Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov

Logistics

- Midterm 2: April 12-14
  - See my Campuswire post #260
  - Syllabus: everything covered post misterm1 syllabus, upto and including Raft
    - multicast, mutual exclusion, leader election, synchronous consensus, Paxos, and Raft.

Agenda for today

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *We will not cover the proof in details*
  - Good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus algorithm
    - Raft (log-based consensus)
    - **Block-chains (distributed consensus)**

Bitcoins

- Implement a *distributed* replicated state machine that maintains an *account ledger (= bank).*
  - No user should be able to "double-spend".
  - Need to know of all transactions to validate this.
  - Who does this validation? Cannot trust a single central authority.
    - Any participant (replica) should be able to validate.
  - All replicas must agree on the single history on transaction ordering.
- Scale to thousands of replicas distributed across the world.
- Allow old replicas to fail, new replicas to join seamlessly.
- Withstand various types of attacks.

Basic Idea

Transactions grouped into a *block* that gets added to the *chain* (history of transactions) by the "leader of that block".

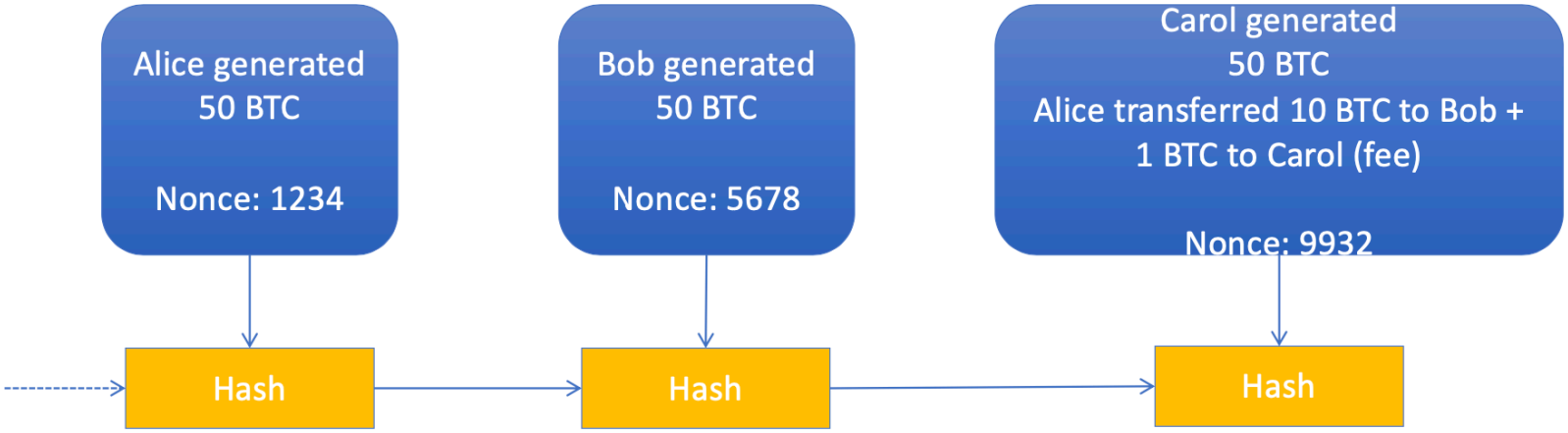
Iterated Hashing / Proof of work

- Repeat:
  - Pick random x, compute y = H(seed, x)
  - If y<T, you win!
- Set threshold T so that on average, one winner every few minutes
- Given a *solution*, x such that H(seed, x) < T, anyone can *verify* the solution in constant time (microseconds).

Using a seed

- Every node picks x, computes H(seed, x)
- What to use as a seed?
- Hash of:
  - Previous log
  - Node identifier
  - New messages to add to log
- The appropriate value of x *changes* for each block.
  - Cannot be reused across blocks.
  - Must be recomputed.

Chaining the blocks



Account	Balance
Alice	39 BTC
Bob	60 BTC
Carol	51 BTC

Protocol Overview

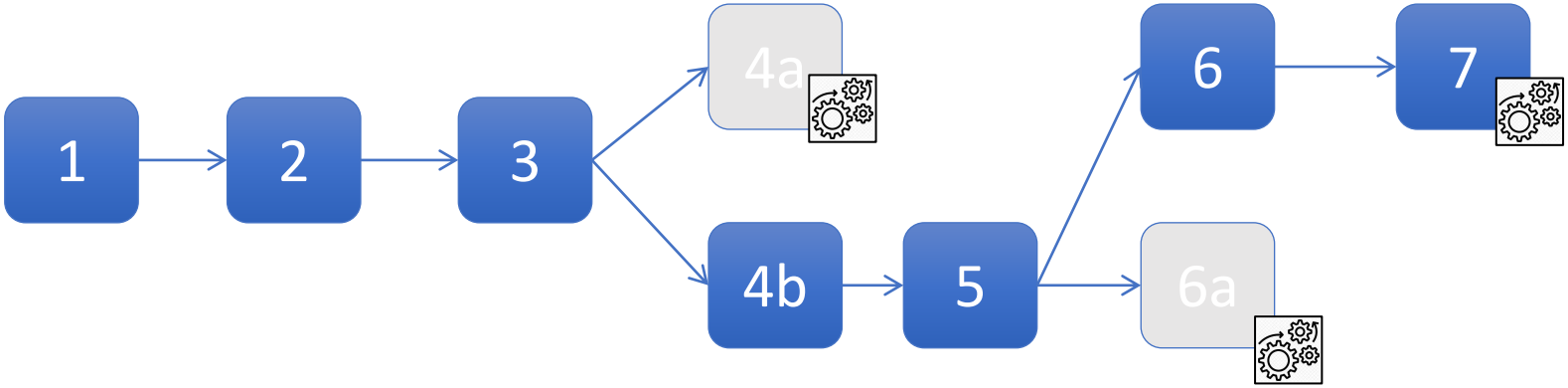
- New transactions broadcast to all nodes.
- Each node collects new transactions into a block.
- Each node works on finding a proof-of-work for its block to become its leader and get it appended to a chain.
  - i.e. finds x, such that $H(seed, x) < T$.
- When a node finds a proof-of-work, it broadcasts it to all nodes.
- Nodes accept a block only if all transactions in it are valid.
- Nodes express their acceptance by working on creating the next block in the chain, using the hash of accepted block as previous hash.

What could go wrong?

- Two different leaders for the next block!
  - Two nodes may end up mining different versions of the next block.

Longest Chain Rule

- Two nodes may end up mining different versions of the next block.
- A node may receive two versions of the next block.
- Will store both, but work on the first one they receive.
- Over time, more blocks will be received.
- The node will switch to working on the *longest chain*.



When is a transaction committed?

- Wait for upto k more blocks to be added in the chain.
- Then commit the transaction.
- k is set to 6 for Bitcoins.

Security Property

- Majority decision is represented by the longest chain.
  - It has greatest "proof-of-work" invested in it.

- If majority of CPU power is controlled by honest nodes, the honest chain will grow fastest and outpace competing chains.

- To modify past blocks, an attacker will need to redo the proof-of-work for that block, and all blocks after it, and then surpass the work of honest nodes.

- Probability of attack reduces as more blocks get added.

Incentives for Logging

- Security better if more people participated in logging.

- Incentivize users to log *others'* transactions
  - Transaction fees: e.g. pay me x% to log your data (or some fixed fee per transaction)
  - Mining reward: each block *creates* bitcoins

Logging Speed

- How to set T?
  - Too small: slows down transactions
  - Too big: wasted effort due to chain splits

- Periodically adjust difficulty T such that one block gets added every 10 minutes.
  - Depends on hardware speed (which typically improves over time) and number of participants (which vary over time).

- Determined algorithmically based on the rate at which blocks are mined
  - Target is 1 block every 10minutes.
  - Difficulty recomputed after every 2016 blocks.

Bitcoin Broadcast

- Need to broadcast:
  - Transactions to all nodes, so they can be included in a block.
  - New blocks to all nodes, so that they can switch to longest chain.

- Is R-multicast a suitable option?
  - Have to send O(N) messages
  - Have to *know* which nodes to send to
  - Not a suitable choice.

Gossip / Viral propagation

- Each node connects to a small set of *neighbors* (10–100).
- Nodes propagate transactions and blocks to neighbors.
- Push method: when you hear a new tx/block, resend them to all (some) of your neighbors (flooding).
- Pull method: periodically poll neighbors for list of blocks/tx's, then request any you are missing.
- Unreliable: some nodes may not receive all transactions or all blocks. But that's ok.

Maintaining Neighbors

- A *seed* service
  - Gives out a list of random or well-connected nodes
  - E.g., seed.bitnodes.io
- Neighbor discovery
  - Ask neighbors about *their* neighbors
  - Randomly connect to some of them

Bitcoin Summary

- Unreliable broadcast using gossip
- Probabilistic "leader" election for mining blocks (tx ordering)
- Longest chain rule to ensure long-term (probabilistic) consistency and security

- Compared with Paxos/Raft:
  - Scales to thousands of participants, dynamic groups
  - Tens of minutes to successfully log a transaction (vs. milliseconds)
  - Ensures safety with high probability (but not 100%)

Agenda for the next 2-3 classes

- Transaction Processing and Concurrency Control
  - Chapter 16
    - Transaction semantics: ACID
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering

- First focus on transactions executed on a single server.
- Look into distributed transactions later (Chapter 17)

Transaction

- Series of *operations* executed by a client on a server (or a set of servers).

- Example: Switch from T4 to T3 section:

  rosters.remove("ece428", "t4", student.name)
  student.schedule.remove("ece428", "t4")
  student.schedule.add("ece428", "t3")
  rosters.add("ece428", "t3", student.name)

Transaction

- Another example:

Client code:

```
int transaction_id = openTransaction();
x = server.getFlightAvailability(ABC123, date);      // read(ABC123, date)
if (x > 0)
    y = server.bookTicket(ABC123, date);             // write(ABC123, date)
closeTransaction(transaction_id);
```

Transaction Properties

- Atomic: all-or-nothing
  - Transaction either executes completely or not at all
- Consistent: required rules are maintained
- Isolation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- Durability: values preserved even after crashes

Transaction Properties

- **A**tomic: all-or-nothing
  - Transaction either executes completely or not at all
- **C**onsistent: required rules maintained
- **I**solation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- **D**urability: values preserved even after crashes

ACID properties

Atomicity

- All-or-nothing
  - Transaction either executes completely or not at all

- What can happen after partial execution?

rosters.remove("ece428", "t4", student.name)
student.schedule.remove("ece428", "t4")
student.schedule.add("ece428", "t3")
rosters.add("ece428", "t3", student.name)

Atomicity

- All-or-nothing
  - Transaction either executes completely or not at all
- Make tentative updates to data.
- **Commit** transaction to make tentative updates permanent.
- **Abort** transaction to roll back to previous values.

Consistency

Various rules about state of objects must be maintained:

- E.g. class enrollment limit, schedule can't conflict
- Account balances have to stay positive
- Consistency must be maintained at end of transaction.
- Checked at commit time, abort if not satisfied

rosters.remove("ece428", "t4", student.name)
student.schedule.remove("ece428", "t4")
student.schedule.add("ece428", "t3")
rosters.add("ece428", "t3", student.name)

Durability

- Committed transactions must persist:
  - Client crashes
  - Server crashes
- How do we ensure this?
  - Permanent storage
  - Replication

Isolation

Multiple clients may execute transactions concurrently on one server.

What could go wrong?

What could go wrong?

Transaction T1

```
x = getSeats(ABC123);
        // x = 10
if(x > 1)
   x = x – 1;
write(x, ABC123);



commit
```

Transaction T2

```
x = getSeats(ABC123);
if(x > 1)                  // x = 10



    x = x – 1;
write(x, ABC123);



commit
```

At Server: seats = 10

seats = 9

seats = 9

1. Lost Update Problem

Transaction T1	**Transaction T2**	At Server: seats = 10
x = getSeats(ABC123);		
// x = 10	x = getSeats(ABC123);	
if(x > 1)	if(x > 1) // x = 10	
x = x – 1;		
write(x, ABC123);		seats = 9
	x = x – 1;	
	write(x, ABC123);	seats = 9
commit		
	commit	

T1's or T2's update was lost!

What else could go wrong?

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
// **ABC123 = 5 now**

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);
// **x = 5, y = 15**

print("Total:" x+y);
// **Prints "Total: 20"**
commit

At Server:
ABC123 = 10
ABC789 = 15

2. Inconsistent Retrieval Problem

Transaction T1

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
// **ABC123 = 5 now**

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);
// **x = 5, y = 15**

print("Total:" x+y);
// **Prints "Total: 20"**
commit

At Server:
ABC123 = 10
ABC789 = 15

T2's sum is the wrong value! Should have been "Total: 25"

Isolation

Multiple clients executing transactions concurrently on one server.

What could go wrong?

- Lost Update Problem
- Inconsistent Retrieval Problem

How to prevent transactions from affecting each other?

Isolation

How to prevent transactions from affecting each other?

- Option 1: Execute them serially at the server (one at a time).
  - Grab a global lock before executing any transaction, release the lock after the transaction has committed (or aborted).
- But this reduces number of concurrent transactions
  - *Transactions per second* directly related to revenue of companies
    - This metric needs to be maximized

Goal: increase concurrency while maintaining correctness (ACID).

Concurrent Transactions

Goal: increase concurrency while maintaining correctness (ACID).

- How do we increase concurrency?
  - Instead of targeting strict serial execution, target **serial equivalence**.

Interleaving

- An ordered sequence of the operations across multiple transactions, where each transaction's operations follows the order defined by the transaction.

- E.g., if $T_1 = \{op_1, op_2, op_3\}$ and $T_2 = \{op^4, op^5, op^6\}$

  then $O = \{op_1, op_2, op^4, op_3, op^5, op^6\}$ is a sample interleaving.

Interleaving: Another example

Transaction T1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);

write(y+5, ABC789);

commit
```

Transaction T2

```
x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit
```

Interleaving

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
x = getSeats(ABC123);
y = getSeats(ABC789);
write(y+5, ABC789);

print("Total:" x+y);
commit
commit
```

Concurrent Transactions

- Allowing transaction operations to be interleaved with one-another increases concurrency.

- To avoid transactions from affecting one another, the interleaving of operations across transactions must be *serially equivalent*.

Serial Equivalence

- An interleaving (say O) of transaction operations is serially equivalent iff (if and only if):
  - There is some ordering (O') of those transactions, one at a time,
  - where the operations of each transaction occur consecutively (in a batch)
  - which gives the same end-result (for all objects and transactions) as the original interleaving O
- Says: Cannot distinguish end-result of given interleaving O from a (fake) *serial* interleaving O'
- E.g., if $T_1 = \{op_1, op_2, op_3\}$ and $T_2 = \{op^4, op^5, op^6\}$

  then O = $\{op_1, op_2, op^4, op_3, op^5, op^6\}$ is a serially equivalent, if:
  - end result of O is same as $\{op_1, op_2, op_3, op^4, op^5, op^6\}$
  - **Or** end result of O is same as $\{op^4, op^5, op^6, op_1, op_2, op_3\}$

Concurrent Transactions

- To be continued in next class….