14-736: DISTRIBUTED SYSTEMS

LECTURE 16 * SPRING 2019 * KESDEN

ACID

- Atomicity
- Consistency (serializability)
- Isolation
- Durability

ACID

- Acid - "All or nothing“
- Consistency -- This implies two types of consistency. It implies that a single system is consistent and that there is consistency across systems. In other words, if $100 is moved from one bank account to another, not only is it subtracted from one and added to another on one host -- it appears this way everywhere. It is this property that allows one transaction to safely follow another.
- Isolation - Regardless of the level of concurrency, transactions must yields the same results as if they were executed one at a time (but any one of perhaps several orderings).
- Durability - permanence. Changes persist over crashes, &c.

TRANSACTION

- *Transactions* are sequences of actions such that **all** of the operations within the transaction succeed (on all recipients) and their effects are permanantly visible, or none of **none** of the operations suceed anywhere and they have no visible effects; this might be because of failure (unintentional) or an abort (intentional).

COMMIT POINT

- Characterisitically, transactions have a *commit point*.
- This is the point of no return. Before this point, we can undo a transaction. After this point, all changes are permanant. If problems occur after the commit point, we can take compensating or corrective action, but we can't wave a magic wand and undo it.

TRANSACTION EXAMPLE

Plan:

1. Transfer $100 from savings to checking
2. Transfer $300 from money market to checking
3. Dispense $350

```
1. savings -= 100
2. checking += 100
3. moneymkt -= 300
4. checking += 300
5. verify: checking > 350
6. checking -= 350
7. {Commit Point}
8. Dispense $350
```

Why Here?

ATOMIC COMMIT PROTOCOL

- A set of rules that, if followed, will ensure that the transaction commits everywhere or aborts everywhere.
- Most common is "Two-Phase Commit (2PC)"

TWO PHASE COMMIT (2PC)

Coordinator

Participant

---------------------- Phase 1 --

Coordinator:

•Precommit (write to log and.or atomic storage)
•Send request to all participants

Participant:

•Wait for request
•Upon request, if ready:
  - Precommit
  - Send coordinator YES
•Upon request, if *not* ready:
  - Send coordinator NO

Coordinator blocks waiting for ALL replies
(A time out is possible -- that would mandate an ABORT)

---------------------- Phase 2 ---

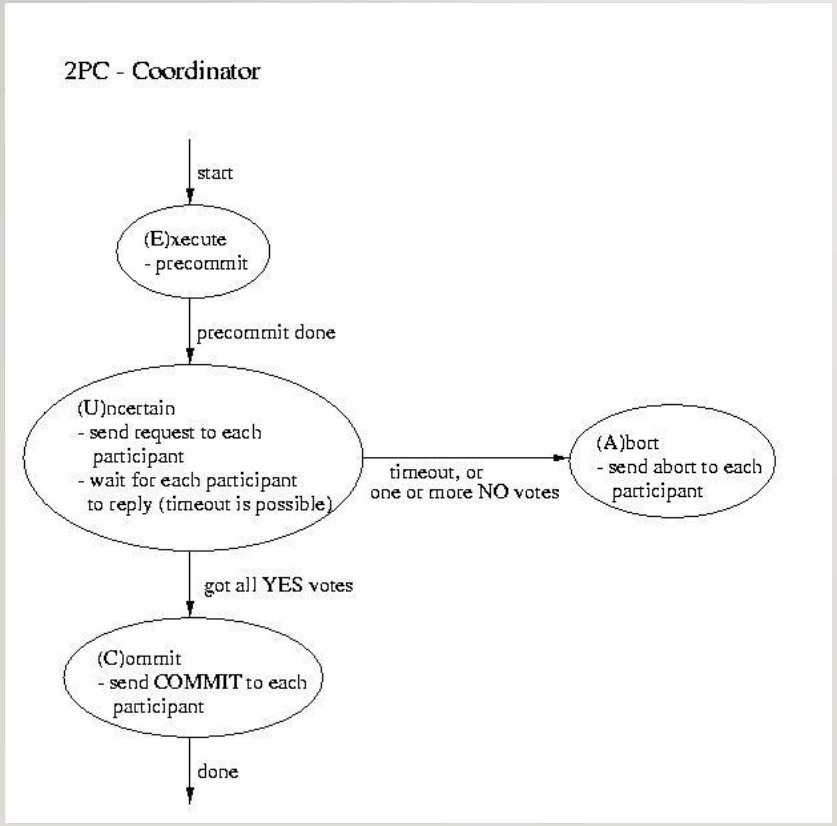
Coordinator:

This is the point of no return!
•If all participants voted YES then send commit to each participant
•Otherwise send ABORT to each participant

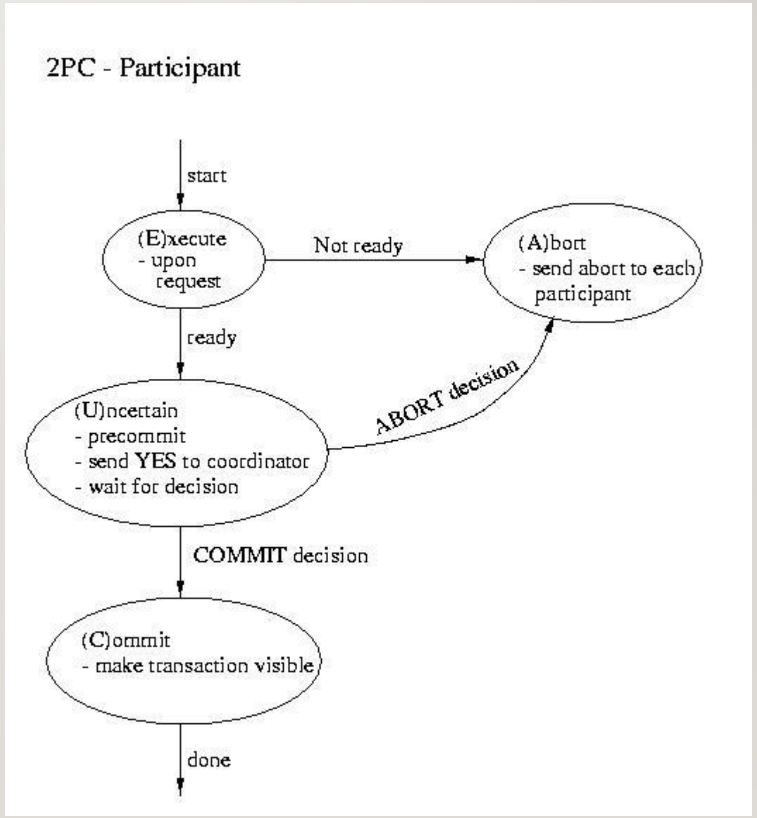
Participant:

Wait for "the word" from the coordinator
•If COMMIT, then COMMIT (transaction becomes visible)
•If ABORT, then ABORT (gone for good)

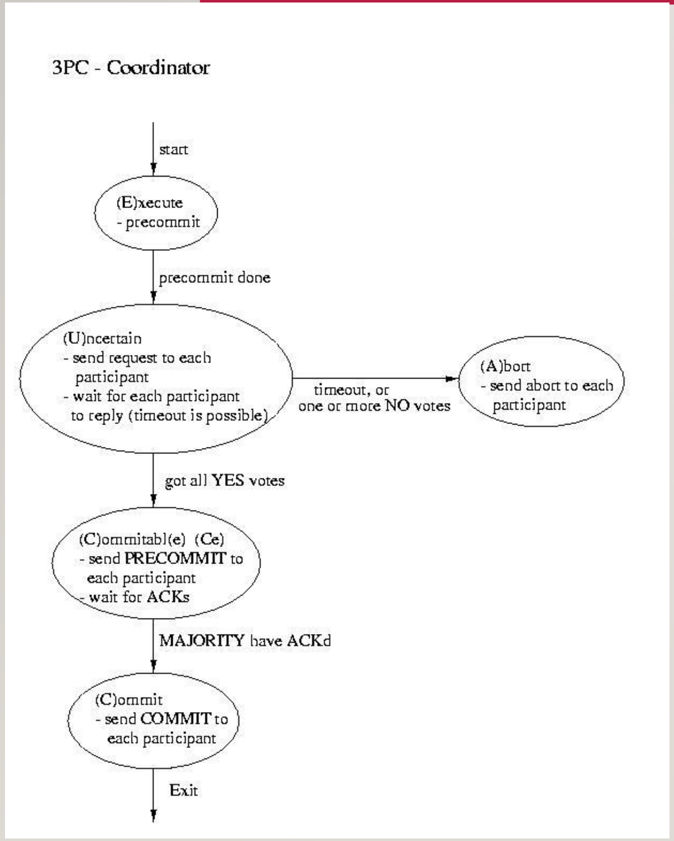
TWO PHASE COMMIT (2PC)

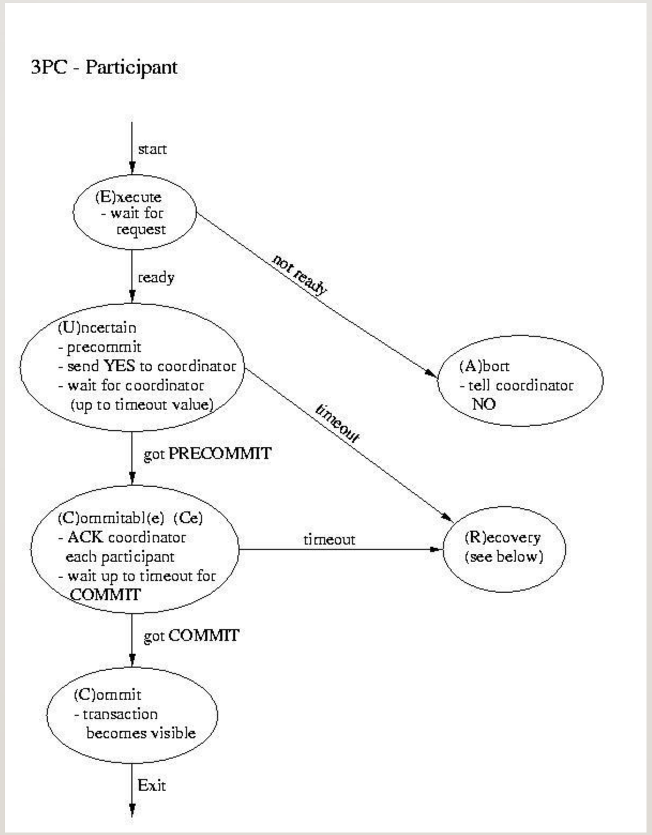


2PC - Coordinator
start
(E)xecute
- precommit
precommit done
(U)ncertain
- send request to each participant
- wait for each participant to reply (timeout is possible)
timeout, or one or more NO votes
(A)bort
- send abort to each participant
got all YES votes
(C)ommit
- send COMMIT to each participant
done



THREE PHASE COMMIT (3PC)


3PC - Coordinator
start
(E)xecute
- precommit
precommit done
(U)ncertain
- send request to each participant
- wait for each participant to reply (timeout is possible)
timeout, or one or more NO votes
(A)bort
- send abort to each participant
got all YES votes
(C)ommitabl(e) (Ce)
- send PRECOMMIT to each participant
- wait for ACKs
MAJORITY have ACKd
(C)ommit
- send COMMIT to each participant
Exit



Another real-world atomic commit protocol is *three-phase commit (3PC)*. This protocol can reduce the amount of blocking and provide for more flexible recovery in the event of failure. Although it is a better choice in unusually failure-prone environments, its complexity makes 2PC a common, if not more common, choice.

THREE PHASE COMMIT (3PC)

- If the participant finds itself in the (R)ecovery state, it assumes that the coordinator did not respond, because it failed. Although this isn't a good thing, it may not prove to be fatal.
- If a majority of the participants are in the uncertain and/or commitable states, it may be possible to elect a new coordinator and continue.
- If any participant has aborted, it sends ABORTs to all (This action is mandatory -- remember "all or none").
- If any participant has committed, it sends COMMIT to all. (This action is mandatory -- remember "all or none").
- If at least one participant is in the commitable state and a majority of the participants are commitable or uncertain, send PRECOMMIT to each participant and proceed with "the standard plan" to commit.
- If there are no committable participants, but more than half are uncertain, send a PREABORT to all participants. Then follow this up with a full-fledged ABORT when more than half of the processes are in the abortable state. PRECOMMIT and abortable are not shown above, but they are complimentary to COMMIT and commitable. This action is necessary, because an abort is the only safe action -- some process may have aborted.
- If none of the above are true, block until more responses are available.

CONCURRENCY AND TRANSACTIONS

- It is desirable to have transactions execute concurrently
  - But they need to execute as if in isolation
- Transactions play with many resources
- We must use concurrency control to protect critical resources
  - Without causing deadlock
- In practice, the key to avoiding deadlock is to avoid "circular wait"
- Want to allow maximum concurrency while ensuring ACID properties

TWO PHASE LOCKING (2PL)

- In a databases class, you'll study many techniques for managing concurrency, many are *optimistic*.
- Here we are only going to talk about the most basic, *Two Phase Locking (2PL)*
  - It is easy to understand and safe, but may not allow as much concurrency as more advanced techniques.
- Protocol:
  - All resources have a precedence and must be acquire din increasing order
  - Growth phase: acquires resources
  - Shrinking phase: releases resources

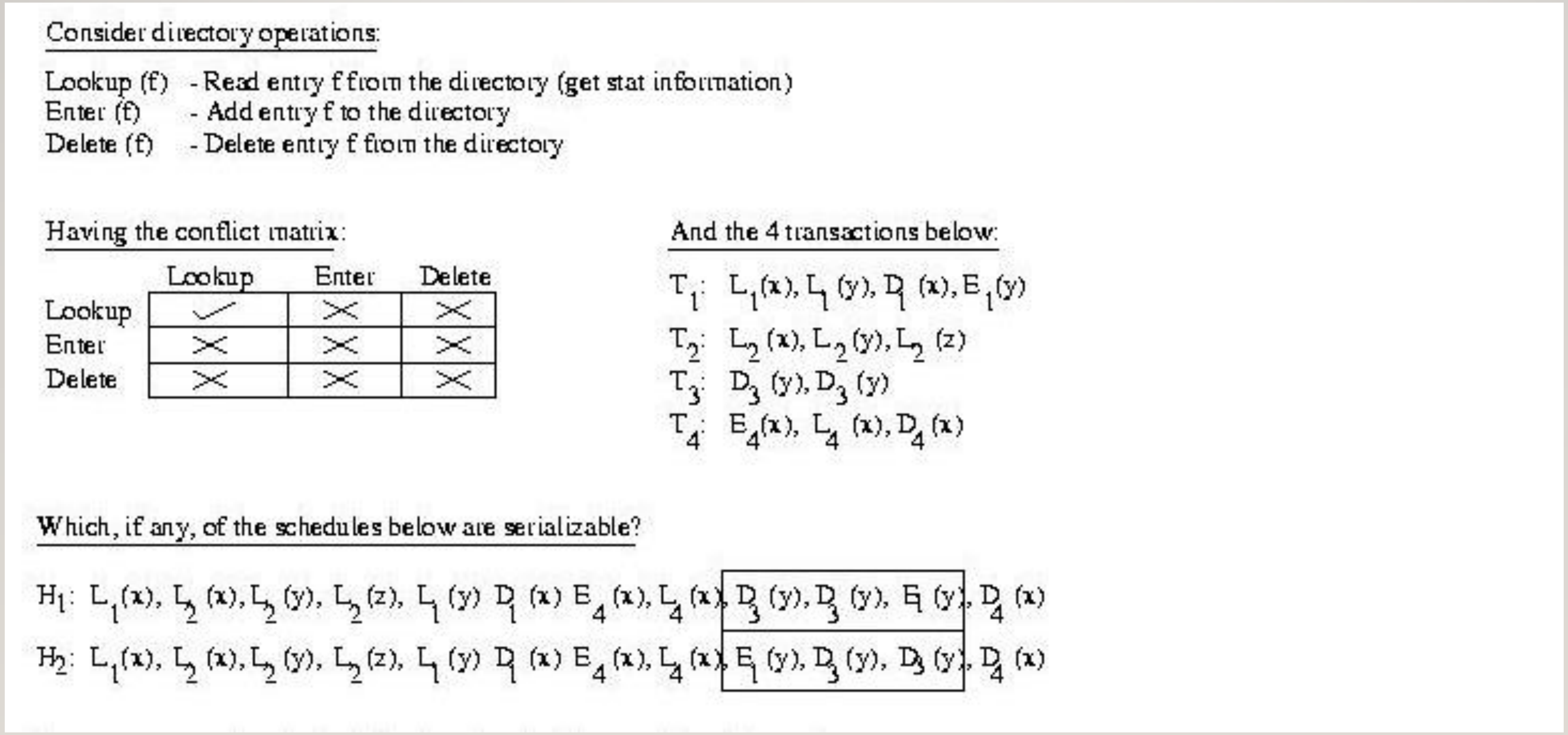
TWO PHASE LOCKING

- Enforcing precedence prevents circular wait
    - No cycles are possible
- Two phase system ensures serializability
    - Equivalent serial schedule

SERIALIZABILITY

- Equivalent Serial Schedule Exists
- “Safe Interleaved Schedule”

SERIALIZABILITY GRAPH



Consider directory operations:

Lookup (f)	- Read entry f from the directory (get stat information)
Enter (f)	- Add entry f to the directory
Delete (f)	- Delete entry f from the directory

Having the conflict matrix:

	Lookup	Enter	Delete
Lookup	✓	✗	✗
Enter	✗	✗	✗
Delete	✗	✗	✗

And the 4 transactions below:

T_1: $L_1(x), L_1(y), D_1(x), E_1(y)$

T_2: $L_2(x), L_2(y), L_2(z)$

T_3: $D_3(y), D_3(y)$

T_4: $E_4(x), L_4(x), D_4(x)$

Which, if any, of the schedules below are serializable?

H_1: $L_1(x), L_2(x), L_2(y), L_2(z), L_1(y)\ D_1(x)\ E_4(x), L_4(x), \boxed{D_3(y), D_3(y), E_1(y)}, D_4(x)$

H_2: $L_1(x), L_2(x), L_2(y), L_2(z), L_1(y)\ D_1(x)\ E_4(x), L_4(x), \boxed{E_1(y), D_3(y), D_3(y)}, D_4(x)$

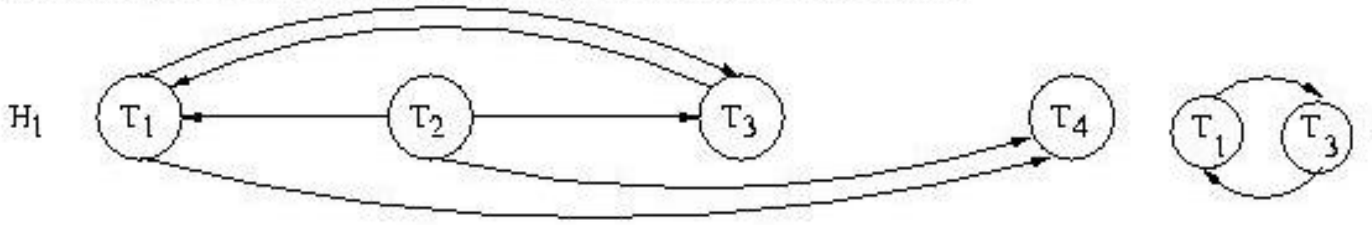
SERIALIZABILITY GRAPH

Which, if any, of the schedules below are serializable?

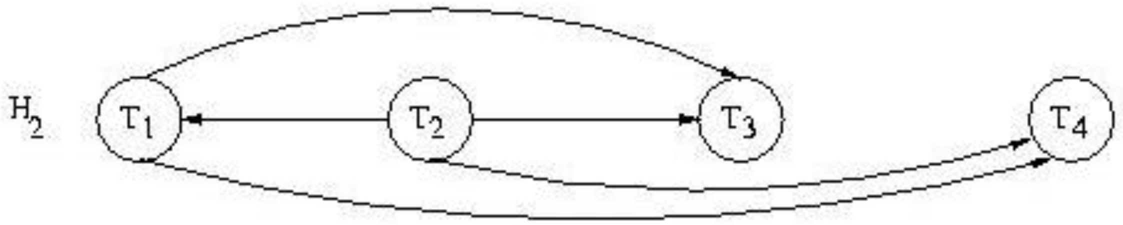
H_1: $L_1(x), L_2(x), L_2(y), L_2(z), L_1(y)\ D_1(x)\ E_4(x), L_4(x), \boxed{D_3(y), D_3(y), E_1(y)}, D_4(x)$

H_2: $L_1(x), L_2(x), L_2(y), L_2(z), L_1(y)\ D_1(x)\ E_4(x), L_4(x), \boxed{E_1(y), D_3(y), D_3(y)}, D_4(x)$

To answer, complete the Serialization Graphs below and check for cycles.



cycle: Not serializable



No cycles (acyclic): Serializable

“DON’T LET THE PERFECT BECOME THE ENEMY OF THE GOOD”

- ACID isn’t always necessary
- Consider shopping on an eCommerce site.
- When does the inventory count need to be perfect?
  - Browsing
  - Putting into cart
  - Check out
  - Charging card

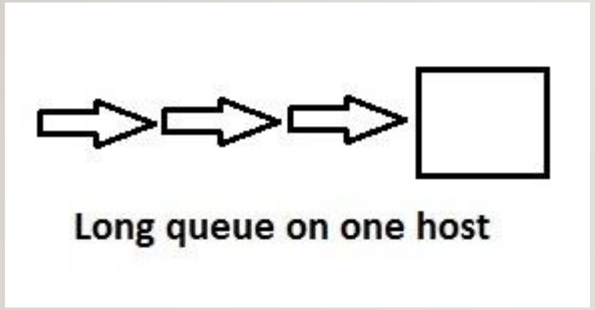
BASE

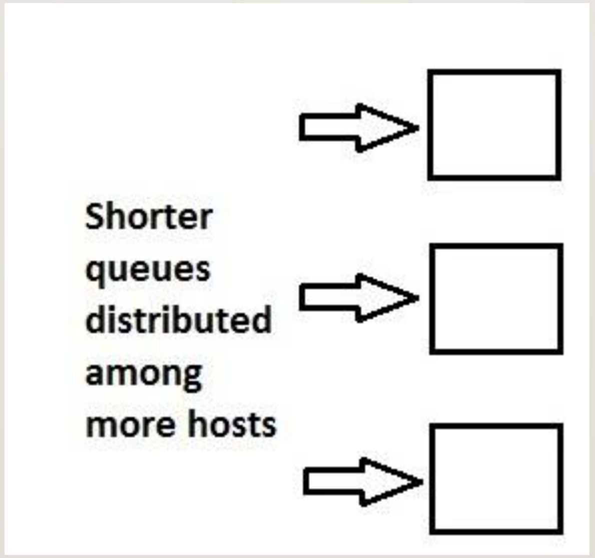
- *Ba*sically *A*vailable means that small failures don't generate large disabilities. It is the same idea as what we call "soft failure" vs "hard failure", but with the added emphasis that a few failures in a large scale system shouldn't really be noticeable.
- *S*oft state is usually intended to convey state that can be generated or refreshed upon demand, rather than necessarily being stored as "hard state". But, in this case, it is being used to convey that values, even after written, will continue to change without any explicit user request. Specifically, they'll propagate out slowly.
- *E*ventual consistency conveys the idea that, although the system might be inconsistent for some time after an update, it will eventually converge to consistency. Without this property, or an approximation thereof, what good would the system be?

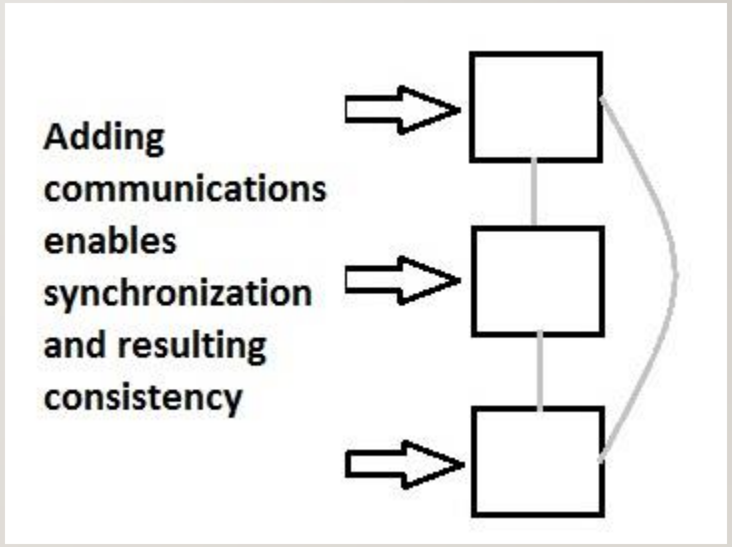
CAP/BREWER CONJECTURE

- It is commonly desirable for distributed systems to exhbit *Consistency*, *Availability*, and *Partition tolerance*.
- By *consistency* we mean that all participating systems share the same view of the data.
  - For example, if one system observes the value five, all systems would, if they looked, observe the value five at that time. None, for example, would be more stale or more fresh than others.
- By *Availability* we mean that the system is able to respond quickly enough for the user's needs.
  - For example, if a Web page times out, or users abort before seeing the results, it is not available.
- By *Partition tolerance* we mean that, in the event of the failure or isolation of some participants, the other participants can continue to do whatever they can.
  - For example, the loss of certain nodes might necessitate disconnecting certain clients or the inability to return certain results -- but should not unduely interfere with the ability of the functioning nodes to service clients and/or return results.

CAP/BREWER CONJECTURE






Adding communications enables synchronization and resulting consistency