolerant Clusters

Definition

Let $\mathcal{C}$ be a cluster deciding on a sequence of transactions $\mathcal{L}$ and L be a learner.

The *Byzantine learning problem* is the problem of sending $\mathcal{L}$ from $\mathcal{C}$ to L such that:

- the learner L will eventually *receive all* decided transactions;
- the learner L will *only receive* decided transactions.

Practical requirements

- Minimizing overall communication.
- Load balancing among all replicas in $\mathcal{C}$.

Background: Information Dispersal Algorithms

Definition

Let v be a value with storage size $s = \|v\|$.
An *information dispersal algorithm* can encode v in $\mathbf{n}$ pieces v'
such that v can be *decoded* from every set of $\mathbf{n} - \mathbf{f}$ such pieces.

Theorem (Rabin 1989)

The IDA algorithm is an optimal information dispersal algorithm:

- *Each piece v' has size* $\left\lceil \frac{\|v\|}{\mathbf{n}-\mathbf{f}} \right\rceil$.
- *The $\mathbf{n} - \mathbf{f}$ pieces necessary for decoding have a total size of* $(\mathbf{n} - \mathbf{f}) \left\lceil \frac{\|v\|}{(\mathbf{n}-\mathbf{f})} \right\rceil \approx \|v\|$.

The Delayed-Replication Algorithm

Idea: $\mathcal{C}$ sends a ledger to learner L

The Delayed-Replication Algorithm

Idea: $\mathcal{C}$ sends a ledger to learner L

1. Partition the ledger in sequences S of $\mathbf{n}$ transactions.

The Delayed-Replication Algorithm

Idea: $\mathcal{C}$ sends a ledger to learner L

1. Partition the ledger in sequences S of $\mathbf{n}$ transactions.
2. Replica $\text{R}_i \in \mathcal{C}$ encodes S into the i-th IDA piece S_i.

The Delayed-Replication Algorithm

Idea: $\mathcal{C}$ sends a ledger to learner L

1. Partition the ledger in sequences S of **n** transactions.
2. Replica $\textsc{r}_i \in \mathcal{C}$ encodes S into the i-th IDA piece S_i.
3. Replica $\textsc{r}_i \in \mathcal{C}$ sends S_i with a checksum $C_i(S)$ of S to L.

The Delayed-Replication Algorithm

Idea: $\mathcal{C}$ sends a ledger to learner L

1. Partition the ledger in sequences S of $\mathbf{n}$ transactions.
2. Replica $\textsc{r}_i \in \mathcal{C}$ encodes S into the i-th IDA piece S_i.
3. Replica $\textsc{r}_i \in \mathcal{C}$ sends S_i with a checksum $C_i(S)$ of S to L.
4. L *receives at least* $\mathbf{n} - \mathbf{f}$ *distinct pieces and decodes* S.

The Delayed-Replication Algorithm

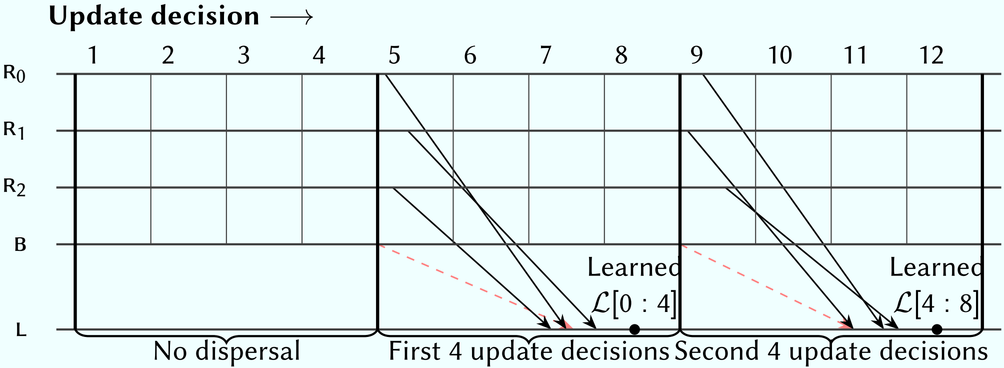
Idea: $\mathcal{C}$ sends a ledger to learner $\textsc{l}$

1. Partition the ledger in sequences S of $\mathbf{n}$ transactions.
2. Replica $\textsc{r}_i \in \mathcal{C}$ encodes S into the i-th IDA piece S_i.
3. Replica $\textsc{r}_i \in \mathcal{C}$ sends S_i with a checksum $C_i(S)$ of S to $\textsc{l}$.
4. $\textsc{l}$ *receives at least* $\mathbf{n} - \mathbf{f}$ *distinct pieces and decodes* S.

Observation ($\mathbf{n} > 2\mathbf{f}$)

- Replica $\textsc{r}_i$ sends at most $B = \left\lceil \frac{\|S\|}{\mathbf{n}-\mathbf{f}} \right\rceil + c \leq \frac{2\|S\|}{\mathbf{n}} + 1 + c = \mathcal{O}(\frac{\|S\|}{\mathbf{n}} + c)$ bytes.
- Learner $\textsc{l}$ receives at most $\mathbf{n} \cdot B = \mathcal{O}(\|S\| + c\mathbf{n})$ bytes.

Communication by the Delayed-Replication Algorithm



Decoding S Using Simple Checksums ($\mathbf{n} > 2\mathbf{f}$)

Decoding S Using Simple Checksums ($\mathbf{n} > 2\mathbf{f}$)

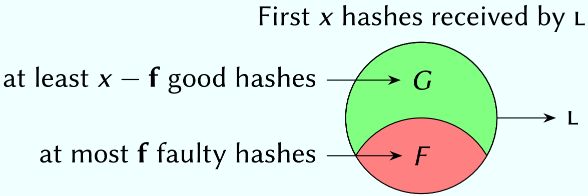
- Use checksums hash(S).

Decoding S Using Simple Checksums ($\mathbf{n} > 2\mathbf{f}$)

- Use checksums hash(S).
- The $\mathbf{n} - \mathbf{f}$ non-faulty replicas will provide correct *pieces*.

Decoding S Using Simple Checksums ($\mathbf{n} > 2\mathbf{f}$)

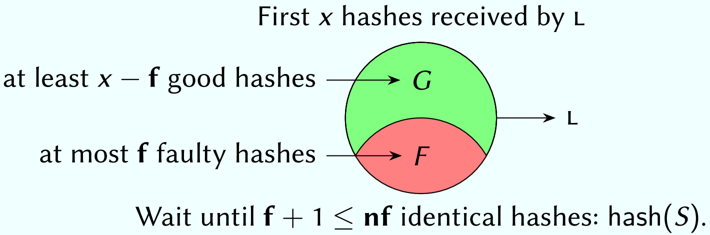
- Use checksums hash(S).
- The $\mathbf{n} - \mathbf{f}$ non-faulty replicas will provide correct *pieces*.
- At least $\mathbf{n} - \mathbf{f} > \mathbf{f}$ messages with correct *checksums*.



Wait until $\mathbf{f} + 1 \leq \mathbf{nf}$ identical hashes: hash(S).

Decoding S Using Simple Checksums ($\mathbf{n} > 2\mathbf{f}$)

- Use checksums hash(S).
- The $\mathbf{n} - \mathbf{f}$ non-faulty replicas will provide correct *pieces*.
- At least $\mathbf{n} - \mathbf{f} > \mathbf{f}$ messages with correct *checksums*.



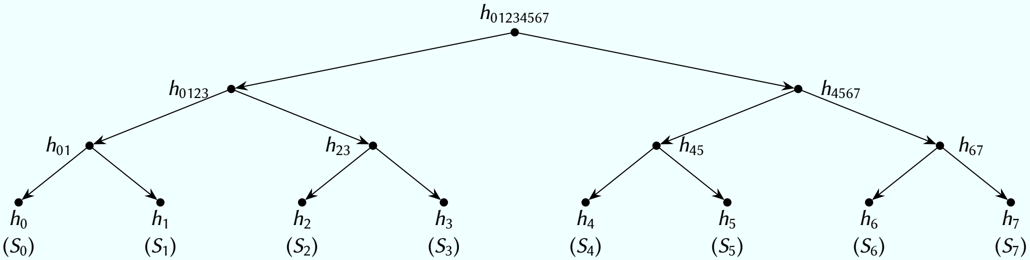
- Intensive for learners: one can choose $\mathbf{n} - \mathbf{f}$ out of $\mathbf{n}$ messages in $\binom{\mathbf{n}}{\mathbf{n}-\mathbf{f}}$ ways
  only one such choice is guaranteed to be correct!

Decoding S Using Tree Checksums

Use Merkle-trees to construct checksums

Consider 8 replicas and a sequence S.

We construct the checksum $C_5(S)$ of S (used by R_5).



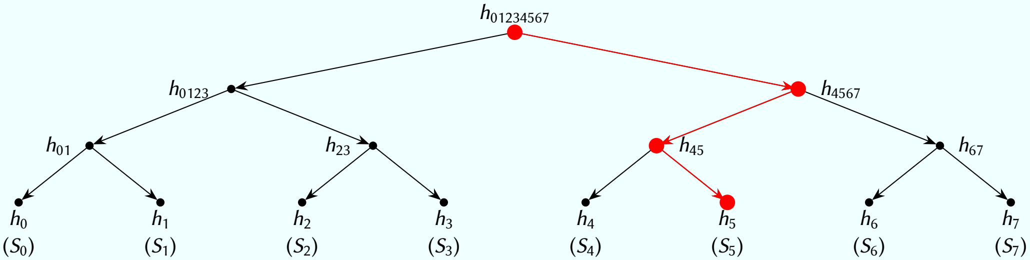
Construct a Merkle tree for pieces $S_0, \ldots, S_7$.

Decoding S Using Tree Checksums

Use Merkle-trees to construct checksums

Consider 8 replicas and a sequence S.

We construct the checksum $C_5(S)$ of S (used by R_5).



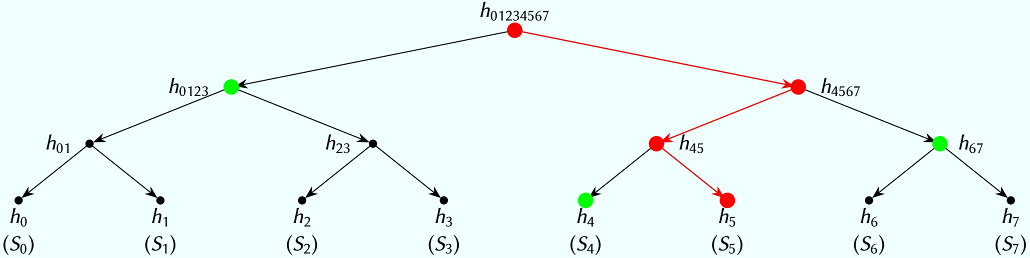
Determine the path from root to S_5.

Decoding S Using Tree Checksums

Use Merkle-trees to construct checksums

Consider 8 replicas and a sequence S.

We construct the checksum $C_5(S)$ of S (used by $\textsc{r}_5$).



Select *root* and *neighbors*: $C_5(S) = [h_4, h_{67}, h_{0123}, h_{01234567}]$.

Delayed-Replication: Main Result ($\mathbf{n} > 2\mathbf{f}$)

Theorem

Consider the learner L*, replica* R*, and decided transactions* $\mathcal{T}$*. The delayed-replication algorithm with tree checksums guarantees*

1. L *will learn* $\mathcal{T}$*;*
2. L *will receive at most* $|\mathcal{T}|$ *messages with a total size of* $\mathcal{O}(\|\mathcal{T}\| + |\mathcal{T}| \log \mathbf{n})$*;*
3. L *will only need at most* $\frac{|\mathcal{T}|}{\mathbf{n}}$ *decode steps;*
4. R *will sent at most* $\frac{|\mathcal{T}|}{\mathbf{n}}$ *messages to* L *of size* $\mathcal{O}(\frac{\|\mathcal{T}\|+|\mathcal{T}| \log \mathbf{n}}{\mathbf{n}})$*.*

Application: Scalable Storage for Resilient Systems

- Replicas typically only need the *current data* V to decide on future updates.
- Replicas only need the full ledger $\mathcal{L}$ for *recovery*.
- We can use *delayed-replication* to reduce the data each replica has to store.

Theorem

The storage cost per replica can be reduced from

$$\mathcal{O}(\|\mathcal{L}\| + \|V\|) \quad \textit{to} \quad \mathcal{O}(\frac{\|\mathcal{L}\|}{\mathbf{n}-\mathbf{f}} + \frac{|\mathcal{L}|}{\mathbf{n}}\log(\mathbf{n}) + \|V\|).$$

Reliable Communication between Fault-Tolerant Clusters

Definition

Let $\mathcal{C}_1, \mathcal{C}_2$ be two clusters, both having non-faulty replicas.

The *cluster-sending problem* is the problem of sending a value v from $\mathcal{C}_1$ to $\mathcal{C}_2$ such that:

1. non-faulty replicas in $\mathcal{C}_2$ *receive* v;
2. non-faulty replicas in $\mathcal{C}_1$ *confirm* that v was received by the non-faulty replicas in $\mathcal{C}_2$;
3. replicas in $\mathcal{C}_2$ only receive v if all non-faulty replicas in $\mathcal{C}_1$ *agree* upon sending v.

Reliable Communication between Fault-Tolerant Clusters

Definition

Let $\mathcal{C}_1, \mathcal{C}_2$ be two clusters, both having non-faulty replicas.

The *cluster-sending problem* is the problem of sending a value v from $\mathcal{C}_1$ to $\mathcal{C}_2$ such that:

1. non-faulty replicas in $\mathcal{C}_2$ *receive* v;
2. non-faulty replicas in $\mathcal{C}_1$ *confirm* that v was received by the non-faulty replicas in $\mathcal{C}_2$;
3. replicas in $\mathcal{C}_2$ only receive v if all non-faulty replicas in $\mathcal{C}_1$ *agree* upon sending v.

Informal Definition

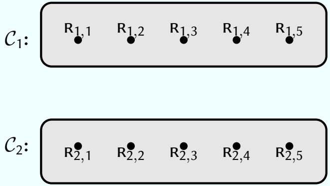
Successfully sending a value v from a cluster $\mathcal{C}_1$ to a $\mathcal{C}_2$ without any faulty replicas being able to *disrupt sending* or send *alternative forged values*.

Basic Cluster-Sending via Broadcasting

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica in $\mathcal{C}_1$ has a *certificate* $\text{cert}(v, \mathcal{C}_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

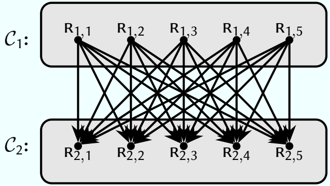


Basic Cluster-Sending via Broadcasting

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica in $\mathcal{C}_1$ has a *certificate* $\text{cert}(v, \mathcal{C}_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



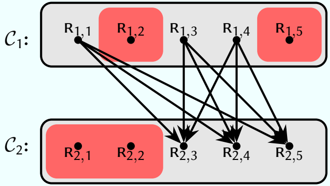
Broadcast: every replica in $\mathcal{C}_1$ sends pairs $(v, \text{cert}(v, \mathcal{C}_1))$ to every replica in $\mathcal{C}_2$.

Basic Cluster-Sending via Broadcasting

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica in $\mathcal{C}_1$ has a *certificate* $\text{cert}(v, \mathcal{C}_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



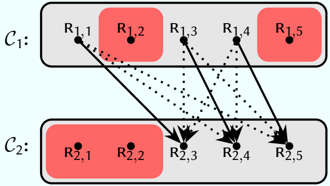
Faulty replicas can *fail* to send (in $\mathcal{C}_1$) or to receive (in $\mathcal{C}_2$).

Basic Cluster-Sending via Broadcasting

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica in $\mathcal{C}_1$ has a *certificate* $\text{cert}(v, \mathcal{C}_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



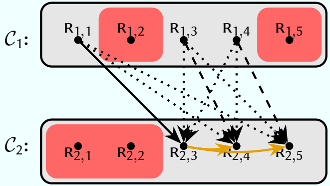
Non-faulty replicas in $\mathcal{C}_2$ only need at-least one message $(v, \text{cert}(v, \mathcal{C}_1))$.

Basic Cluster-Sending via Broadcasting

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica in $\mathcal{C}_1$ has a *certificate* $\text{cert}(v, \mathcal{C}_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



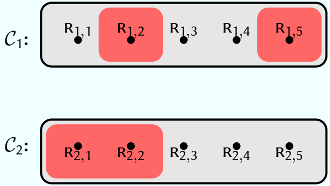
Replicas in $\mathcal{C}_2$ can redistribute $(v, \text{cert}(v, \mathcal{C}_1))$.

Basic Cluster-Sending via Broadcasting

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica in $\mathcal{C}_1$ has a *certificate* $\text{cert}(v, \mathcal{C}_1)$ that proves agreement.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



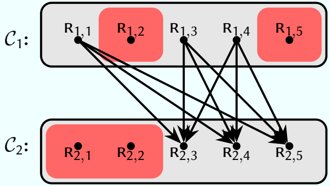
With certificates: a *single* message between non-faulty sender and receiver is sufficient!

Basic Cluster-Sending via Broadcasting (Without Certificates)

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica $\textsc{r} \in \mathcal{C}_1$ can only *claim* agreement via a digital signature $\mathrm{cert}(v, \textsc{r})$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.

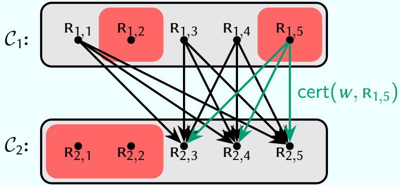


Basic Cluster-Sending via Broadcasting (Without Certificates)

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica $\textsc{r} \in \mathcal{C}_1$ can only *claim* agreement via a digital signature $\text{cert}(v, \textsc{r})$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



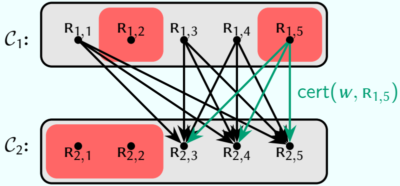
Faulty replicas can *lie* and send $\text{cert}(w, \textsc{r})$ without agreement on w.

Basic Cluster-Sending via Broadcasting (Without Certificates)

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica $\textsc{r} \in \mathcal{C}_1$ can only *claim* agreement via a digital signature $\mathrm{cert}(v, \textsc{r})$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



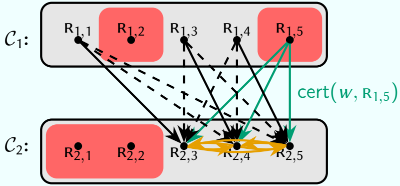
Claims from *three* distinct replicas in $\mathcal{C}_1$: at-least one from a non-faulty replica.

Basic Cluster-Sending via Broadcasting (Without Certificates)

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica $\textsc{r} \in \mathcal{C}_1$ can only *claim* agreement via a digital signature $\text{cert}(v, \textsc{r})$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



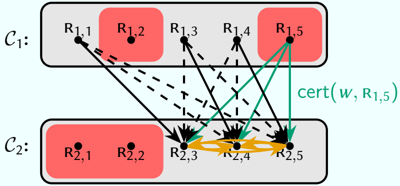
Replicas in $\mathcal{C}_2$ can redistribute $(v, \text{cert}(v, \textsc{r}))$.

Basic Cluster-Sending via Broadcasting (Without Certificates)

Goal: send a value v from cluster $\mathcal{C}_1$ to cluster $\mathcal{C}_2$.

Assumptions

- Every replica $\textsc{r} \in \mathcal{C}_1$ can only *claim* agreement via a digital signature $\text{cert}(v, \textsc{r})$.
- Communication is *reliable*.
- At-most *two* replicas faulty in each cluster.



Without certificates: *at least* $\mathbf{f}_{\mathcal{C}_1} + 1$ distinct received messages by non-faulty senders!

Efficient Cluster-Sending

Cluster-Sending via broadcasting: straightforward, *not efficient*:

- With certificates: $(\mathbf{f}_{\mathcal{C}_1} + 1)(\mathbf{f}_{\mathcal{C}_2} + 1) \approx \mathbf{f}_{\mathcal{C}_1} \times \mathbf{f}_{\mathcal{C}_2}$ messages.
- With claims: $(2\mathbf{f}_{\mathcal{C}_1} + 1)(\mathbf{f}_{\mathcal{C}_2} + 1) \approx 2\mathbf{f}_{\mathcal{C}_1} \times \mathbf{f}_{\mathcal{C}_2}$ messages.

Efficient Cluster-Sending

Cluster-Sending via broadcasting: straightforward, *not efficient*:

- With certificates: $(\mathbf{f}_{\mathcal{C}_1} + 1)(\mathbf{f}_{\mathcal{C}_2} + 1) \approx \mathbf{f}_{\mathcal{C}_1} \times \mathbf{f}_{\mathcal{C}_2}$ messages.
- With claims: $(2\mathbf{f}_{\mathcal{C}_1} + 1)(\mathbf{f}_{\mathcal{C}_2} + 1) \approx 2\mathbf{f}_{\mathcal{C}_1} \times \mathbf{f}_{\mathcal{C}_2}$ messages.

Local communication versus global communication

	Ping round-trip times (ms)						*Bandwidth (Mbit/s)*					
	OR	IA	Mont.	BE	TW	Syd.	OR	IA	Mont.	BE	TW	Syd.
Oregon	≤ 1	38	65	136	118	161	7998	669	371	194	188	136
Iowa		≤ 1	33	98	153	172		10004	752	243	144	120
Montreal			≤ 1	82	186	202			7977	283	111	102
Belgium				≤ 1	252	270				9728	79	66
Taiwan					≤ 1	137					7998	160
Sydney						≤ 1						7977

Goal: Minimize communication *between* clusters.

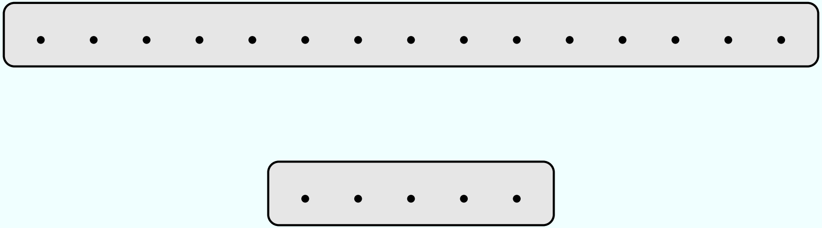
Towards a Lower-Bound for Cluster-Sending (Example)

$$\mathbf{n}_{\mathcal{C}_1} = 15 \qquad \mathbf{f}_{\mathcal{C}_1} = 7$$

$$\mathbf{n}_{\mathcal{C}_2} = 5 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



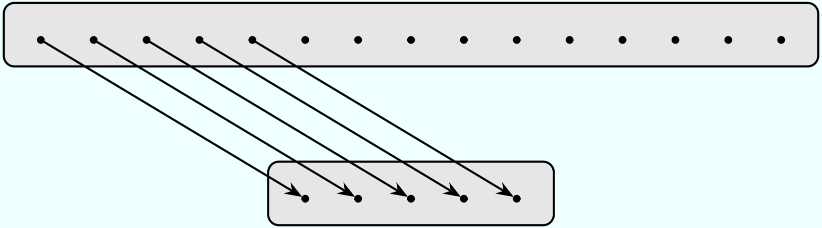
Towards a Lower-Bound for Cluster-Sending (Example)

$$\mathbf{n}_{\mathcal{C}_1} = 15 \qquad \mathbf{f}_{\mathcal{C}_1} = 7$$

$$\mathbf{n}_{\mathcal{C}_2} = 5 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



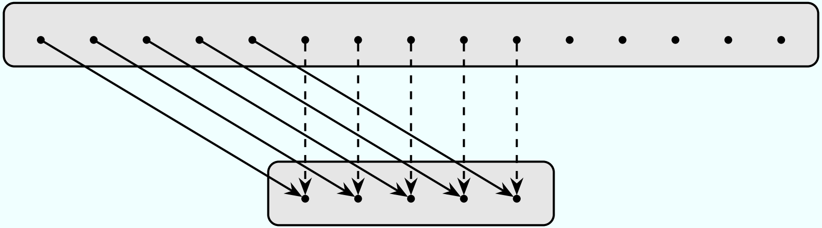
Minimize impact of faulty replicas: minimum number of messages per participant.

Towards a Lower-Bound for Cluster-Sending (Example)

$$\mathbf{n}_{\mathcal{C}_1} = 15 \qquad \mathbf{f}_{\mathcal{C}_1} = 7$$
$$\mathbf{n}_{\mathcal{C}_2} = 5 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



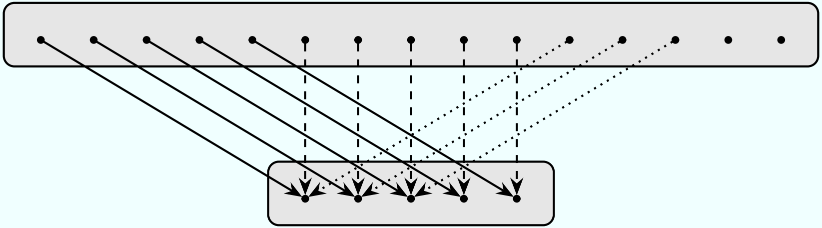
Minimize impact of faulty replicas: minimum number of messages per participant.

Towards a Lower-Bound for Cluster-Sending (Example)

$$\mathbf{n}_{\mathcal{C}_1} = 15 \qquad \mathbf{f}_{\mathcal{C}_1} = 7$$
$$\mathbf{n}_{\mathcal{C}_2} = 5 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



Minimize impact of faulty replicas: minimum number of messages per participant.

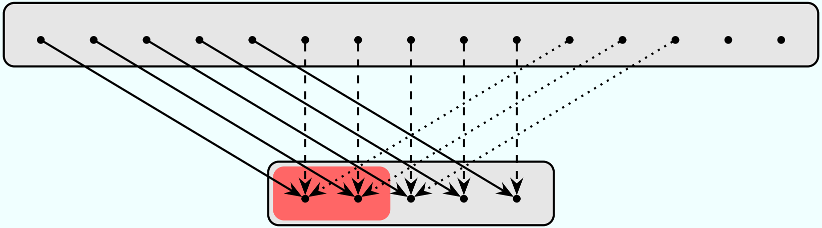
Towards a Lower-Bound for Cluster-Sending (Example)

$\mathbf{n}_{\mathcal{C}_1} = 15$ $\mathbf{f}_{\mathcal{C}_1} = 7$

$\mathbf{n}_{\mathcal{C}_2} = 5$ $\mathbf{f}_{\mathcal{C}_2} = 2$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



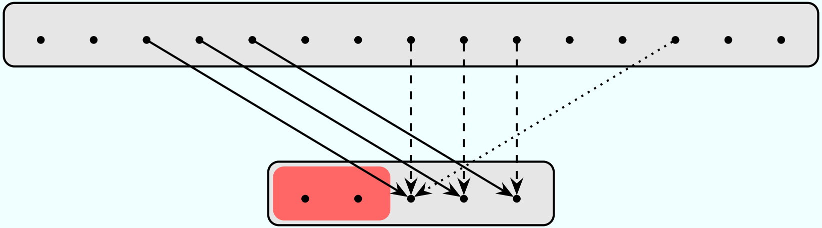
Any $\mathbf{f}_{\mathcal{C}_2}$ replicas in $\mathcal{C}_2$ can be faulty: top $\mathbf{f}_{\mathcal{C}_2}$ receivers receive at-least 6 messages.

Towards a Lower-Bound for Cluster-Sending (Example)

$$\mathbf{n}_{\mathcal{C}_1} = 15 \qquad \mathbf{f}_{\mathcal{C}_1} = 7$$
$$\mathbf{n}_{\mathcal{C}_2} = 5 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



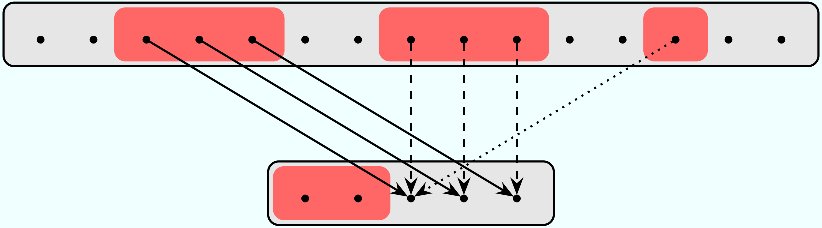
Towards a Lower-Bound for Cluster-Sending (Example)

$$\mathbf{n}_{\mathcal{C}_1} = 15 \qquad \mathbf{f}_{\mathcal{C}_1} = 7$$

$$\mathbf{n}_{\mathcal{C}_2} = 5 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



Only $\mathbf{f}_{\mathcal{C}_1}$ messages remaining, can all be sent by faulty replicas in $\mathcal{C}_1$.

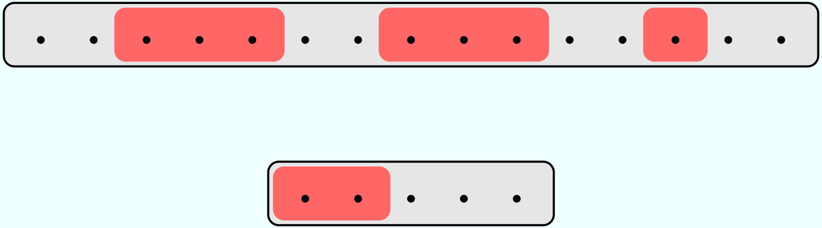
Towards a Lower-Bound for Cluster-Sending (Example)

$$\mathbf{n}_{\mathcal{C}_1} = 15 \qquad \mathbf{f}_{\mathcal{C}_1} = 7$$

$$\mathbf{n}_{\mathcal{C}_2} = 5 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

Proposition (assuming certificates)

Any correct algorithm needs to send at least 14 *messages*.



Lower-Bound for Cluster-Sending with Certificates

Basic Idea

- One message needs to be exchanged between a non-faulty sender and receiver.
- Have to deal with size imbalances between clusters.

Lower-Bound for Cluster-Sending with Certificates

Basic Idea

- One message needs to be exchanged between a non-faulty sender and receiver.
- Have to deal with size imbalances between clusters.

Theorem

Let $\mathcal{C}_1, \mathcal{C}_2$ be two clusters and let $\{i, j\} = \{1, 2\}$ such that $\mathbf{n}_{\mathcal{C}_i} \geq \mathbf{n}_{\mathcal{C}_j}$. Let

$$q_i = (\mathbf{f}_{\mathcal{C}_i} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_j},$$
$$r_i = (\mathbf{f}_{\mathcal{C}_i} + 1) \bmod \mathbf{nf}_{\mathcal{C}_j},$$
$$\sigma_i = q_i \mathbf{n}_{\mathcal{C}_j} + r_i + \mathbf{f}_{\mathcal{C}_j} \operatorname{sgn} r_i.$$

Any protocol that solves the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$ needs to exchange at least σ_i messages.

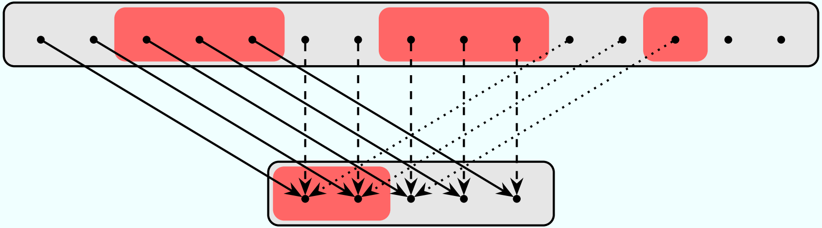
Lower-Bound for Cluster-Sending with Certificates (Example)

Theorem

Let $\mathcal{C}_1, \mathcal{C}_2$ *be two clusters and let*

$$q_1 = (\mathbf{f}_{\mathcal{C}_1} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_2} = 7 \operatorname{div} 3 = 2,$$
$$r_1 = (\mathbf{f}_{\mathcal{C}_1} + 1) \bmod \mathbf{nf}_{\mathcal{C}_2} = 7 \bmod 3 = 1,$$
$$\sigma_1 = q_1 \mathbf{n}_{\mathcal{C}_2} + r_1 + \mathbf{f}_{\mathcal{C}_2} \operatorname{sgn} r_1 = 2 \cdot 5 + 1 + 3 = 14.$$

Any protocol that solves the cluster-sending problem in which $\mathcal{C}_1$ *sends a value v to* $\mathcal{C}_2$ *needs to exchange at least* $\sigma_1 = 14$ *messages.*



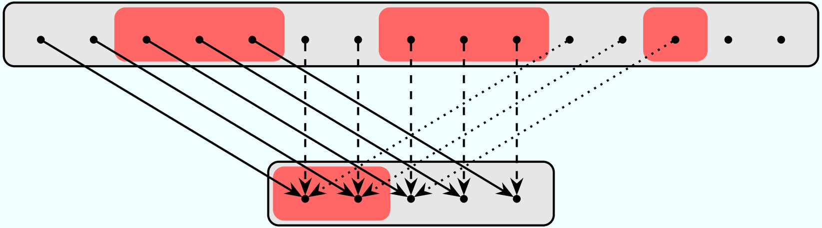
Lower-Bound for Cluster-Sending with Certificates (Example)

Theorem

Let $\mathcal{C}_1, \mathcal{C}_2$ be two clusters and let

$$\begin{aligned} q_1 &= (\mathbf{f}_{\mathcal{C}_1} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_2} = 7 \operatorname{div} 3 = 2, \\ r_1 &= (\mathbf{f}_{\mathcal{C}_1} + 1) \bmod \mathbf{nf}_{\mathcal{C}_2} = 7 \bmod 3 = 1, \\ \sigma_1 &= \boxed{q_1 \mathbf{n}_{\mathcal{C}_2}} + r_1 + \mathbf{f}_{\mathcal{C}_2} \operatorname{sgn} r_1 = \boxed{2 \cdot 5} + 1 + 3 = 14. \end{aligned}$$

Any protocol that solves the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$ needs to exchange at least $\sigma_1 = 14$ messages.



Lower-Bound for Cluster-Sending with Certificates (Example)

Theorem

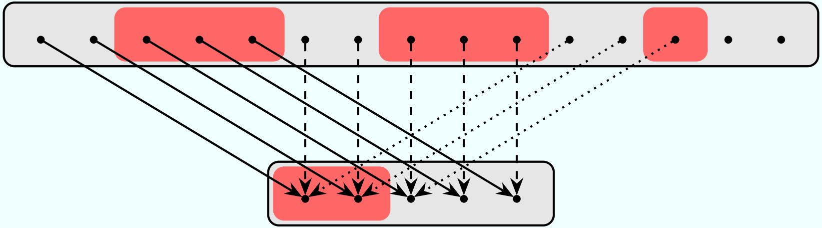
Let $\mathcal{C}_1, \mathcal{C}_2$ be two clusters and let

$$q_1 = (\mathbf{f}_{\mathcal{C}_1} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_2} = 7 \operatorname{div} 3 = 2,$$

$$r_1 = (\mathbf{f}_{\mathcal{C}_1} + 1) \bmod \mathbf{nf}_{\mathcal{C}_2} = 7 \bmod 3 = 1,$$

$$\sigma_1 = q_1 \mathbf{n}_{\mathcal{C}_2} + \boxed{r_1 + \mathbf{f}_{\mathcal{C}_2} \operatorname{sgn} r_1} = 2 \cdot 5 + \boxed{1 + 3} = 14.$$

Any protocol that solves the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$ needs to exchange at least $\sigma_1 = 14$ messages.



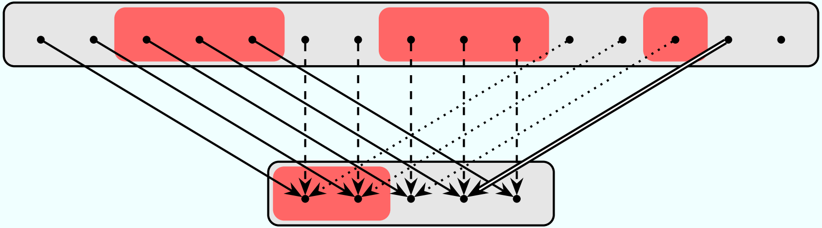
Lower-Bound for Cluster-Sending with Certificates (Example)

Theorem

Let $\mathcal{C}_1, \mathcal{C}_2$ be two clusters and let

$$q_1 = (\mathbf{f}_{\mathcal{C}_1} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_2} = 7 \operatorname{div} 3 = 2,$$
$$r_1 = (\mathbf{f}_{\mathcal{C}_1} + 1) \bmod \mathbf{nf}_{\mathcal{C}_2} = 7 \bmod 3 = 1,$$
$$\sigma_1 = q_1 \mathbf{n}_{\mathcal{C}_2} + r_1 + \mathbf{f}_{\mathcal{C}_2} \operatorname{sgn} r_1 = 2 \cdot 5 + 1 + 3 = 14.$$

Any protocol that solves the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$ needs to exchange at least $\sigma_1 = \boxed{14}$ messages.



Lower-Bound for Cluster-Sending with Claims

Basic Idea

- $\mathbf{f}_{\mathcal{C}_1} + 1$ message needs to be sent by distinct non-faulty senders to non-faulty receiver.
- Have to deal with size imbalances between clusters.

Theorem

Let $\mathcal{C}_1, \mathcal{C}_2$ be two clusters and let $\{i, j\} = \{1, 2\}$ such that $\mathbf{n}_{\mathcal{C}_i} \geq \mathbf{n}_{\mathcal{C}_j}$. Let

$$q_1 = (2\mathbf{f}_{\mathcal{C}_1} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_2}, \qquad q_2 = (\mathbf{f}_{\mathcal{C}_2} + 1) \operatorname{div} (\mathbf{nf}_{\mathcal{C}_1} - \mathbf{f}_{\mathcal{C}_1})$$

$$r_1 = (2\mathbf{f}_{\mathcal{C}_1} + 1) \bmod \mathbf{nf}_{\mathcal{C}_2}, \qquad r_2 = (\mathbf{f}_{\mathcal{C}_2} + 1) \bmod (\mathbf{nf}_{\mathcal{C}_1} - \mathbf{f}_{\mathcal{C}_1})$$

$$\tau_1 = q_1 \mathbf{n}_{\mathcal{C}_2} + r_1 + \mathbf{f}_{\mathcal{C}_2} \operatorname{sgn} r_1 \qquad \tau_2 = q_2 \mathbf{n}_{\mathcal{C}_1} + r_2 + 2\mathbf{f}_{\mathcal{C}_1} \operatorname{sgn} r_2.$$

Any protocol that solves the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$ needs to exchange at least τ_i messages.

Bijective Sending with Certificates

Assume $\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1 \leq \min(\mathbf{n}_{\mathcal{C}_1}, \mathbf{n}_{\mathcal{C}_2})$.

We have $\sigma_1 = \sigma_2 = \mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$.

Protocol for the sending cluster $\mathcal{C}_1$:

```
All replicas in 𝒢_{𝒞1} agree on v and construct cert(v, 𝒞1).
Choose replicas S1 ⊆ 𝒞1 and S2 ⊆ 𝒞2 with n_{S2} = n_{S1} = f_{𝒞1} + f_{𝒞2} + 1.
Choose a bijection b : S1 → S2.
for R1 ∈ S1 do
    R1 sends (v, cert(v, 𝒞1)) to b(R1).
```

Protocol for the receiving cluster $\mathcal{C}_2$:

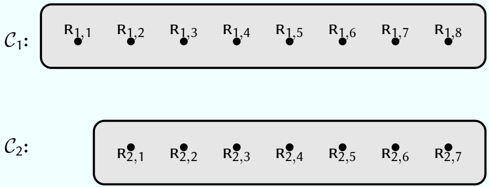
```
event R2 ∈ 𝒢_{𝒞2} receives (w, cert(w, 𝒞1)) from R1 ∈ 𝒞1 do
    Broadcast (w, cert(w, 𝒞1)) to all replicas in 𝒞2.
event R2′ ∈ 𝒢_{𝒞2} receives (w, cert(w, 𝒞1)) from R2 ∈ 𝒞2 do
    R2′ considers w received.
```

Bijective Sending with Certificates: Example

$$\mathbf{n}_{\mathcal{C}_1} = 8 \qquad \mathbf{f}_{\mathcal{C}_1} = 3$$

$$\mathbf{n}_{\mathcal{C}_2} = 7 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

$$\sigma_1 = 6.$$

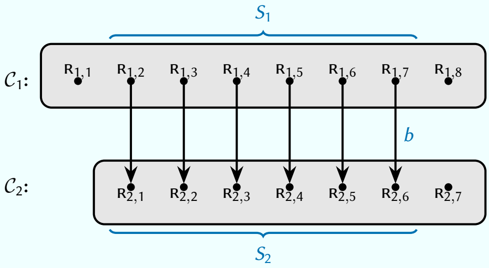


Bijective Sending with Certificates: Example

$$\mathbf{n}_{\mathcal{C}_1} = 8 \qquad \mathbf{f}_{\mathcal{C}_1} = 3$$

$$\mathbf{n}_{\mathcal{C}_2} = 7 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

$$\sigma_1 = 6.$$

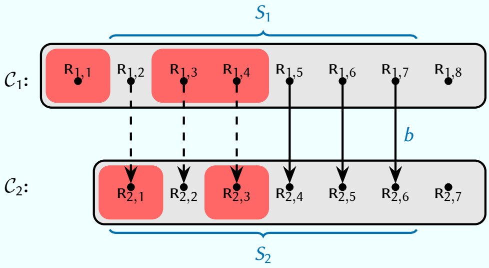


Bijective Sending with Certificates: Example

$$\mathbf{n}_{\mathcal{C}_1} = 8 \qquad \mathbf{f}_{\mathcal{C}_1} = 3$$

$$\mathbf{n}_{\mathcal{C}_2} = 7 \qquad \mathbf{f}_{\mathcal{C}_2} = 2$$

$$\sigma_1 = 6.$$



Bijective Sending with Claims

Assume $2\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1 \leq \min(\mathbf{n}_{\mathcal{C}_1}, \mathbf{n}_{\mathcal{C}_2})$.

We have $\tau_1 = \tau_2 = 2\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$.

Protocol for the sending cluster $\mathcal{C}_1$:

All replicas in $\mathcal{G}_{\mathcal{C}_1}$ agree on v.
Choose replicas $S_1 \subseteq \mathcal{C}_1$ and $S_2 \subseteq \mathcal{C}_2$ with $\mathbf{n}_{S_2} = \mathbf{n}_{S_1} = 2\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$.
Choose bijection $b : S_1 \to S_2$.
for $\textsc{r}_1 \in S_1$ **do**
    $\textsc{r}_1$ sends $(v, \text{cert}(v, \textsc{r}_1))$ to $b(\textsc{r}_1)$.

Protocol for the receiving cluster $\mathcal{C}_2$:

. . . .

Bijective Sending with Claims

Assume $2\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1 \leq \min(\mathbf{n}_{\mathcal{C}_1}, \mathbf{n}_{\mathcal{C}_2})$.

We have $\tau_1 = \tau_2 = 2\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$.

Protocol for the sending cluster $\mathcal{C}_1$:

. . . .

Protocol for the receiving cluster $\mathcal{C}_2$:

event $\mathrm{R}_2 \in \mathcal{G}_{\mathcal{C}_2}$ receives $(w, \mathrm{cert}(w, \mathrm{R}_1'))$ from $\mathrm{R}_1' \in \mathcal{C}_1$ **do**

  Broadcast $(w, \mathrm{cert}(w, \mathrm{R}_1'))$ to all replicas in $\mathcal{C}_2$.

event $\mathrm{R}_2' \in \mathcal{G}_{\mathcal{C}_2}$ receives $\mathbf{f}_{\mathcal{C}_1} + 1$ messages $(w, \mathrm{cert}(w, \mathrm{R}_1'))$:

(i) each message is sent by a replica in $\mathcal{C}_2$;

(ii) each message carries the same value w; and

(iii) each message has a distinct signature $\mathrm{cert}(w, \mathrm{R}_1')$, $\mathrm{R}_1' \in \mathcal{C}_1$

do

  R_2' considers w *received*.

Generalizing Bijective Sending

Consider bijective sending from $\mathcal{C}_1$ to $\mathcal{C}_2$, $\mathbf{n}_{\mathcal{C}_1} \geq \sigma_1 > \mathbf{n}_{\mathcal{C}_2}$, with certificates.

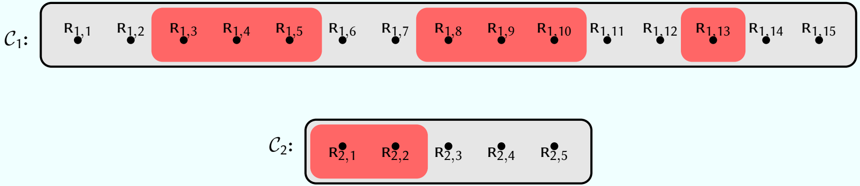
- Bijective sending requires $\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalizing Bijective Sending

Consider bijective sending from $\mathcal{C}_1$ to $\mathcal{C}_2$, $\mathbf{n}_{\mathcal{C}_1} \geq \sigma_1 > \mathbf{n}_{\mathcal{C}_2}$, with certificates.

- Bijective sending requires $\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalize bijective sending



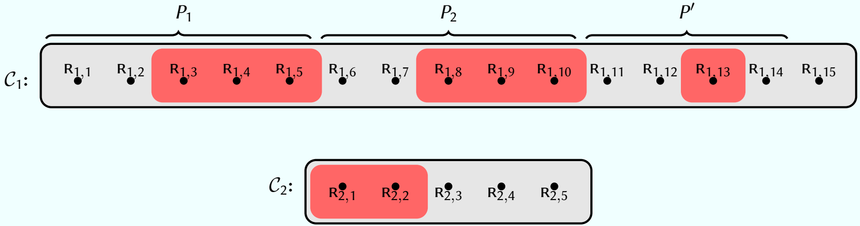
Generalizing Bijective Sending

Consider bijective sending from $\mathcal{C}_1$ to $\mathcal{C}_2$, $\mathbf{n}_{\mathcal{C}_1} \geq \sigma_1 > \mathbf{n}_{\mathcal{C}_2}$, with certificates.

- Bijective sending requires $\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalize bijective sending

- Partition σ_1 replicas of $\mathcal{C}_1$ into $\mathbf{n}_{\mathcal{C}_2}$-sized clusters.



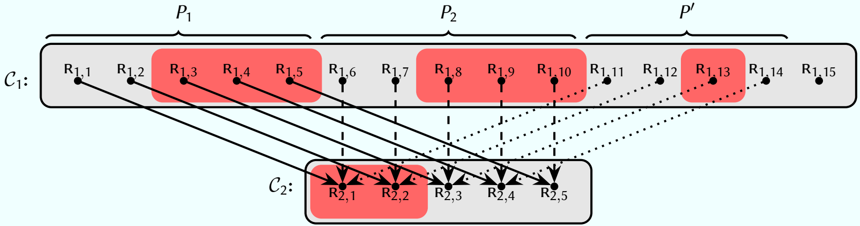
Generalizing Bijective Sending

Consider bijective sending from $\mathcal{C}_1$ to $\mathcal{C}_2$, $\mathbf{n}_{\mathcal{C}_1} \geq \sigma_1 > \mathbf{n}_{\mathcal{C}_2}$, with certificates.

- Bijective sending requires $\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalize bijective sending

- Partition σ_1 replicas of $\mathcal{C}_1$ into $\mathbf{n}_{\mathcal{C}_2}$-sized clusters.
- Bijective send from each cluster in the partition to $\mathcal{C}_2$.



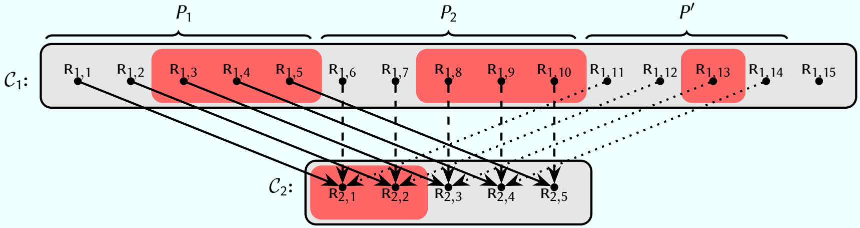
Generalizing Bijective Sending

Consider bijective sending from $\mathcal{C}_1$ to $\mathcal{C}_2$, $\mathbf{n}_{\mathcal{C}_1} \geq \sigma_1 > \mathbf{n}_{\mathcal{C}_2}$, with certificates.

- Bijective sending requires $\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$ distinct replicas in both clusters.
- Restrictive: clusters of roughly the same size.

Generalize bijective sending

- Partition σ_1 replicas of $\mathcal{C}_1$ into $\mathbf{n}_{\mathcal{C}_2}$-sized clusters.
- Bijective send from each cluster in the partition to $\mathcal{C}_2$.
- $\mathbf{n}_{\mathcal{C}_1} \geq \sigma_1$ holds always if $\mathbf{n}_{\mathcal{C}_1} > 3\mathbf{f}_{\mathcal{C}_1}$ and $\mathbf{n}_{\mathcal{C}_2} > 3\mathbf{f}_{\mathcal{C}_2}$.



Partitioned Bijective Sending

Corollary

Consider the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$.

1. *If $\mathbf{n}_{\mathcal{C}} > 3\mathbf{f}_{\mathcal{C}}$ for all clusters $\mathcal{C}$ and replicas only have crash failures or omit failures, then (partitioned) bijective sending solves cluster-sending with optimal message complexity.*
2. *If $\mathbf{n}_{\mathcal{C}} > 3\mathbf{f}_{\mathcal{C}}$ for all clusters $\mathcal{C}$ and clusters can produce certificates, then (partitioned) bijective sending solves cluster-sending with optimal message complexity.*
3. *If $\mathbf{n}_{\mathcal{C}} > 4\mathbf{f}_{\mathcal{C}}$ for all clusters $\mathcal{C}$ and replicas can digitally sign claims, then (partitioned) bijective sending solves cluster-sending with optimal message complexity.*

These protocols solve cluster-sending using $\mathcal{O}(\max(\mathbf{n}_{\mathcal{C}_1}, \mathbf{n}_{\mathcal{C}_2}))$ messages of size $\mathcal{O}(\|v\|)$ each.

Cluster-sending: Can we do Better?

Pessimistic

No: these algorithms are worst-case optimal.
Cannot do better than *linear communication* in the size of the clusters.

Cluster-sending: Can we do Better?

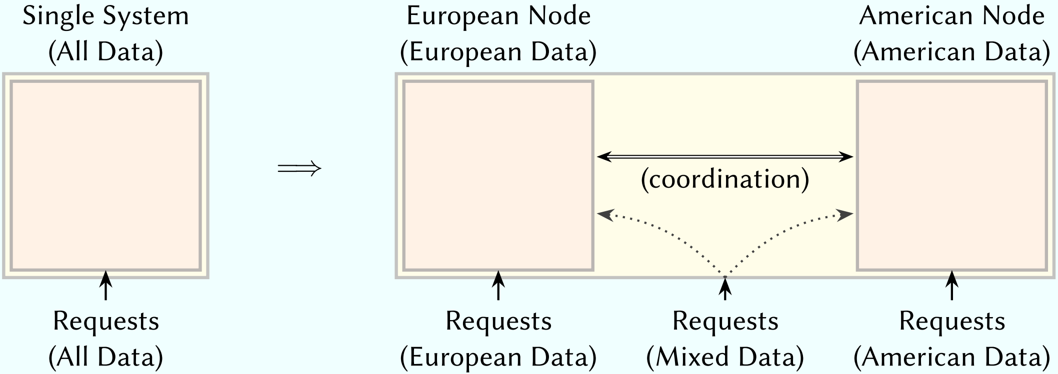
Pessimistic

No: these algorithms are worst-case optimal.
Cannot do better than *linear communication* in the size of the clusters.

Probabilistic

Yes: if we randomly choose sender and receiver, then we often do much better!
Probabilistic approach: expected-case only *constant communication* (four steps).

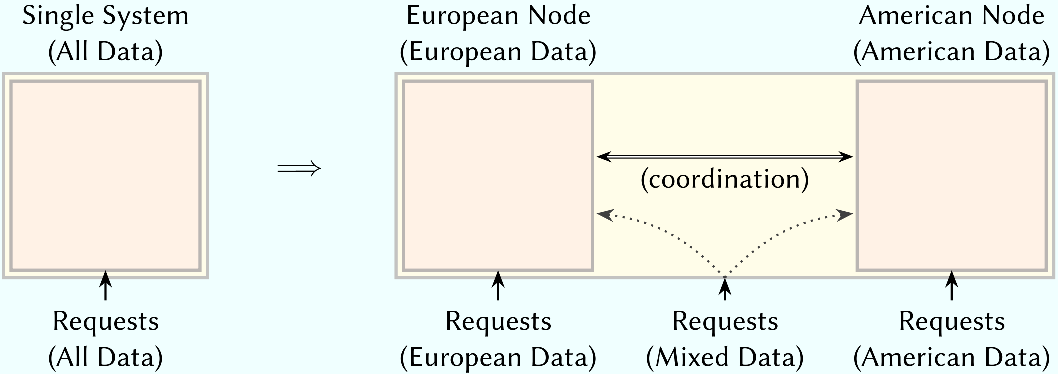
Motivation: High-Performance Resilient Systems



Partition the system: More storage and *potentially* more performance.
Potentially *lower latencies* if data ends up closer to users.

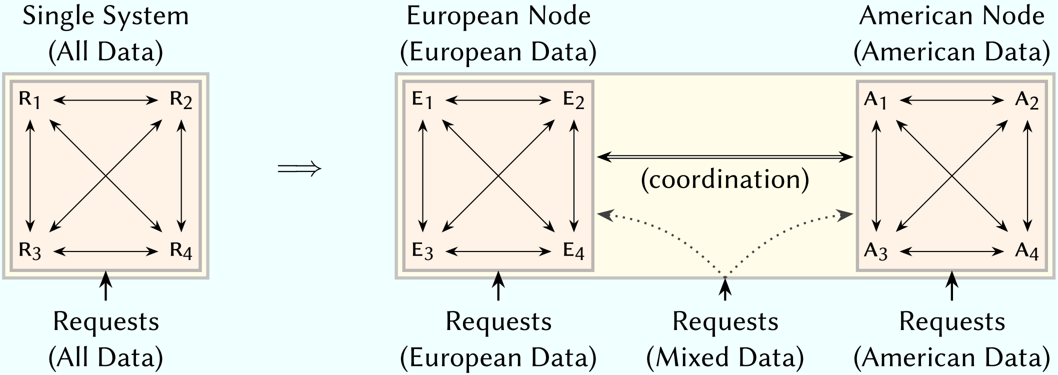
Adding shards $\Longrightarrow$ adding throughput (parallel processing), adding storage.

Motivation: High-Performance Resilient Systems



Resilient system

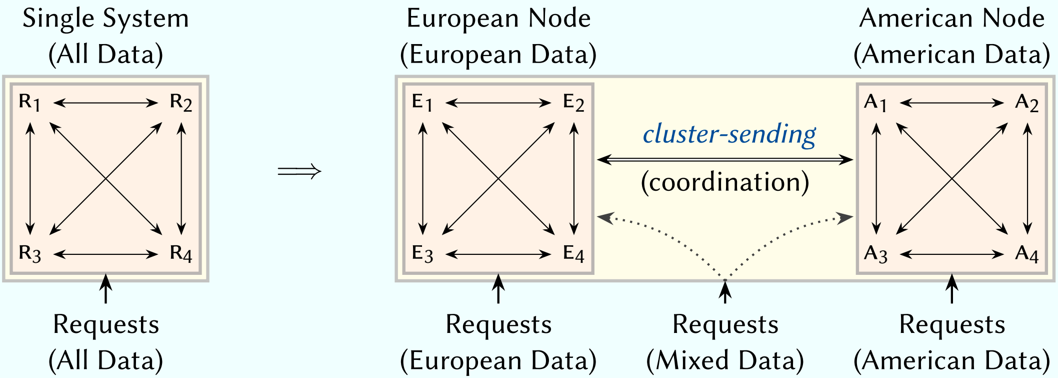
Motivation: High-Performance Resilient Systems



Resilient system

- Individual shards are consensus-operated *blockchains*.

Motivation: High-Performance Resilient Systems



Resilient system

- Individual shards are consensus-operated *blockchains*.
- Communication between shards via *cluster-sending*.

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:
the basic unit of change as seen by the DBMS.

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:
the basic unit of change as seen by the DBMS.

A transaction can be

- a single query;

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:
the basic unit of change as seen by the DBMS.

A transaction can be

- a single query;
- a set of queries;

Transactions

A user interaction with a DBMS: *transaction*.

Definition

A *transaction* is any one execution of a user program in a DBMS:
the basic unit of change as seen by the DBMS.

A transaction can be

- a single query;
- a set of queries;
- a interactive dialog between DBMS and program;
- ....

The ACID Properties

Contract between a DBMS and its users.

The ACID Properties

Contract between a DBMS and its users.

Given a *transaction* τ, a DBMS maintains

Atomicity. Either all or none of the operations of τ are reflected in the database.

Consistency Execution of τ in *isolation* preserves data consistency.
E.g., integrety constraints—this is *stronger* than CAP-Consistency.

Isolation τ is "unaware" of other transactions executing concurrently
"As-if" all transactions are executed in a *sequential order*.

Durability After τ completes successfully, the changes τ made persist.
If τ fails, then *no* changes persist due to atomicity.

The ACID Properties

Contract between a DBMS and its users.

Given a *transaction* τ, a DBMS maintains

Atomicity. Either all or none of the operations of τ are reflected in the database.

Consistency Execution of τ in *isolation* preserves data consistency.
E.g., integrety constraints—this is *stronger* than CAP-Consistency.

Isolation τ is "unaware" of other transactions executing concurrently
"As-if" all transactions are executed in a *sequential order*.

Durability After τ completes successfully, the changes τ made persist.
If τ fails, then *no* changes persist due to atomicity.

Assumption: individual transactions *make sense* (do not violate consistency).

The ACID Properties

Contract between a DBMS and its users.

Given a *transaction* τ, a DBMS maintains

Atomicity. Either all or none of the operations of τ are reflected in the database.

Consistency Execution of τ in *isolation* preserves data consistency.
E.g., integrety constraints—this is *stronger* than CAP-Consistency.

Isolation τ is "unaware" of other transactions executing concurrently
"As-if" all transactions are executed in a *sequential order*.

Durability After τ completes successfully, the changes τ made persist.
If τ fails, then *no* changes persist due to atomicity.

Assumption: individual transactions *make sense* (do not violate consistency).

Durability is *strong*: crashing or killing the DBMS program, power outage, ...

The ACID Properties

Contract between a DBMS and its users.

Given a *transaction* τ, a DBMS maintains

Atomicity. Either all or none of the operations of τ are reflected in the database.

Consistency Execution of τ in *isolation* preserves data consistency.
E.g., integrety constraints—this is *stronger* than CAP-Consistency.

Isolation τ is "unaware" of other transactions executing concurrently
"As-if" all transactions are executed in a *sequential order*.

Durability After τ completes successfully, the changes τ made persist.
If τ fails, then *no* changes persist due to atomicity.

Assumption: individual transactions *make sense* (do not violate consistency).

Durability is *strong*: crashing or killing the DBMS program, power outage, ...
Typical assumption: *storage* is permanent & reliable.

Background on Resilience

Consider a transaction τ requested by client c in a resilient system.

Background on Resilience

Consider a transaction τ requested by client c in a resilient system.

τ is processed in *five* steps

1. τ needs to be *received* by the system;
2. τ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order* for τ;
4. the replicas each need to *execute* τ and *update* their current state accordingly;
5. the client c needs to be *informed* about the result.

Background on Resilience

Consider a transaction τ requested by client c in a resilient system.

τ is processed in *five* steps

1. τ needs to be *received* by the system;
2. τ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order* for τ;
4. the replicas each need to *execute* τ and *update* their current state accordingly;
5. the client c needs to be *informed* about the result.

Non-sharded resilient systems

- Consensus solves all of the above.
- In particular *replication order* is *execution order*.
- Consecutive execution guarantees ACID.

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

$\tau_1 =$ "add \$500 to *Ana*";

$\tau_2 =$ "add \$200 to *Bo* and \$300 to *Elisa*";

$\tau_3 =$ "move \$30 from *Ana* to *Elisa*";

$\tau_4 =$ "remove \$70 from *Elisa*";

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

$\tau_1 =$ "add \$500 to *Ana*";

$\tau_2 =$ "add \$200 to *Bo* and \$300 to *Elisa*";

$\tau_3 =$ "move \$30 from *Ana* to *Elisa*";

$\tau_4 =$ "remove \$70 from *Elisa*";

Ana	\$0
Bo	\$0
Elisa	\$0

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

$\tau_1 =$ "add \$500 to *Ana*";

$\tau_2 =$ "add \$200 to *Bo* and \$300 to *Elisa*";

$\tau_3 =$ "move \$30 from *Ana* to *Elisa*";

$\tau_4 =$ "remove \$70 from *Elisa*";

Ana	\$0
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_1}$

Ana	\$500
Bo	\$0
Elisa	\$0

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

τ_1 = "add \$500 to *Ana*";

τ_2 = "add \$200 to *Bo* and \$300 to *Elisa*";

τ_3 = "move \$30 from *Ana* to *Elisa*";

τ_4 = "remove \$70 from *Elisa*";

Ana	\$0
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_1}$

Ana	\$500
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_2}$

Ana	\$500
Bo	\$200
Elisa	\$300

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

$$\tau_1 = \text{“add \$500 to } \textit{Ana}\text{”;}$$
$$\tau_2 = \text{“add \$200 to } \textit{Bo} \text{ and \$300 to } \textit{Elisa}\text{”;}$$
$$\tau_3 = \text{“move \$30 from } \textit{Ana} \text{ to } \textit{Elisa}\text{”;}$$
$$\tau_4 = \text{“remove \$70 from } \textit{Elisa}\text{”;}$$

Ana	\$0
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_1}$

Ana	\$500
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_2}$

Ana	\$500
Bo	\$200
Elisa	\$300

$\xrightarrow{\tau_3}$

Ana	\$470
Bo	\$200
Elisa	\$330

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

$$\tau_1 = \text{“add \$500 to } \textit{Ana}\text{”};$$
$$\tau_2 = \text{“add \$200 to } \textit{Bo} \text{ and \$300 to } \textit{Elisa}\text{”};$$
$$\tau_3 = \text{“move \$30 from } \textit{Ana} \text{ to } \textit{Elisa}\text{”};$$
$$\tau_4 = \text{“remove \$70 from } \textit{Elisa}\text{”};$$

Ana	\$0
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_1}$

Ana	\$500
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_2}$

Ana	\$500
Bo	\$200
Elisa	\$300

$\xrightarrow{\tau_3}$

Ana	\$470
Bo	\$200
Elisa	\$330

$\xrightarrow{\tau_4}$

Ana	\$470
Bo	\$200
Elisa	\$260

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

$$\tau_1 = \text{“add \$500 to } \textit{Ana}\text{”};$$
$$\tau_2 = \text{“add \$200 to } \textit{Bo} \text{ and \$300 to } \textit{Elisa}\text{”};$$
$$\tau_3 = \text{“move \$30 from } \textit{Ana} \text{ to } \textit{Elisa}\text{”};$$
$$\tau_4 = \text{“remove \$70 from } \textit{Elisa}\text{”};$$
$$\tau_5 = \text{“move \$500 from } \textit{Ana} \text{ to } \textit{Bo}\text{”}.$$

Ana	\$0
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_1}$

Ana	\$500
Bo	\$0
Elisa	\$0

$\xrightarrow{\tau_2}$

Ana	\$500
Bo	\$200
Elisa	\$300

$\xrightarrow{\tau_3}$

Ana	\$470
Bo	\$200
Elisa	\$330

$\xrightarrow{\tau_4}$

Ana	\$470
Bo	\$200
Elisa	\$260

Running Example: A Banking System

Setting: Transactions change the balance of one or more accounts

The *current state* is the balance of each account obtained by executing transactions.

τ_1 = “add \$500 to *Ana*”;

τ_2 = “add \$200 to *Bo* and \$300 to *Elisa*”;

τ_3 = “move \$30 from *Ana* to *Elisa*”;

τ_4 = “remove \$70 from *Elisa*”;

τ_5 = *aborted* (would invalidate balances).

Ana	$0
Bo	$0
Elisa	$0

$\xrightarrow{\tau_1}$

Ana	$500
Bo	$0
Elisa	$0

$\xrightarrow{\tau_2}$

Ana	$500
Bo	$200
Elisa	$300

$\xrightarrow{\tau_3}$

Ana	$470
Bo	$200
Elisa	$330

$\xrightarrow{\tau_4}$

Ana	$470
Bo	$200
Elisa	$260

Toward a Sharded and Resilient System

Consider a transaction τ requested by client c in a resilient system.

τ is processed in *five* steps

1. τ needs to be *received* by the system;
2. τ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order* for τ;
4. the replicas each need to *execute* τ and *update* their current state accordingly;
5. the client c needs to be *informed* about the result.

Toward a Sharded and Resilient System

Consider a transaction τ requested by client c in a resilient system.

τ is processed in *five* steps

1. τ needs to be *received* by the system;
2. τ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order* for τ;
4. the replicas each need to *execute* τ and *update* their current state accordingly;
5. the client c needs to be *informed* about the result.

τ must be *replicated* among all replicas of all shards affected by τ!

Toward a Sharded and Resilient System

Consider a transaction τ requested by client c in a resilient system.

τ is processed in *five* steps

1. τ needs to be *received* by the system;
2. τ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order* for τ;
4. the replicas each need to *execute* τ and *update* their current state accordingly;
5. the client c needs to be *informed* about the result.

What is a consistent execution order *across* shards? Does it relate to the *replication order*?

Toward a Sharded and Resilient System

Consider a transaction τ requested by client c in a resilient system.

τ is processed in *five* steps

1. τ needs to be *received* by the system;
2. τ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order* for τ;
4. the replicas each need to *execute* τ and *update* their current state accordingly;
5. the client c needs to be *informed* about the result.

Dependencies on data in *other shards*? Writes to data in *other shards*?

Toward a Sharded and Resilient System

Consider a transaction τ requested by client c in a resilient system.

τ is processed in *five* steps

1. τ needs to be *received* by the system;
2. τ must be *replicated* among all replicas in the system;
3. the replicas need to agree on an *execution order* for τ;
4. the replicas each need to *execute* τ and *update* their current state accordingly;
5. the client c needs to be *informed* about the result.

A single consensus does no longer solve all of the above!

Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system

Say we have 26 shards: $\mathcal{C}_a, \mathcal{C}_b, \ldots, \mathcal{C}_z$,
such that shard $\mathcal{C}_\xi$ holds accounts of people whose name starts with ξ.

Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system

Say we have 26 shards: $\mathcal{C}_a, \mathcal{C}_b, \ldots, \mathcal{C}_z$,
such that shard $\mathcal{C}_\xi$ holds accounts of people whose name starts with ξ.

We write shards(τ) to denote the shards affected by transaction τ.

Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system

Say we have 26 shards: $\mathcal{C}_a, \mathcal{C}_b, \ldots, \mathcal{C}_z$,
such that shard $\mathcal{C}_\xi$ holds accounts of people whose name starts with ξ.

We write $\text{shards}(\tau)$ to denote the shards affected by transaction τ.

$\tau_1 =$ "add \$500 to *Ana*",

$\tau_2 =$ "add \$200 to *Bo* and \$300 to *Elisa*",

$\tau_3 =$ "move \$30 from *Ana* to *Elisa*";

$\tau_4 =$ "remove \$70 from *Elisa*",

Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system

Say we have 26 shards: $\mathcal{C}_a, \mathcal{C}_b, \dots, \mathcal{C}_z$,
such that shard $\mathcal{C}_\xi$ holds accounts of people whose name starts with ξ.

We write $\text{shards}(\tau)$ to denote the shards affected by transaction τ.

$\tau_1 =$ "add \$500 to *Ana*", $\text{shards}(\tau_1) = \{\mathcal{C}_a\}$;

$\tau_2 =$ "add \$200 to *Bo* and \$300 to *Elisa*", $\text{shards}(\tau_2) = \{\mathcal{C}_b, \mathcal{C}_e\}$;

$\tau_3 =$ "move \$30 from *Ana* to *Elisa*"; $\text{shards}(\tau_3) = \{\mathcal{C}_a, \mathcal{C}_e\}$;

$\tau_4 =$ "remove \$70 from *Elisa*", $\text{shards}(\tau_4) = \{\mathcal{C}_e\}$.

Sharding Data

Sharded system: Data is distributed over all shards.

A sharded banking system

Say we have 26 shards: $\mathcal{C}_a, \mathcal{C}_b, \ldots, \mathcal{C}_z$,
such that shard $\mathcal{C}_\xi$ holds accounts of people whose name starts with ξ.

We write shards(τ) to denote the shards affected by transaction τ.

$\tau_1 =$ "add \$500 to *Ana*",	$\text{shards}(\tau_1) = \{\mathcal{C}_a\}$;	(single-shard)
$\tau_2 =$ "add \$200 to *Bo* and \$300 to *Elisa*",	$\text{shards}(\tau_2) = \{\mathcal{C}_b, \mathcal{C}_e\}$;	(multi-shard)
$\tau_3 =$ "move \$30 from *Ana* to *Elisa*";	$\text{shards}(\tau_3) = \{\mathcal{C}_a, \mathcal{C}_e\}$;	(multi-shard)
$\tau_4 =$ "remove \$70 from *Elisa*",	$\text{shards}(\tau_4) = \{\mathcal{C}_e\}$.	(single-shard)

An Example of Concurrent Execution

Consider a banking example in which

- Bo wants to transfer \$400 to Ana
  if Ana has at-least \$100 and Bo has at-least \$700,
- Ana wants to transfer \$300 to Elisa
  if Ana has at-least \$500,

and no account is allowed to have a negative balance.

An Example of Concurrent Execution

Consider a banking example in which

- Bo wants to transfer \$400 to Ana
  if Ana has at-least \$100 and Bo has at-least \$700,
- Ana wants to transfer \$300 to Elisa
  if Ana has at-least \$500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

An Example of Concurrent Execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

An Example of Concurrent Execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

$$\xrightarrow[\substack{A \geq 100? \\ A := A + 400}]{\tau_1 \text{ at } \mathcal{C}_a}$$

A	\$500
B	\$300
E	\$0

An Example of Concurrent Execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

$$\xrightarrow[\substack{A \geq 100? \\ A := A + 400}]{\tau_1 \text{ at } \mathcal{C}_a}$$

A	\$500
B	\$300
E	\$0

$$\xrightarrow[\substack{A \geq 500? \\ A := A - 300}]{\tau_2 \text{ at } \mathcal{C}_a}$$

A	\$200
B	\$300
E	\$0

An Example of Concurrent Execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

$\xrightarrow[A \geq 100?\; A := A + 400]{\tau_1 \text{ at } \mathcal{C}_a}$

A	\$500
B	\$300
E	\$0

$\xrightarrow[A \geq 500?\; A := A - 300]{\tau_2 \text{ at } \mathcal{C}_a}$

A	\$200
B	\$300
E	\$0

$\xrightarrow[B \geq 700?\; \text{(cancel)}]{\tau_1 \text{ at } \mathcal{C}_b}$

A	\$200
B	\$300
E	\$0

An Example of Concurrent Execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

$\xrightarrow[A \geq 100?,\ A := A + 400]{\tau_1 \text{ at } \mathcal{C}_a}$

A	\$500
B	\$300
E	\$0

$\xrightarrow[A \geq 500?,\ A := A - 300]{\tau_2 \text{ at } \mathcal{C}_a}$

A	\$200
B	\$300
E	\$0

$\xrightarrow[B \geq 700?\ \text{(cancel)}]{\tau_1 \text{ at } \mathcal{C}_b}$

A	\$200
B	\$300
E	\$0

$\xrightarrow[A := A - 400\ \text{(undo)}]{\tau_1 \text{ at } \mathcal{C}_a}$

A	-\$200
B	\$300
E	\$0

An Example of Concurrent Execution

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A	\$100
B	\$300
E	\$0

$\xrightarrow[A \geq 100?,\ A := A + 400]{\tau_1 \text{ at } \mathcal{C}_a}$

A	\$500
B	\$300
E	\$0

$\xrightarrow[A \geq 500?,\ A := A - 300]{\tau_2 \text{ at } \mathcal{C}_a}$

A	\$200
B	\$300
E	\$0

$\xrightarrow[B \geq 700?\ \text{(cancel)}]{\tau_1 \text{ at } \mathcal{C}_b}$

A	\$200
B	\$300
E	\$0

$\xrightarrow[A := A - 400\ \text{(undo)}]{\tau_1 \text{ at } \mathcal{C}_a}$

A	-\$200
B	\$300
E	\$0

$\xrightarrow[E := E + 300]{\tau_2 \text{ at } \mathcal{C}_e}$

A	-\$200
B	\$300
E	\$300

An Example of Concurrent Execution–Revisited

Consider a banking example in which

- Bo wants to transfer \$400 to Ana
  if Ana has at-least \$100 and Bo has at-least \$700,
- Ana wants to transfer \$300 to Elisa
  if Ana has at-least \$500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

An Example of Concurrent Execution–Revisited

Consider a banking example in which

- Bo wants to transfer \$400 to Ana
  if Ana has at-least \$100 and Bo has at-least \$700,
- Ana wants to transfer \$300 to Elisa
  if Ana has at-least \$500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Transactions τ_1 and τ_2 make sense:
their isolated execution will never make balances negative.

An Example of Concurrent Execution–Revisited

Consider a banking example in which

- Bo wants to transfer \$400 to Ana
  if Ana has at-least \$100 and Bo has at-least \$700,
- Ana wants to transfer \$300 to Elisa
  if Ana has at-least \$500,

and no account is allowed to have a negative balance.

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Transactions τ_1 and τ_2 make sense:
their isolated execution will never make balances negative.

Guarantee by an ACID-compliant system

No account will ever have a negative balance.

Serializability: a High Standard for Isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Serializability: a High Standard for Isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Serializability: a High Standard for Isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Definition

A *serializable schedule* is a schedule whose effect on any consistent instance is guaranteed to be identical to that of some serial schedule over the *committed transactions* in S.

Serializability: a High Standard for Isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Definition

A *serializable schedule* is a schedule whose effect on any consistent instance is guaranteed to be identical to that of some serial schedule over the *committed transactions* in S.

Serializability assumes *aborted* transactions have no side effects.

Serializability: a High Standard for Isolation

Consider a set of transactions $S = \{\tau_1, \ldots, \tau_n\}$.

Definition

A *serial schedule* is an execution of S without *interleaving* of transaction steps.
Hence, each transaction is executed in sequence, one at a time.

Definition

A *serializable schedule* is a schedule whose effect on any consistent instance is guaranteed to be identical to that of some serial schedule over the *committed transactions* in S.

Serializability assumes *aborted* transactions have no side effects.
This is not always the case (example later).

Simplified Transaction Notation

Consider the transaction τ:

$\tau =$ "if *Ana* has \$500 and *Bo* has \$200, then
move \$400 from *Ana* to *Elisa*;
move \$100 from *Bo* to *Elisa*".

Simplified Transaction Notation

Consider the transaction τ:

$\tau =$ "if *Ana* has \$500 and *Bo* has \$200, then
move \$400 from *Ana* to *Elisa*;
move \$100 from *Bo* to *Elisa*".

What are the operations of τ?

Depending on *how* the system executes τ and the database state:

- Might read from *Ana*'s account.
- Might read from *Bo*'s account.
- Might write to *Ana*'s account.
- Might write to *Bo*'s account.
- Might write to *Elisa*'s account.

Simplified Transaction Notation

Consider the transaction τ:

$\tau =$ "if *Ana* has \$500 and *Bo* has \$200, then
move \$400 from *Ana* to *Elisa*;
move \$100 from *Bo* to *Elisa*".

Simplifying assumption

Each transaction is a sequence of read and write operations ending in *commit* or *abort*.

Simplified Transaction Notation

Consider the transaction τ:

$\tau =$ "if *Ana* has \$500 and *Bo* has \$200, then
move \$400 from *Ana* to *Elisa*;
move \$100 from *Bo* to *Elisa*".

Simplifying assumption

Each transaction is a sequence of read and write operations ending in *commit* or *abort*.

$\mathsf{Read}_\tau(\mathit{Ana}), \mathsf{Read}_\tau(\mathit{Bo}), \mathsf{Write}_\tau(\mathit{Ana}), \mathsf{Write}_\tau(\mathit{Bo}), \mathsf{Read}_\tau(\mathit{Elisa}), \mathsf{Write}_\tau(\mathit{Elisa}), \mathsf{Commit}_\tau.$

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (insufficient funds)

Instance (initial)

A	\$100
B	\$300
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (insufficient funds)

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	
	$\text{Read}_{\tau_2}(A)$
	Abort_{τ_2}

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (insufficient funds)

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	
	$\text{Read}_{\tau_2}(A)$
	Abort_{τ_2}

Instance (final)

A	\$100
B	\$300
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_1, then τ_2 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}

Instance (final)

A	\$200
B	\$400
E	\$300

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	Abort_{τ_2}
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Bob has sufficient funds)

Instance (initial)

A	\$100
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	Abort_{τ_2}
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

Instance (final)

A	\$500
B	\$400
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Ana has sufficient funds)

Instance (initial)

A	\$500
B	\$300
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Ana has sufficient funds)

Instance (initial)

A	\$500
B	\$300
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Serial schedule: τ_2, then τ_1 (Ana has sufficient funds)

Instance (initial)

A	\$500
B	\$300
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

Instance (final)

A	\$200
B	\$300
E	\$300

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Earlier example

Instance (initial)

A	\$100
B	\$300
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Earlier example

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(B)$	
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Earlier example

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Read}_{\tau_1}(B)$	
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
Abort_{τ_1}	

Instance (final)

A	-\$200
B	\$300
E	\$300

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Another example

Instance (initial)

A	\$500
B	\$800
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Another example

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—Another example

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

Instance (final)

A	\$900
B	\$400
E	\$300

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—A third example

Instance (initial)

A	\$500
B	\$800
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—A third example

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Non-serial schedule—A third example

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}

Instance (final)

A	\$200
B	\$400
E	\$300

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A serializable schedule (that is non-serial)

Instance (initial)

A	\$500
B	\$800
E	\$0

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A serializable schedule (that is non-serial)

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
	Commit_{τ_2}
Commit_{τ_1}	

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$

$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

A serializable schedule (that is non-serial)

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
	Commit_{τ_2}
Commit_{τ_1}	

Instance (final)

A	\$600
B	\$400
E	\$300

An Example of Schedules

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Key observation: Serial schedules

Individual transactions *make sense* (do not violate consistency):

- No balance will ever get negative.
- No money disappears or appears out of thin air.

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- All transactions operate on *shared data* (the database instance).

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

As all data is shared: should the entire transaction be a single critical section?

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ….

As all data is shared: should the entire transaction be a single critical section?

What if each transaction *locks the system*, executes, *releases the system*?

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

As all data is shared: should the entire transaction be a single critical section?

What if each transaction *locks the system*, executes, *releases the system*?

This will enforce a *serial schedule*.

Guaranteeing Isolation

Simplified point-of-view

- A transaction is a *thread* in a multi-threaded program.
- All transactions operate on *shared data* (the database instance).
- We need to coordinate access to this shared data!
  In traditional multi-threaded programs:
  - Use *critical sections* in which shared data is accessed.
  - Enforce *critical sections* with locks (e.g., mutex).
  - Ensure proper lock usage to avoid deadlocks, ...

As all data is shared: should the entire transaction be a single critical section?

What if each transaction *locks the system*, executes, *releases the system*?

This will enforce a *serial schedule* and eliminate any concurrency.

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.

E.g., accounts, tables, rows, ...

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Using fine-grained locks

A transaction τ that wants to access database object O will:

- waits until it obtains a lock on O ($\text{Lock}_\tau(O)$),
- then perform its operations on O (e.g., $\text{Read}_\tau(O)$ and $\text{Write}_\tau(O)$), and
- finally release the lock on O ($\text{Release}_\tau(O)$).

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Read}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(E)$
	$\text{Write}_{\tau_2}(E)$
	Commit_{τ_2}
$\text{Write}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(B)$	
$\text{Write}_{\tau_1}(B)$	
Commit_{τ_1}	

Instance (final)

A	\$900
B	\$400
E	\$300

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues . . .

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	...
	Commit_{τ_2}

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	...
	Commit_{τ_2}
...	
Commit_{τ_1}	

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

Lock-based access solves *some* issues ...

Instance (initial)

A	\$500
B	\$800
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Read}_{\tau_2}(A)$
	...
	Commit_{τ_2}
...	
Commit_{τ_1}	

Instance (final)

A	\$600
B	\$400
E	\$300

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ….

In our examples we abstract from details: *accounts* are database objects.

…but not *all* issues …

Instance (initial)

A	\$100
B	\$300
E	\$0

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

...but not *all* issues ...

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	...
	Commit_{τ_2}
...	
Abort_{τ_1}	

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

...but not *all* issues ...

Instance (initial)

A	\$100
B	\$300
E	\$0

Schedule

$\text{Lock}_{\tau_1}(A)$	
$\text{Read}_{\tau_1}(A)$	
$\text{Write}_{\tau_1}(A)$	
$\text{Release}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(A)$
	$\text{Read}_{\tau_2}(A)$
	$\text{Write}_{\tau_2}(A)$
	...
	Commit_{τ_2}
...	
Abort_{τ_1}	

Instance (final)

A	-\$200
B	\$300
E	\$300

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

...and introduces *new* issues.
Consider two transactions that both want to access *Ana* and *Bo*:

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \ldots; \qquad \tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_1}(A), \ldots$$

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

...and introduces *new* issues.

Consider two transactions that both want to access *Ana* and *Bo*:

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \ldots; \qquad \tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_1}(A), \ldots$$

Schedule

$\mathsf{Lock}_{\tau_1}(A)$	
	$\mathsf{Lock}_{\tau_2}(B)$
$\mathsf{Lock}_{\tau_1}(B)$	
	$\mathsf{Lock}_{\tau_2}(A)$

Improving Isolation using Locks

Idea: Use a fine-grained set of locks on *database objects*.
E.g., accounts, tables, rows, ...

In our examples we abstract from details: *accounts* are database objects.

...and introduces *new* issues.
Consider two transactions that both want to access *Ana* and *Bo*:

$$\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \ldots; \qquad \tau_2 = \text{Lock}_{\tau_2}(B), \text{Lock}_{\tau_1}(A), \ldots$$

Schedule

$\text{Lock}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(B)$
$\text{Lock}_{\tau_1}(B)$	
	$\text{Lock}_{\tau_2}(A)$

Both transactions will wait forever: a deadlock!

Achieving Serializability with Locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) that do guarantee *serializability*.

Achieving Serializability with Locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) that do guarantee *serializability*.

Two-phase locking protocol (2PL)

Execution of transaction τ adheres to 2PL if the execution is performed in two phases:

Growing phase during which execution can obtains locks, and *not* release them; and

Shrinking phase during which execution can release locks, and *not* obtain them,

and any database object O is only operated on while holding lock $\text{Lock}_{\tau}(O)$.

Achieving Serializability with Locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) that do guarantee *serializability*.

Two-phase locking protocol (2PL)

Execution of transaction τ adheres to 2PL if the execution is performed in two phases:

Growing phase during which execution can obtains locks, and *not* release them; and

Shrinking phase during which execution can release locks, and *not* obtain them,

and any database object O is only operated on while holding lock $\text{Lock}_\tau(O)$.

Strict 2PL: locks are only released after completion (Commit_τ or Abort_τ).

Achieving Serializability with Locks

Locking itself does not guarantee *serializability*.

Some *locking protocols* (sets of rules on when to use locks) that do guarantee *serializability*.

Two-phase locking protocol (2PL)

Execution of transaction τ adheres to 2PL if the execution is performed in two phases:

Growing phase during which execution can obtains locks, and *not* release them; and

Shrinking phase during which execution can release locks, and *not* obtain them,

and any database object O is only operated on while holding lock $\text{Lock}_\tau(O)$.

Strict 2PL: locks are only released after completion (Commit_τ or Abort_τ).

Notice—Nothing to deal with *deadlocks*.

An Example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

An Example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B),$$
$$\mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(A), \mathsf{Release}_{\tau_1}(B);$$
$$\tau_2 = \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E),$$
$$\mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).$$

An Example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B), \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(A), \mathsf{Release}_{\tau_1}(B);$$
$$\tau_2 = \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E), \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).$$

An Example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B),$$
$$\mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(B), \mathsf{Release}_{\tau_1}(A);$$
$$\tau_2 = \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E),$$
$$\mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).$$

An Example of 2PL

Consider again the transactions

$$\tau_1 = A \geq 100?, A := A + 400, B \geq 700?, B := B - 400;$$
$$\tau_2 = A \geq 500?, A := A - 300, E := E + 300.$$

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(A), \mathsf{Read}_{\tau_1}(B), \mathsf{Write}_{\tau_1}(B),$$
$$\mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(B), \mathsf{Release}_{\tau_1}(A);$$
$$\tau_2 = \mathsf{Lock}_{\tau_2}(E), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(A), \mathsf{Write}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(E), \mathsf{Write}_{\tau_2}(E),$$
$$\mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(E).$$

These are all *strict* 2PL: locks are released after the transactions commit.

An Example of 2PL

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(A), \text{Read}_{\tau_1}(B), \text{Write}_{\tau_1}(B), \text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(B), \text{Release}_{\tau_1}(A);$$

$$\tau_2 = \text{Lock}_{\tau_2}(E), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(A), \text{Write}_{\tau_2}(A), \text{Read}_{\tau_2}(E), \text{Write}_{\tau_2}(E), \text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(E).$$

Consider any schedule with any interleaving of operations of τ_1 and τ_2

An Example of 2PL

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\begin{aligned}\tau_1 = {} & \mathrm{Lock}_{\tau_1}(A), \mathrm{Lock}_{\tau_1}(B), \mathrm{Read}_{\tau_1}(A), \mathrm{Write}_{\tau_1}(A), \mathrm{Read}_{\tau_1}(B), \mathrm{Write}_{\tau_1}(B),\\ & \mathrm{Commit}_{\tau_1}, \mathrm{Release}_{\tau_1}(B), \mathrm{Release}_{\tau_1}(A);\\ \tau_2 = {} & \mathrm{Lock}_{\tau_2}(E), \mathrm{Lock}_{\tau_2}(A), \mathrm{Read}_{\tau_2}(A), \mathrm{Write}_{\tau_2}(A), \mathrm{Read}_{\tau_2}(E), \mathrm{Write}_{\tau_2}(E),\\ & \mathrm{Commit}_{\tau_2}, \mathrm{Release}_{\tau_2}(A), \mathrm{Release}_{\tau_2}(E).\end{aligned}$$

Consider any schedule with any interleaving of operations of τ_1 and τ_2

- If τ_1 executes $\mathrm{Lock}_{\tau_1}(A)$ *before* τ_2 executes $\mathrm{Lock}_{\tau_2}(A)$:
  all read and write operations of τ_1 effectively happen before those of τ_2.

An Example of 2PL

Assumption: Both transactions will succeed (Alice and Bob have sufficient funds)

$$\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(A), \text{Read}_{\tau_1}(B), \text{Write}_{\tau_1}(B), \text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(B), \text{Release}_{\tau_1}(A);$$

$$\tau_2 = \text{Lock}_{\tau_2}(E), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(A), \text{Write}_{\tau_2}(A), \text{Read}_{\tau_2}(E), \text{Write}_{\tau_2}(E), \text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(E).$$

Consider any schedule with any interleaving of operations of τ_1 and τ_2

- If τ_1 executes $\text{Lock}_{\tau_1}(A)$ *before* τ_2 executes $\text{Lock}_{\tau_2}(A)$:
  all read and write operations of τ_1 effectively happen before those of τ_2.
- If τ_2 executes $\text{Lock}_{\tau_2}(A)$ *before* τ_1 executes $\text{Lock}_{\tau_1}(A)$:
  all read and write operations of τ_2 effectively happen before those of τ_1.

Two-Phase Locking and Deadlocks

Consider the transactions

$$\tau_1 = \mathsf{Lock}_{\tau_1}(A), \mathsf{Lock}_{\tau_1}(B), \mathsf{Read}_{\tau_1}(A), \mathsf{Write}_{\tau_1}(B), \mathsf{Commit}_{\tau_1}, \mathsf{Release}_{\tau_1}(A), \mathsf{Release}_{\tau_1}(B);$$

$$\tau_2 = \mathsf{Lock}_{\tau_2}(B), \mathsf{Lock}_{\tau_2}(A), \mathsf{Read}_{\tau_2}(B), \mathsf{Write}_{\tau_2}(A), \mathsf{Commit}_{\tau_2}, \mathsf{Release}_{\tau_2}(A), \mathsf{Release}_{\tau_2}(B).$$

These transactions are strict 2PL.

Two-Phase Locking and Deadlocks

Consider the transactions

$\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(B), \text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(A), \text{Release}_{\tau_1}(B);$

$\tau_2 = \text{Lock}_{\tau_2}(B), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(B), \text{Write}_{\tau_2}(A), \text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(B).$

These transactions are strict 2PL.

Some schedules will cause a deadlock

Schedule

$\text{Lock}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(B)$
$\text{Lock}_{\tau_1}(B)$	
	$\text{Lock}_{\tau_2}(A)$

Two-Phase Locking and Deadlocks

Consider the transactions

$$\tau_1 = \text{Lock}_{\tau_1}(A), \text{Lock}_{\tau_1}(B), \text{Read}_{\tau_1}(A), \text{Write}_{\tau_1}(B), \text{Commit}_{\tau_1}, \text{Release}_{\tau_1}(A), \text{Release}_{\tau_1}(B);$$

$$\tau_2 = \text{Lock}_{\tau_2}(B), \text{Lock}_{\tau_2}(A), \text{Read}_{\tau_2}(B), \text{Write}_{\tau_2}(A), \text{Commit}_{\tau_2}, \text{Release}_{\tau_2}(A), \text{Release}_{\tau_2}(B).$$

These transactions are strict 2PL.

Some schedules will cause a deadlock

Schedule

$\text{Lock}_{\tau_1}(A)$	
	$\text{Lock}_{\tau_2}(B)$
$\text{Lock}_{\tau_1}(B)$	
	$\text{Lock}_{\tau_2}(A)$

Deadlocks are one of the issues arising from *lock contention*.

Dealing with Deadlocks: Pessimistic Approach

Pessimistic: make sure deadlocks *cannot happen*

Dealing with Deadlocks: Pessimistic Approach

Pessimistic: make sure deadlocks *cannot happen*

Enforce that all transactions obtain their locks in a unique predetermined order.
E.g., first locks on Ana, then Bo, then Celeste, then Dafni, then Elisa, ...

Dealing with Deadlocks: Pessimistic Approach

Pessimistic: make sure deadlocks *cannot happen*

Enforce that all transactions obtain their locks in a unique predetermined order.
E.g., first locks on Ana, then Bo, then Celeste, then Dafni, then Elisa, ...

Example

Consider the transaction

$$\tau = \text{“if } \textit{Bo} \text{ has \$500, then move \$200 from } \textit{Bo} \text{ to } \textit{Ana}\text{”.}$$

Any schedule for τ needs to start with:

$$\mathsf{Lock}_\tau(\mathit{Ana}), \mathsf{Lock}_\tau(\mathit{Bo}), \dots,$$

we even lock Ana if Bo does *not have funds*.

Dealing with Deadlocks: Optimistic Approach

Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

Dealing with Deadlocks: Optimistic Approach

Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for *deadlock detection* or *prevention*.

Dealing with Deadlocks: Optimistic Approach

Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for *deadlock detection* or *prevention*.
- Very easy to implement.

Dealing with Deadlocks: Optimistic Approach

Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for *deadlock detection* or *prevention*.
- Very easy to implement.
- Minimizes the costs for transactions that are able to commit.

Dealing with Deadlocks: Optimistic Approach

Optimistic: Optimize for no lock-contention

If a transaction tries to obtain a lock that is already held: *abort the transaction entirely*.

- No need for *deadlock detection* or *prevention*.
- Very easy to implement.
- Minimizes the costs for transactions that are able to commit.
- Will perform badly when there is a high amount of lock-contention.

Practice: Read and Write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Practice: Read and Write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Goal: prevent writes concurrent with other activity, but minimize cost for reads.

Practice: Read and Write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Goal: prevent writes concurrent with other activity, but minimize cost for reads.

Introduce separate read and write locks

- Multiple transactions can hold a lock at the same time *if they all hold read locks*.
- Only one transaction can hold a lock *if that transaction holds a write lock*.

Practice: Read and Write locks

- Locks need to be *fine-grained* to maximize concurrency.
- Concurrency issues only arise when a transaction is writing.
- In most workloads: reads are much more frequent than writes.

Goal: prevent writes concurrent with other activity, but minimize cost for reads.

Introduce separate read and write locks

- Multiple transactions can hold a lock at the same time *if they all hold read locks*.
- Only one transaction can hold a lock *if that transaction holds a write lock*.

Result

- Many transactions can read at the same time.
- Read-write, write-read, and write-write conflicts are prevented.

The Cost of Serializability

- Serializability provides *strong* isolation guarantees.
- Providing these guarantees *will* impact concurrency
  (independent of the implementation mechanism, e.g., locks).

The Cost of Serializability

- Serializability provides *strong* isolation guarantees.
- Providing these guarantees *will* impact concurrency (independent of the implementation mechanism, e.g., locks).

To improve performance, you can *give up* on serializability!

Degrees of Isolation in SQL[1]

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

[1] There are excellent papers on this topic! E.g., `https://doi.org/10.1145/568271.223785` and `https://doi.org/10.1016/0950-5849(96)01109-3` are recommended.

Degrees of Isolation in SQL[1]

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

[1] There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of Isolation in SQL[1]

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **READ UNCOMMITTED**

- no read locks,
- *long-duration* write (and predicate) locks before writing data.

[1] There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of Isolation in SQL[1]

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **READ COMMITTED**

- *short-duration* read (and predicate) locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

[1]There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of Isolation in SQL[1]

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **REPEATABLE READ**

- *short-duration* predicate locks and *long-duration* read locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

[1]There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Degrees of Isolation in SQL[1]

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **SERIALIZABLE**

- *long-duration* read (and predicate) locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

[1]There are excellent papers on this topic! E.g., `https://doi.org/10.1145/568271.223785` and `https://doi.org/10.1016/0950-5849(96)01109-3` are recommended.

Degrees of Isolation in SQL[1]

Level	Dirty Reads	Unrepeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Each *level* can be defined in terms of a locking protocol.

Locking protocol for **SERIALIZABLE** (2PL)

- *long-duration* read (and predicate) locks before reading data, and
- *long-duration* write (and predicate) locks before writing data.

[1] There are excellent papers on this topic! E.g., https://doi.org/10.1145/568271.223785 and https://doi.org/10.1016/0950-5849(96)01109-3 are recommended.

Safe Execution without Locks

Concurrent transaction execution can make sense *without* isolation.
E.g., one can use application-specific knowledge!

Safe Execution without Locks

Concurrent transaction execution can make sense *without* isolation.
E.g., one can use application-specific knowledge!

A Banking System

Observe: undoing a withdraw increases balance, undoing deposits decreases balance!

Safe Execution without Locks

Concurrent transaction execution can make sense *without* isolation.
E.g., one can use application-specific knowledge!

A Banking System

Observe: undoing a withdraw increases balance, undoing deposits decreases balance!

Consider executions in which all steps can:

- always withdraw money;
- only deposit money after either *commit* or *abort* is decided.

Safe Execution without Locks

Concurrent transaction execution can make sense *without* isolation.
E.g., one can use application-specific knowledge!

A Banking System

Observe: undoing a withdraw increases balance, undoing deposits decreases balance!

Consider executions in which all steps can:

- always withdraw money;
- only deposit money after either *commit* or *abort* is decided.

These executions guarantee that no account will have a negative balance!

Ingredients of Sharding in a Resilient Environment

Multi-shard transaction execution of τ requires

Replication of τ among shards.
E.g., a two-phase commit step.

Concurrency control to guarantee consistent execution of τ.
E.g., using *locks* to prevent concurrent access to accounts.

To One needs *computations* within a shard and *communication* between shards.

Ingredients of Sharding in a Resilient Environment

Multi-shard transaction execution of τ requires

Replication of τ among shards.
E.g., a two-phase commit step.

Concurrency control to guarantee consistent execution of τ.
E.g., using *locks* to prevent concurrent access to accounts.

To One needs *computations* within a shard and *communication* between shards.

Fault-tolerant shards

Each shard is a cluster of replicas that can be faulty.

Consensus for each *computation* within shards.

Cluster-sending for any *communication* between shards.

Consensus is costly: Minimize its use.

The Orchestrate-Execute Model for Multi-Shard Transactions

Consider a multi-shard transaction τ:

- Processing is broken down into three types of *shard-steps*: vote, commit, and abort.
- Each shard-step is performed via *one* consensus step.
- Transfer control between steps using *cluster-sending*.

Execution method determines the local operations of a shard-step:
locks, *checking conditions*, *updating state*, *...*.

Orchestration method determines how *control is transferred* between shard-steps:
perform *votes*, collect *votes*, decide *commit* or *abort* τ.

Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

$\tau =$ "if *Ana* has \$500 and *Bo* has \$200, then move \$400 from *Ana* to *Bo*."

Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

$$\tau = \text{“if } \textit{Ana} \text{ has \$500 and } \textit{Bo} \text{ has \$200, then move \$400 from } \textit{Ana} \text{ to } \textit{Bo}\text{.”}$$

$$\sigma_1 = \text{“}\mathsf{Lock}_\tau(\textit{Ana})\text{; if } \textit{Ana} \text{ has \$500, then forward } \sigma_2 \text{ to } \mathcal{C}_b \text{ (commit vote) else } \mathsf{Release}_\tau(\textit{Ana}) \text{ (abort vote).”}$$

vote-step

σ_1 at $\mathcal{C}_a$

Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

$$\tau = \text{“if } \textit{Ana} \text{ has \$500 and } \textit{Bo} \text{ has \$200, then move \$400 from } \textit{Ana} \text{ to } \textit{Bo}\text{.”}$$

$$\sigma_2 = \text{“Lock}_\tau(\textit{Bo})\text{; if } \textit{Bo} \text{ has \$200, then add \$400 to } \textit{Bo}\text{; Release}_\tau(\textit{Bo})\text{; and forward } \sigma_3 \text{ to } \mathcal{C}_a \text{ (commit) else Release}_\tau(\textit{Bo}) \text{ and forward } \sigma_4 \text{ to } \mathcal{C}_a \text{ (abort).”}$$

vote-step σ_1 at $\mathcal{C}_a$ —*vote commit*→ **vote-step** σ_2 at $\mathcal{C}_b$

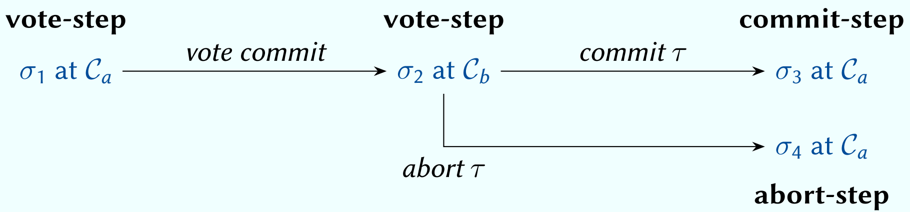
Example of the Orchestrate-Execute Model

Shard accounts by first letter of name

$$\tau = \text{“if } \textit{Ana} \text{ has \$500 and } \textit{Bo} \text{ has \$200, then move \$400 from } \textit{Ana} \text{ to } \textit{Bo}\text{.”}$$

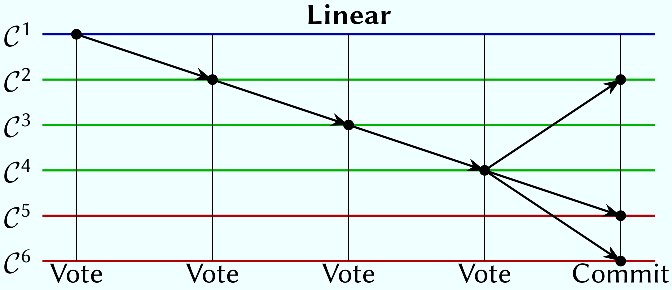
$$\sigma_3 = \text{“remove \$400 from } \textit{Ana} \text{ and } \mathrm{Release}_\tau(\textit{Ana})\text{.”}$$

$$\sigma_4 = \text{“}\mathrm{Release}_\tau(\textit{Ana})\text{.”}$$



Resilient Orchestration Methods

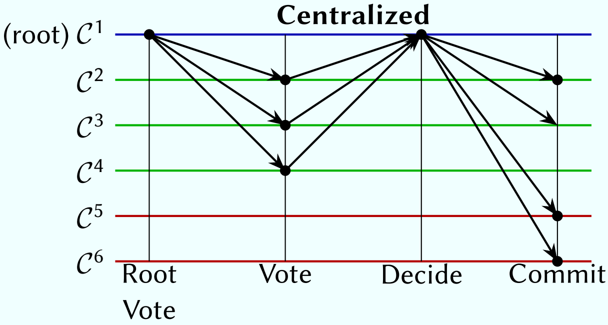
Orchestration ≈ two-phase commit, except that *shards never fail*.



Vote-steps in *sequence*, decide *centralized*, commit or abort in *parallel*.

Resilient Orchestration Methods

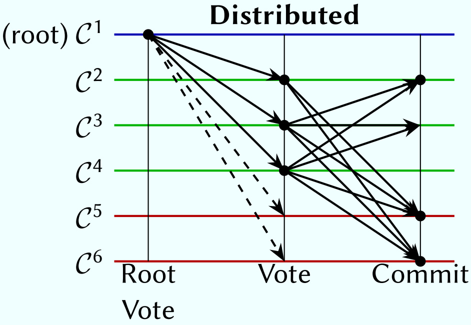
Orchestration ≈ two-phase commit, except that *shards never fail*.



Vote-steps in *parallel*, decide *centralized*, commit or abort in *parallel*.

Resilient Orchestration Methods

Orchestration ≈ two-phase commit, except that *shards never fail*.



Vote-steps in *parallel*, decide *decentralized*, commit or abort in *parallel*.

Resilient Execution Methods

Execution updates state and performs *concurrency control*.

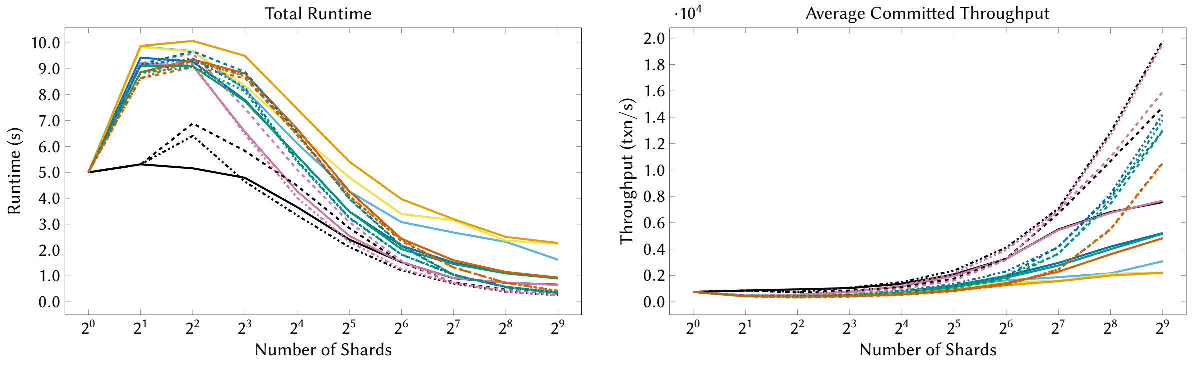
- Write uncommitted execution for *free*:
  Due to consensus, shard-steps are performed in sequence on that shard.

- Higher isolation levels via *two-phase locking*:
  - read uncommitted execution: only *write locks*;
  - read committed execution: *read locks* during steps;
  - serializable execution: *read and write locks*.

- Blocking locks (with linear orchestration) versus non-blocking locks.

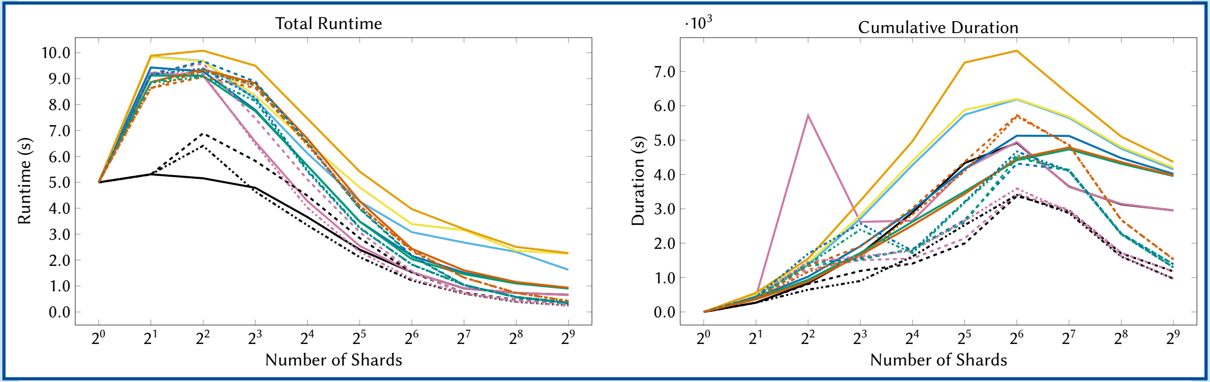
Evaluation

	Isolation-Free execution (write uncommitted)		Lock-based execution: Read Uncommitted		Read Committed		Serializable	
	unsafe	*safe*	*blocking*	*non-blocking*	*blocking*	*non-blocking*	*blocking*	*non-blocking*
Linear	LIFu	LIFs	LRUb	LRUnb	LRCb	LRCnb	LSb	LSnb
Centralized	CIFu	CIFs		CRUnb		CRCnb		CSnb
Distributed	DIFu	DIFs		DRUnb		DRCnb		DSnb



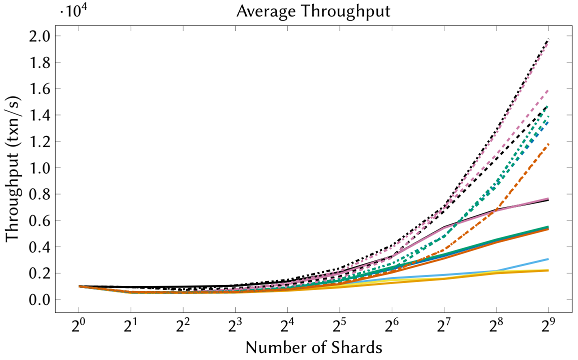
Evaluation

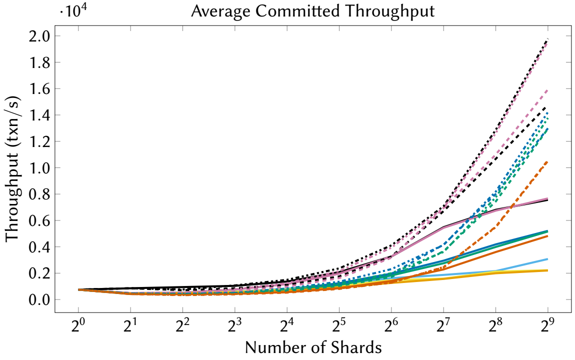
	Isolation-Free execution (write uncommitted)		Lock-based execution					
			Read Uncommitted		Read Committed		Serializable	
	unsafe	*safe*	*blocking*	*non-blocking*	*blocking*	*non-blocking*	*blocking*	*non-blocking*
Linear	LIFu	LIFs	LRUb	LRUnb	LRCb	LRCnb	LSb	LSnb
Centralized	CIFu	CIFs		CRUnb		CRCnb		CSnb
Distributed	DIFu	DIFs		DRUnb		DRCnb		DSnb



Evaluation

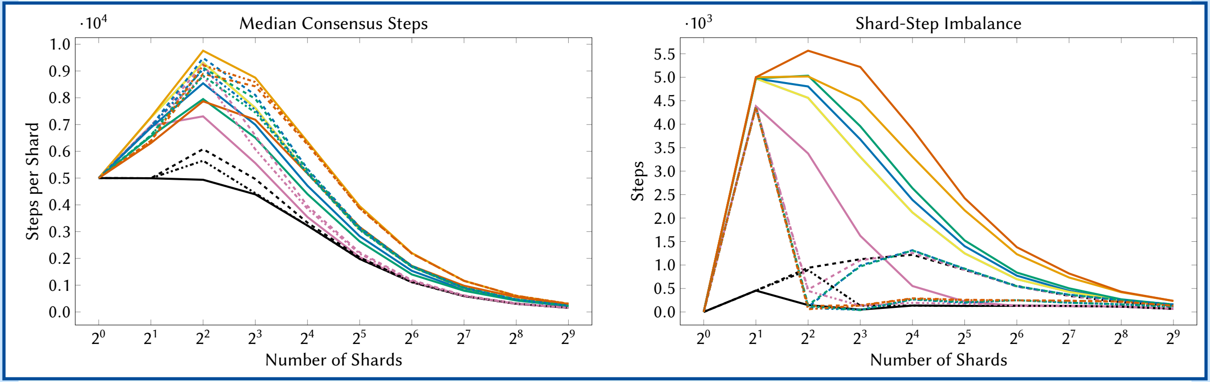
	Isolation-Free execution (write uncommitted)		Lock-based execution					
			Read Uncommitted		Read Committed		Serializable	
	unsafe	*safe*	*blocking*	*non-blocking*	*blocking*	*non-blocking*	*blocking*	*non-blocking*
Linear	LIFu	LIFs	LRUb	LRUnb	LRCb	LRCnb	LSb	LSnb
Centralized	CIFu	CIFs		CRUnb		CRCnb		CSnb
Distributed	DIFu	DIFs		DRUnb		DRCnb		DSnb





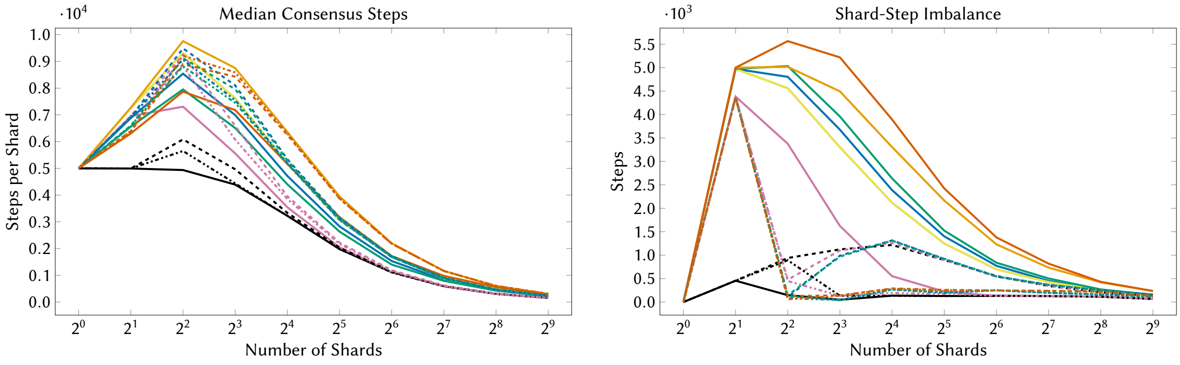
Evaluation

	Isolation-Free execution (write uncommitted)		Lock-based execution: Read Uncommitted		Lock-based execution: Read Committed		Lock-based execution: Serializable	
	unsafe	*safe*	*blocking*	*non-blocking*	*blocking*	*non-blocking*	*blocking*	*non-blocking*
Linear	LIFu	LIFs	LRUb	LRUnb	LRCb	LRCnb	LSb	LSnb
Centralized	CIFu	CIFs		CRUnb		CRCnb		CSnb
Distributed	DIFu	DIFs		DRUnb		DRCnb		DSnb



Evaluation

	Isolation-Free execution (write uncommitted)		Lock-based execution					
			Read Uncommitted		Read Committed		Serializable	
	unsafe	*safe*	*blocking*	*non-blocking*	*blocking*	*non-blocking*	*blocking*	*non-blocking*
Linear	LIFu	LIFs	LRUb	LRUnb	LRCb	LRCnb	LSb	LSnb
Centralized	CIFu	CIFs		CRUnb		CRCnb		CSnb
Distributed	DIFu	DIFs		DRUnb		DRCnb		DSnb



Evaluation

	Isolation-Free execution (write uncommitted)		Lock-based execution					
			Read Uncommitted		Read Committed		Serializable	
	unsafe	*safe*	*blocking*	*non-blocking*	*blocking*	*non-blocking*	*blocking*	*non-blocking*
Linear	LIFu	LIFs	LRUb	LRUnb	LRCb	LRCnb	LSb	LSnb
Centralized	CIFu	CIFs		CRUnb		CRCnb		CSnb
Distributed	DIFu	DIFs		DRUnb		DRCnb		DSnb