Lesson 7: Failure and Consensus

Phillip J. Windley, Ph.D.
CS462 – Large-Scale Distributed Systems

Contents

Lesson 07: Failure and Consensus

Introduction

Consensus is among the most fundamental problems in distributed systems.

As an example, consider Dropbox and other file sharing services. Their goal is for the files on my laptop, their cloud server, and any other machines I own to be the same. That is, they want consensus on what files are there and what contents they have.

There are various ways to define consensus and lots of consensus algorithms, some tuned to specific problems. Consensus is easy to achieve when everything works perfectly. Consequently, good algorithms focus on consensus in the presence of failure.

In this lesson, we'll first consider transactions which allow closely-coordinated processes to achieve consensus on values about failure itself and maintain a consistent state.

Transactions are too expensive for most distributed systems where recognizing failure and handling it correctly requires different techniques.

Beyond outright failure, synchronizing values is an interesting problem. We'll look at Paxos as an example of a consensus algorithm.

Synchronization when some actors are malicious is an important problem known as the Byzantine Generals Problem. Bitcoin is an example of how Byzantine faults can be overcome.

Lesson Objectives

After completing this lesson, you should be able to:

1. Describe the relationship between failure and consistency.
2. Understand ACID properties and the levels of isolation in transactions.
3. Explain the limitations of classic transactions in distributed systems and the options for handling failure.
4. Describe the Paxos algorithm, understand how it achieves consensus, and how it might fail.
5. Describe byzantine failure. Understand how the blockchain is designed to overcome byzantine failure.

Reading

Read the following:

- ACID from Wikipedia (http://en.wikipedia.org/wiki/ACID)
- Isolation from Wikipedia (http://en.wikipedia.org/wiki/Isolation_%28database_systems%29)
- Starbucks Does Not Use Two-Phase Commit (PDF) by Gregor Hohpe (http://www.eaipatterns.com/docs/IEEE_Software_Design_2PC.pdf)
- Paxos Made Simple (PDF) by Leslie Lamport (http://cseweb.ucsd.edu/classes/sp11/cse223b/papers/paxos-simple.pdf)
- Paxos By Example by Angus MacDonald (http://angus.nyc/2012/paxos-by-example/)
- Byzantine fault tolerance from Wikipedia (https://en.wikipedia.org/wiki/Byzantine_fault_tolerance)
- How the Bitcoin protocol actually works by Michael Nielsen (http://www.michaelnielsen.org/ddi/how-the-bitcoin-protocol-actually-works/)

Additional Resources

Additional Resources:

- The Transaction Concept: Virtues and Limitations (PDF) by Jim Gray (http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf)
- How to GET a Cup of Coffee (http://www.infoq.com/articles/webber-rest-workflow) - also assigned in Lesson 5: APIs
- The Part-Time Parliament (PDF) by Leslie Lamport (http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf)
- Lecture 10. Unit 2 Paxos-Algorithm by Seif Haridi (https://www.youtube.com/watch?v=5scBtoyz8HU)
- Bitcoin: A Peer-to-Peer Electronic Cash System (PDF) by Satoshi Nakamoto (https://bitcoin.org/bitcoin.pdf)
- The Future of Money by Mike Hearn (https://www.youtube.com/watch?v=Pu4PAMFPo5Y)
- A Social Operating System by Stephan Tual (https://medium.com/ursium-blog/a-social-operating-system-e768072e1b84)

Transactions

Computer systems have been using transactions to ensure consistency since the 1960’s.

Booking Passage on a Cruise

Suppose you're building a system for taking reservations on a cruise ship.

You can think of *booking passage* as an operation that involves three different entities:

- **reservation**—the process responsible for storing the reservation details like customer, cabin, cruise, and price.
- **payment**—the process responsible for processing the credit card
- **ticket**—the process that creates the ticket for the customer.

The Travel Agent

The `bookPassage()` operation happens in an entity representing the travel agent. This operation completes the booking by creating and storing the reservation, processing the credit card, and finally creating and returning the ticket.

```
public TicketDO bookPassage(CreditCardDO card, double price)
    try {
        Reservation reservation = new Reservation(customer, cruise, cabin, price);
        entityManager.persist(reservation);
        this.processPayment.byCredit(customer, card, price);
        TicketDO ticket = new TicketDO(customer, cruise, cabin, price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

All or Nothing

The travel agent entity should only return a ticket if all three sub-processes complete successfully.

A *transaction* is a sequence of operations that must all complete successfully or leave the system in the same state it was before the sequence began.

Transactions occur regularly in computations.

This all or nothing property is called *atomicity*. Without atomicity, computations fail in non-deterministic ways.

For example, in our travel agent example, we might fail to collect the money for the tickets, issue tickets that aren't associated with a valid reservation and end up double-booking, or take the money and not deliver a ticket.

A Thought Exercise

For this exercise, consider the code for the travel agent entity we saw earlier.

Assume that each of the three sub processes throw an exception when they fail.

- What would you change in the code to protect yourself from a failure when creating the reservation?
- How about the payment or ticketing processes?
- Can you write try-catch blocks that ensure atomicity?
- What makes this difficult?
- Would it be easier if there were just two sub processes?

ACID Properties

In addition to atomicity, transactions can have other important properties including *consistency*, *isolation*, and *durability*. Together these are known as the ACID properties.

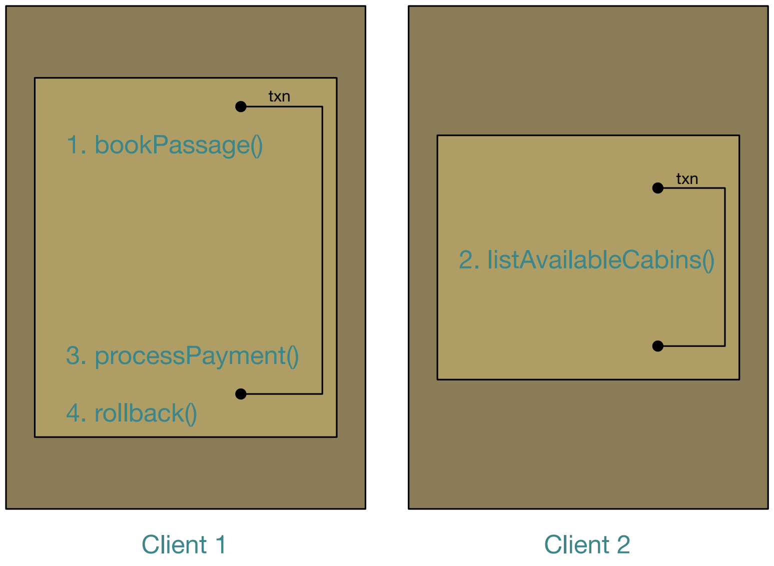
Atomicity—a transaction executes completely or not at all

Consistency—a transaction always maintains the integrity of the underlying data store (i.e. the data is in harmony with the real world that it models)

Isolation—the operations in the transaction are not affected by operations outside the transaction

Durable—the changed made by the operations in the transaction must persist in the face of system crashes

Dirty Reads



An example of a dirty read is shown in the figure.

Suppose Client 1 begins to book passage and reserves a cabin.

Subsequently, Client 2 asks for available cabins and doesn't see the cabin that Client 1 reserved.

Sometime goes wrong with the payment and Client 1 rolls back the transaction.

Client 2 has an invalid list of cabins because the cabin Client 1 had reserved is now available due to the rollback.

Client 2 saw the change Client 1 made.

Non-repeatable Reads

The figure shows an example of a non-repeatable read.

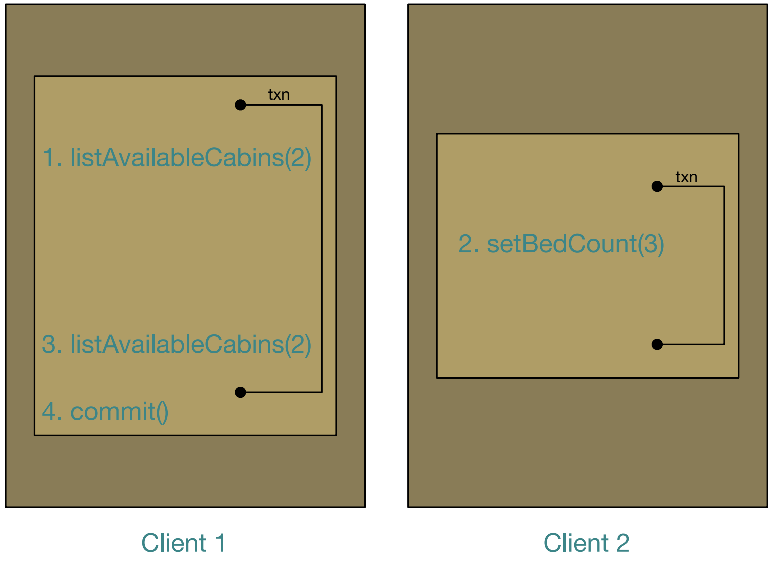
Client 1 gets a list of cabins with two beds.

Client 2 updates the bed count to one of the cabins in the list Client 1 has to 3.

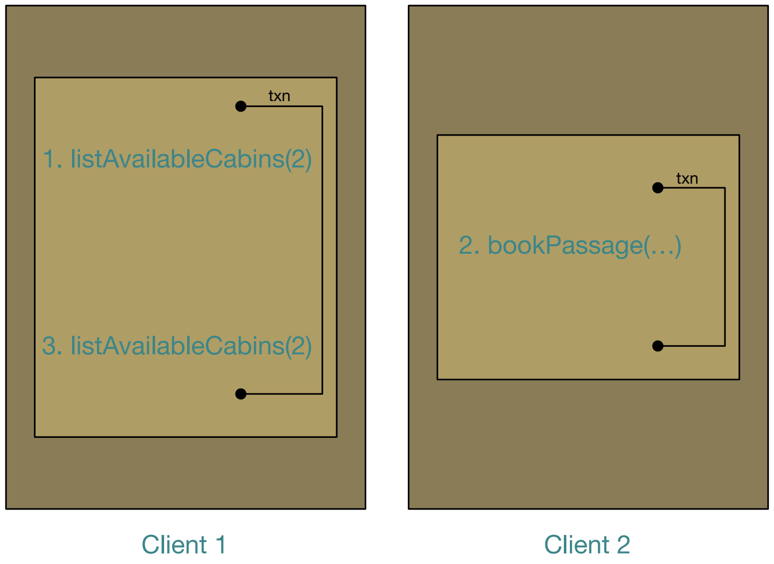
Client 1 again asks for a list of cabins with two beds.

If the second list is different than the first, then Client 1 had a non-repeatable read.

Non-repeatable reads are relevant to information in a database row that changes.



Phantom Reads



The figure shows an example of a phantom read.

Client 1 asks for a list of cabins with two beds.

Client 2 books passage in a cabin with two beds.

Client 1, in the same transaction as before lists cabins with two beds and the cabin Client 2 booked is not in the list.

Phantom reads are relevant to rows that are inserted or deleted from a table.

Isolation Levels

Isolation levels declare the kinds of reads that are permitted. These are listed from least restrictive to most restrictive.

Read uncommitted—the transaction can read uncommitted data allowing dirty, non-repeatable, and phantom reads.

Read committed—the transaction cannot read uncommitted data. Dirty reads are prevented, but non-repeatable and phantom reads can still occur.

Repeatable read—the transaction cannot change data that is being read by another transaction. Dirty and non-repeatable reads are prevented. Phantom reads can still occur.

Serializable—the transaction has exclusive read and write access to the data. Dirty, non-repeatable, and phantom reads are prevented.

Isolation Levels and Reads

The table shows the relationship between isolation levels and the kinds of reads that are allowed.

	Dirty	Non-Repeatable	Phantom
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed		Allowed	Allowed
Repeatable Read			Allowed
Serializable			

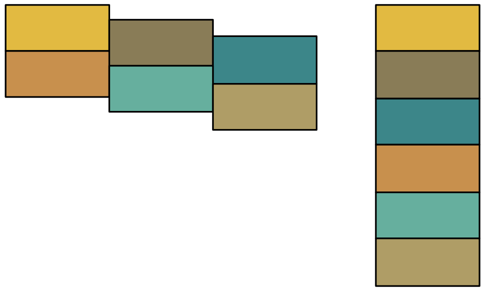
Balancing Consistency and Performance

As transaction become more and more restrictive, performance decreases.

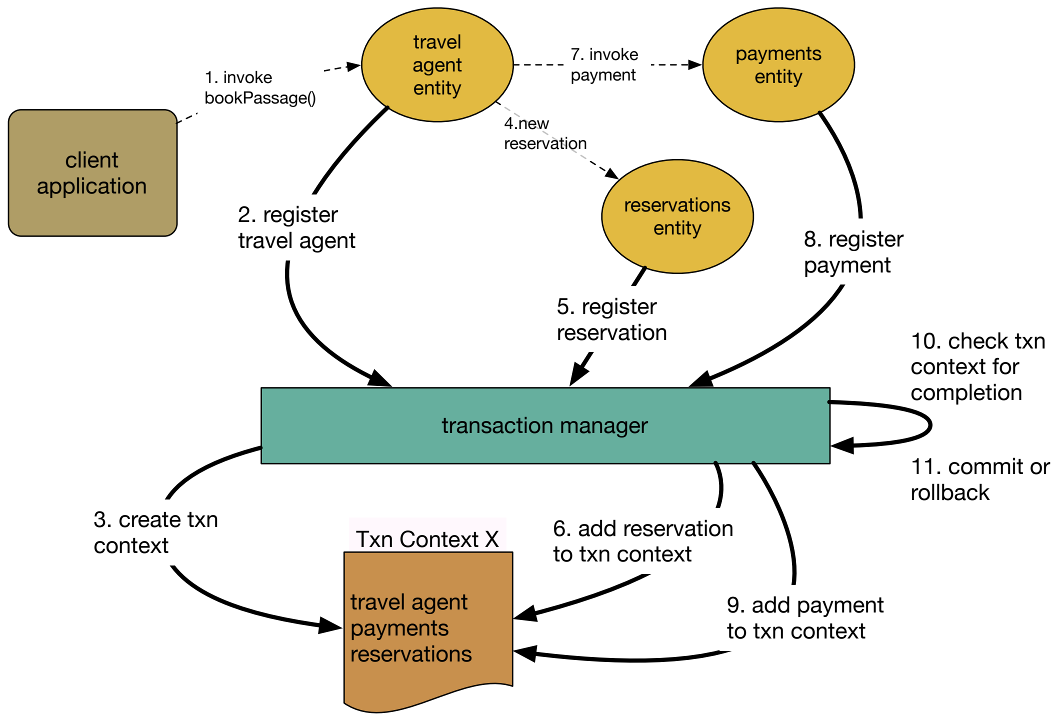
Consistency and performance are trade offs. Shared resource contention is a major cause of performance problems because it creates a bottleneck and decreases scalability.

You may initially think it's a good idea to serialize everything. But serializing transactions does exactly what it's name implies: each transaction must wait for the previous one to finish before it can start. We get concurrency without parallel execution.

Instead, each operation should be evaluated for what isolation level is appropriate. Sometimes it's easy to fix the problems caused by lack of isolation than it is to solve the performance problems caused by extreme isolation.



Implementing Transactions



Transaction managers create and manage transactions automatically in systems that support them using a consensus protocol known as *two-phase commit*.

This figure shows how a transaction manager would work in the travel agent example.

The transaction manager is in charge of telling each entity whether to complete (commit) or not after all of them have reported based on a record called the transaction context.

The entities must support standardized interactions with the transaction manager for this to work.

Failure and Distributed Systems

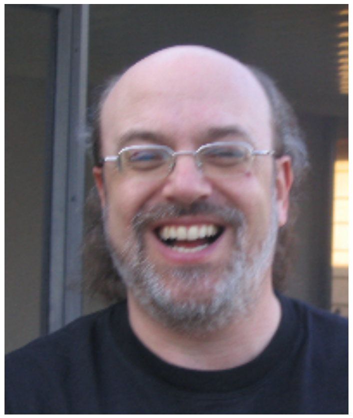
Failure is a defining chracteristive of distributed systems. Handling failure correctly distinguishes well-designed distributed systems from the poorly designed.

Failure Happens All the Time

“Failure is the defining difference between distributed and local programming, so you have to design distributed systems with the expectation of failure. Imagine asking people, ‘If the probability of something happening is one in 10^{13}, how often would it happen?’ Common sense would be to answer, ‘Never.’ That is an infinitely large number in human terms. But if you ask a physicist, she would say, ‘All the time. In a cubic foot of air, those things happen all the time.’

“When you design distributed systems, you have to say, ‘Failure happens all the time.’ So when you design, you design for failure. It is your number one concern.”

—Ken Arnold



Strategies for Failure: Learning from Starbucks

When you order a drink at Starbucks, the cashier writes your name on the cup and puts it in a queue. Baristas pick cups from the queue, not necessarily in order, and fill the order. Several things could go wrong:

- Your payment could fail
- The barista could make the wrong drink
- The machine could fail or they could run out of an ingredient

Starbucks has a strategy for dealing with each of these problems:

- **Write-off**—if the payment fails they don't make the drink or throw it away if it's already made.
- **Retry**—if the barista makes the wrong drink, they start over and remake it.
- **Compensate**—if a machine fails or they can't make your order, they refund your money.

Starbucks doesn’t take your order, validate that you can pay, make your drink, and after it has been successfully made, take your money.

Why?

Exception Handling

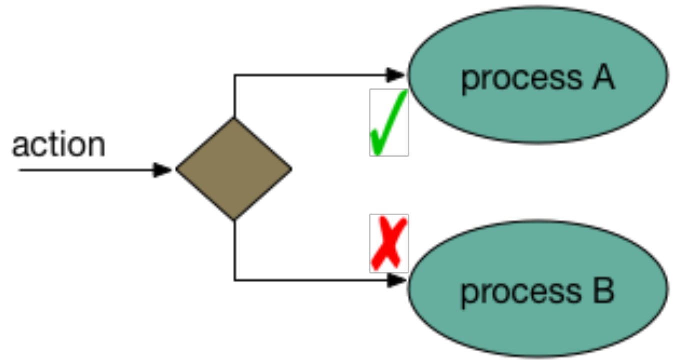
Starbucks handles exceptions to maximize throughput rather than using a transaction-like strategy to prevent waste.

Starbucks and many other companies use exception handling to recover from failure because the alternative is too expensive. Starbuck's major concern is throughput and the problems are rare enough that handling them as exceptions is good enough.

Taking your order and verifying your payment and then waiting for the order to be processed before actually taking payment is just too much overhead. Of course, when the stakes are larger, a traditional *two-phase commit protocol* can be used.

As we saw, two-phase commit interactions involve a number of messages to the transaction manager—not to get things done, but simply to manage the transaction—as the latency between processes in a disitributed system increases, two-phase commit becomes more and more expensive.

Write-Off

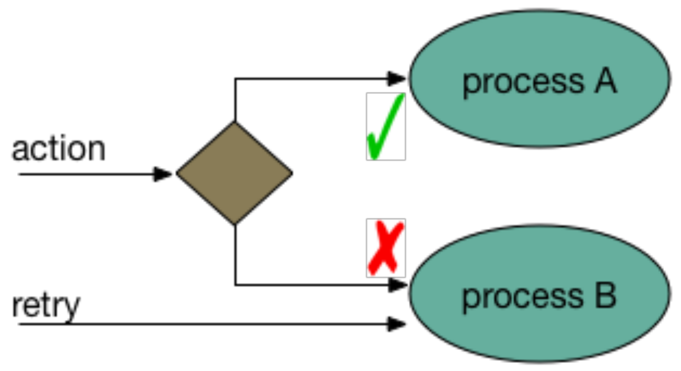


Writing off a loss may seem like a waste, but in some cases that may be the best strategy.

For example, in one system I built, several processes needed to complete in order to build a complete report. But it wasn’t critical if some of the data was missing and the chances of failure were small, so we decided to just do nothing when failure occurred.

This made the code simpler and left time for working on other features that really made a difference.

Retry



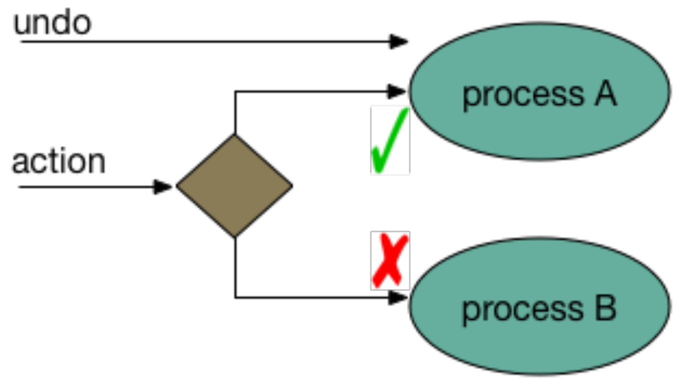
When the chances for success on subsequent calls are good, then retrying a failed action may be the best strategy.

Retrying asks a failed process to try again. If the failure was due to a timeout, failed message delivery, or some other temporary condition, this can be a good strategy.

Retying risks running the operation twice (i.e. the failure may be only that a completion signal wasn't received). Idempotency is important in retrying.

Retrying depends on knowing that something went wrong. For example in the case of failed message delivery, you may need a time out to more the process along.

Compensating Actions



Compensating for a failed action asks the processes that succeeded to undo their action.

In the same way that retrying depends on idempotency, compensating depends on reversibility. The undo actions have to be built into the software. Programmers have to also take into account the possibility that the undo might fail.

Conversations and Protocols

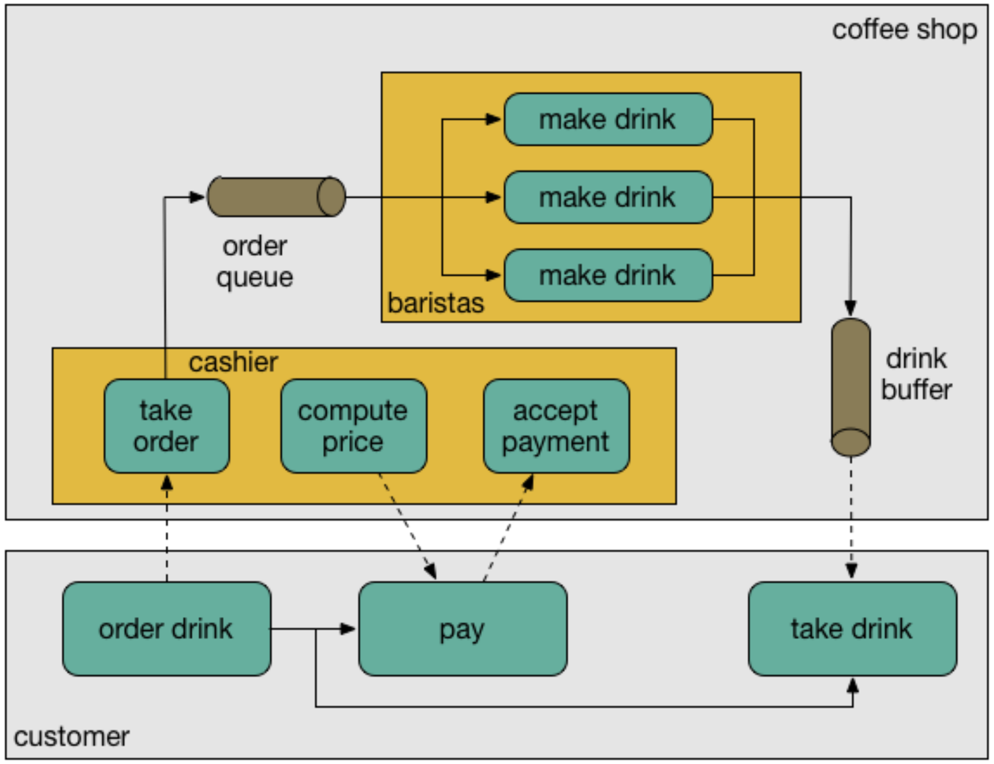
Interactions between asynchronous processes resemble a conversation.

The customer and the cashier engage in a synchronous interaction to pay for the drink.

The cashier converses with the barista using an order queue.

The barista engages the customer by buffering drinks on the counter.

We call a scripted conversational style a *protocol*.



Correlation Identifiers

Parties to an asynchronous conversation have to be able to identify the subject of the conversation.

For example, in the case of the coffee shop, the customer picks up the drink asynchronously from paying. How do they know which drink is theirs? Starbucks writes their name on the cup so the barista can call is out as the drink is placed on the counter.

The customer's name on the cup is an example of a *correlation identifier*. Asynchronous processes make use of correlation identifiers to link conversational state when the conversation is picked up after a hiatus.

Leaderless Election Algorithms

Paxos is a famous algorithm for achieving consensus in distributed systems.

Getting Rid of the Locks

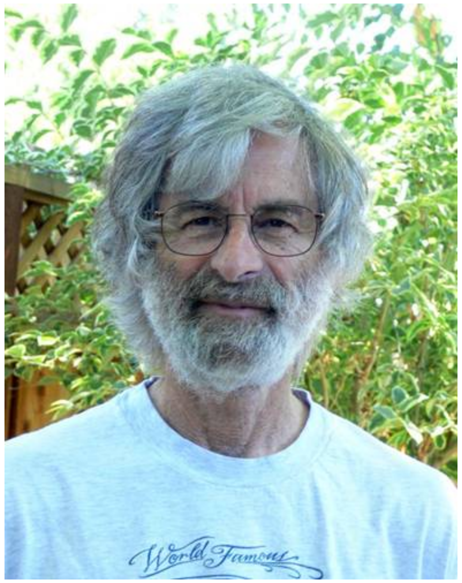
Transactions work by using locks to block each process before it is allow to complete so it can not commit if there's a rollback.

Isolation is also use locks to only allow one concurrent process to access the data at a time. Locks have some problems:

- Locks stop processes from moving forward.
- There may be no master to create and manage locks
- Locks can fail, leaving the system in deadlock

How can we achieve consensus in a system that doesn't use locks and where processes can fail (in non-malicious ways)?

Paxos



Leslie Lamport proposed a consensus algorithm called Paxos that can reach agreement among many non-malicious actors without the use of locks and in the presence of failure.

Lamport explained the algorithm by anthropomorphizing it in terms of a hypothetical part-time legislature that meets sporadically on the Greek island of Paxos.

Because this algorithm reaches consensus without a central controller, it is an example a *leaderless election algorithm*.

The details of the algorithm, and the proofs that it behaves as desired, can be complex, but the idea is remarkably simple.

Safety and Liveness

Distributed algorithms are evaluated in terms of their ability to achieve properties that can be classified as pertaining to *safety* and *liveness*.

Safety means nothing bad happens.

Examples of safety include:

- The balance of accounts is the same before and after a transfer
- The system is secure

Safety properties are associated with invariants.

Liveness means that good things eventually happen.

Examples of liveness include:

- The money is eventually transferred between accounts.
- A message is eventually delivered.

Liveness properties are associated with well-foundedness arguments.

Agents in Paxos

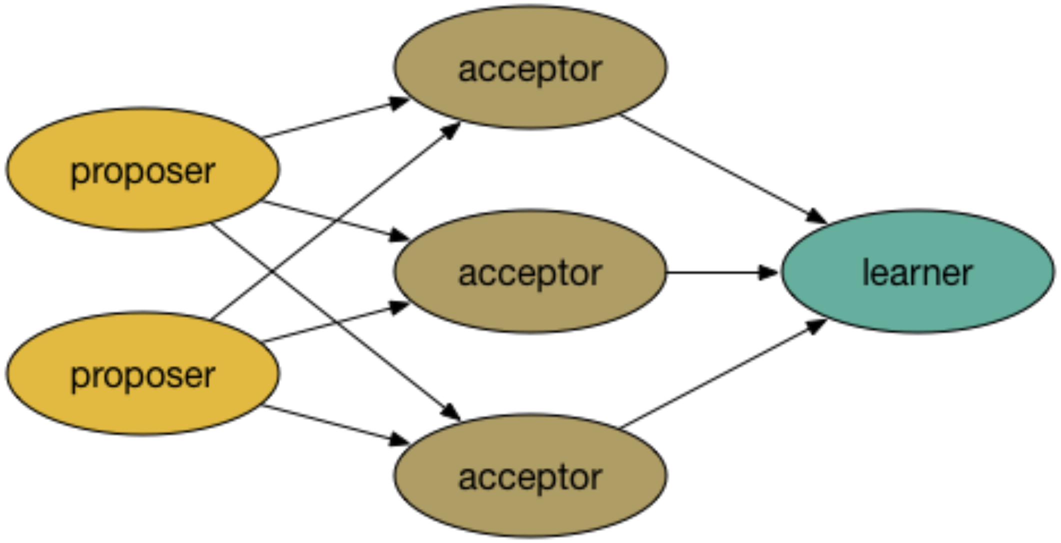
The goal of Paxos is for a set of agents to agree on a value without a centralized leader.

Agents operate at arbitrary speed, may fail by stopping, and can restart. When agents restart, they can remember their state when they stopped. Messages can be lost. There are three kinds of agents. Usually a process plays all three roles at the same time. We treat them separately for clarity.

Proposer -- send messages with a proposed value to a set of acceptors

Acceptor -- may accept any proposed value they receive. Acceptors can only accept one value.

Learner -- learn when a value has been accepted by a majority of acceptors and consensus is reached.



Safety and Liveness for Paxos

Safety Properties

Only a proposed value may be chosen.

Only one value is chosen.

Nodes never learn that a value has been chosen unless it actually has been.

Liveness Properties

Some proposed value is eventually chosen.

Once a value is chosen, a node can eventually learn the value.

Paxos Algorithm

The Paxos algorithm operates in two phases, a *prepare phase* where proposers propose values and an *accept phase* where acceptors accept values and notify the designated learners of their choice.

Prepare Phase:

a) A proposer selects a proposal number, n, and sends a request with that number and the proposed value to a majority of acceptors.

b) If an acceptor receives a *prepare request* with number greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals with a smaller number and with the value of highest-numbered proposal (if any) that it has accepted.

Accept Phase:

a) If the proposer receives a response to its prepare requests (with number n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

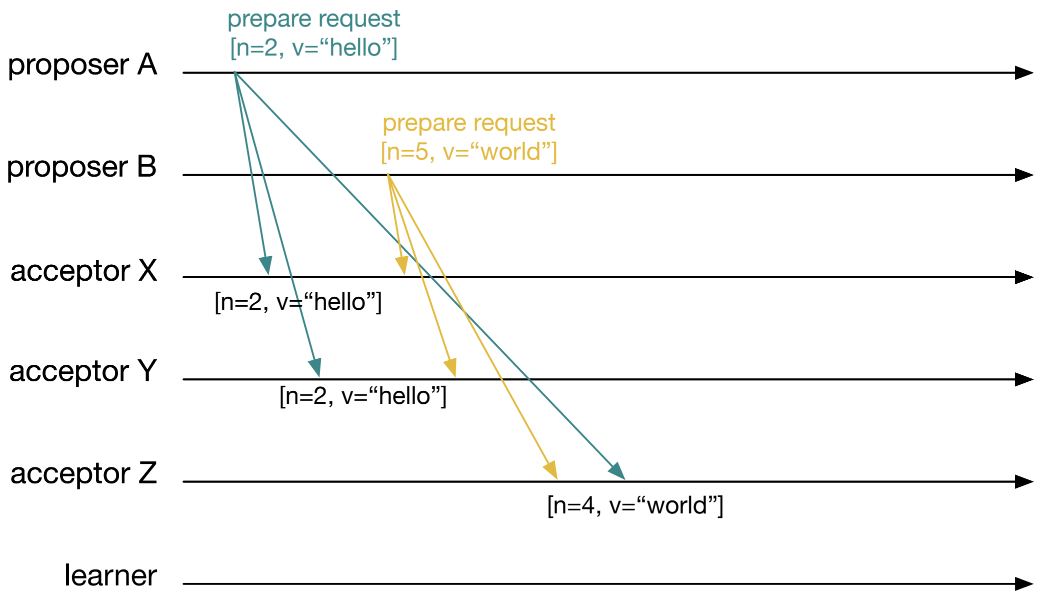
b) If an acceptor receives an *accept request* for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

Paxos Example: Prepare Phase Request

In this example, there are two proposers, A and B and three acceptors, X, Y, and Z.

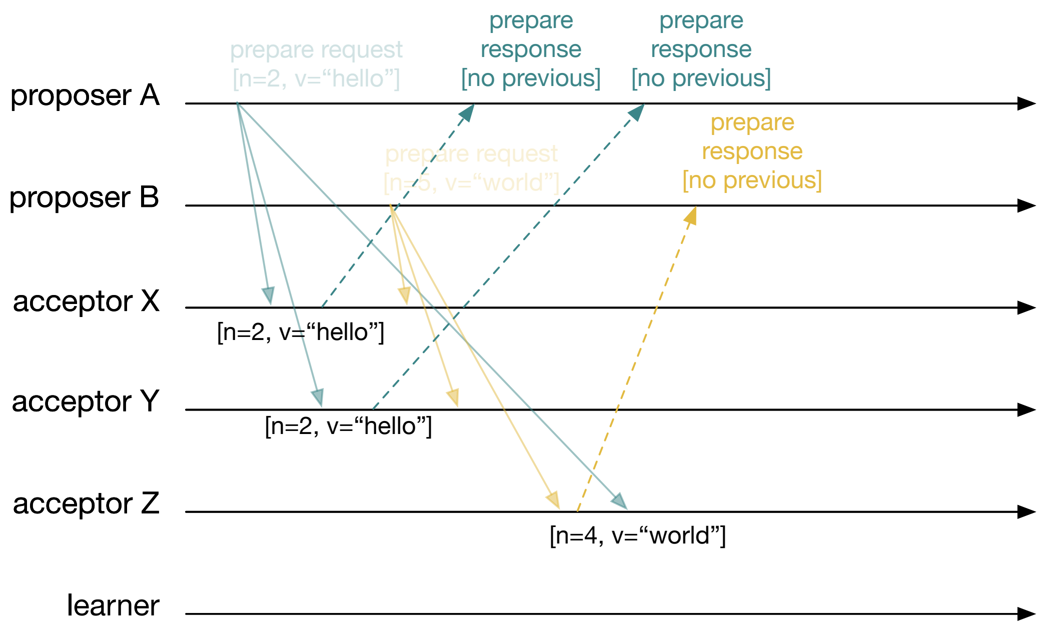
Each proposer sends a request to all of the acceptors they know about.

In this case, A's requests reach acceptors X and Y before B's. B's request beats A's to acceptor Z.



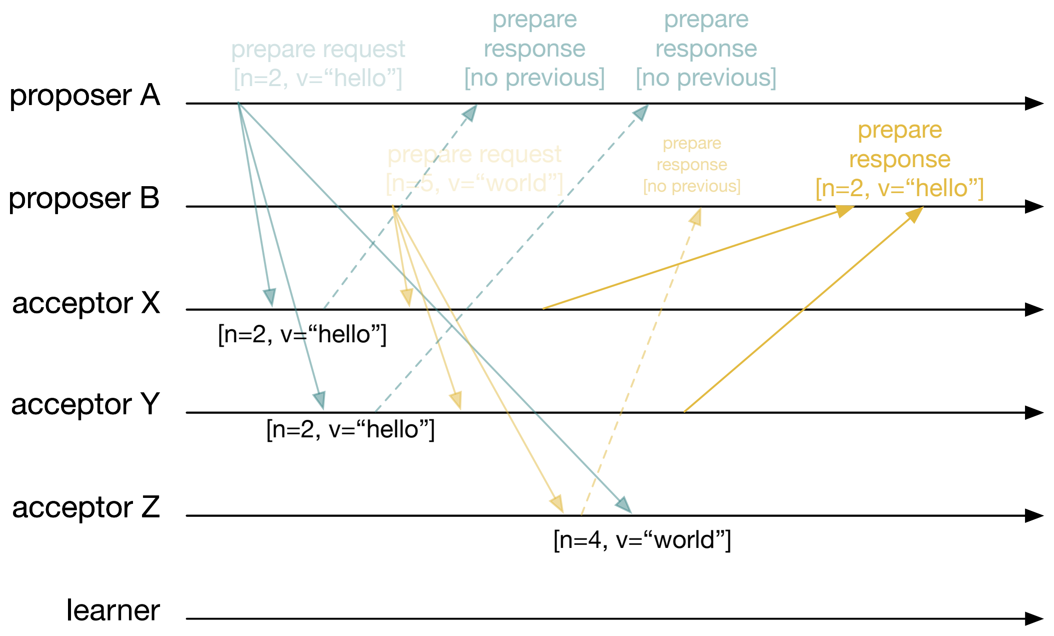
Paxos Example: Prepare Phase Response

In the response phase, acceptors X, Y, and Z all respond to the first prepare request they receive stating that they've had no previous request.



Paxos Example: Prepare Phase Response

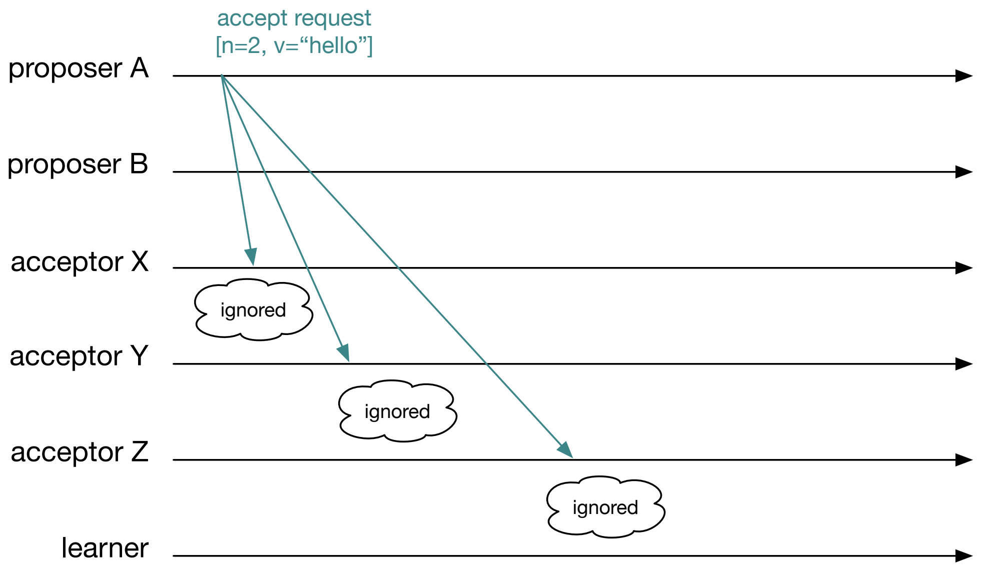
Later, after receiving B's proposal, X and Y both respond with a message saying they've already responded to a request with n=2 and v = "hello."


proposer A
proposer B
acceptor X
acceptor Y
acceptor Z
learner
prepare request [n=2, v="hello"]
prepare response [no previous]
prepare response [no previous]
prepare request [n=5, v="world"]
prepare response [no previous]
prepare response [n=2, v="hello"]
[n=2, v="hello"]
[n=2, v="hello"]
[n=4, v="world"]

Paxos Example: Accept Phase Request

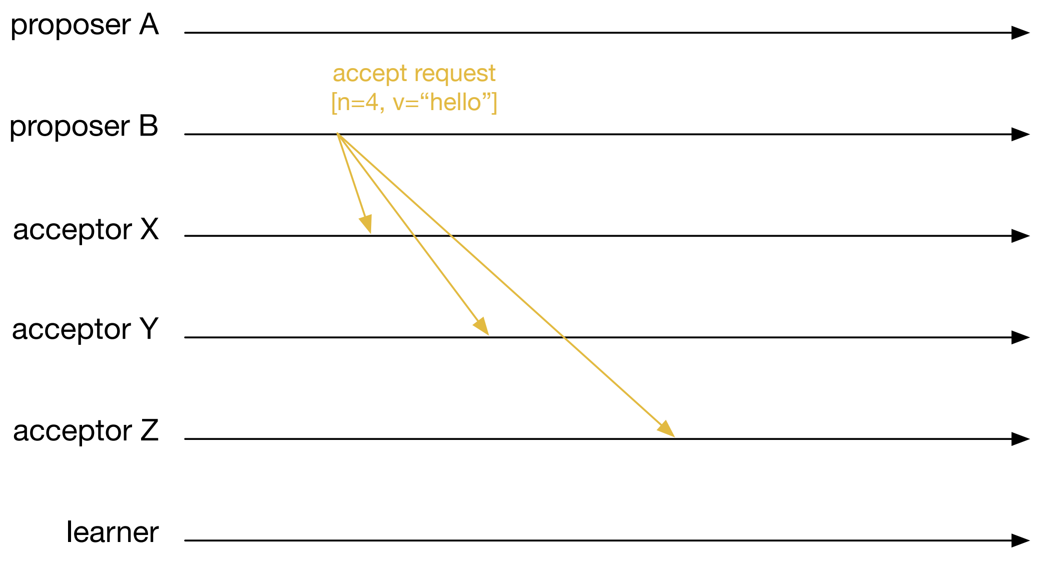
Since A received responses indicating that X and Y had received no previous proposals, it sends an accept request message to all the acceptors for its original number and value.

But since each acceptor has promised to not accept anything less than n=4 after receiving B's proposal, these accept requests are all ignored



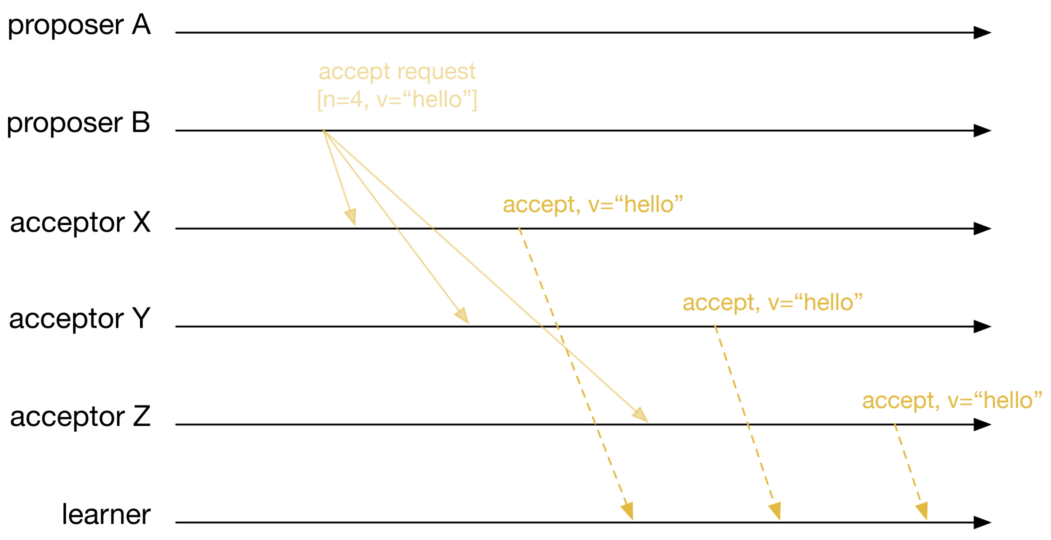
Paxos Example: Accept Phase Request

B, on the other hand, has received responses that indicate a majority of acceptors accept the value "hello," so it sends an accept request with it's original proposal number but the value, "hello," accepted by the majority.



Paxos Example: Accept Phase Response

Having received an accept request from B with a number equal to or greater than the largest proposal number they've seen, each acceptor tells the designated learner that it accepts the value "hello."



Notes on Paxos

Paxos isn't about making a decision about the right value, it's about gather consensus on a value.

Paxos assumes that the actors are honest although they may go away or fail to deliver messages.

Paxos is just one example of a consensus algorithm.

Questions About Paxos

Answer these questions by yourself or in a small group

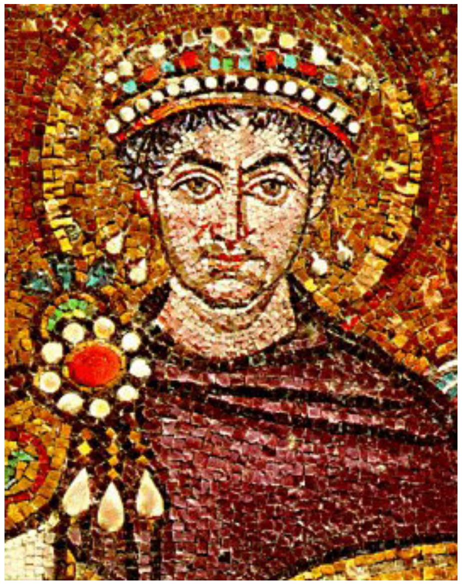
1. What happens if another proposer, C, proposes a higher-number prepare request (e.g. [n=6, v="kitten"]) after the acceptors have sent a prepare response to B's request?
2. Can you convince yourself or a knowledgeable friend that the example in the previous slides meets properties $P1^a$ and $P2^c$ in the Paxos Made Simple paper?
3. How would you ensure that each proposer never selects the same proposal number as another without communication overhead?
4. Why does Lamport suggest using a distinguished proposer and how is such a proposer selected?

Byzantine Failures and Bitcoin

Bitcoin uses a distributed ledger called the blockchain to achieve consensus in spite of the failure or even bad intentions of some actors.

Byzantine Failure

The Byzantine Empire was known for intrigue and political plotting. Thus, when Lamport was looking for a story to illustrate the problem or reaching consensus with participants who are acting in bad faith, he called it the "Byzantine Generals Problem" and failure where actors can report false values is known as *Byzantine failure*.



Bitcoin and Byzantine Faults

While there are several excellent examples of algorithms that exhibit byzantine fault tolerance, Bitcoin makes use of one that is particularly interesting.

Imagine you want to create a digital currency like Bitcoin that has no central point of control. It is distributed, decentralized, and heterarchical. One of the primary problems is creating a ledger that tells everyone how much Bitcoin everyone else has even though

- Some people may lie
- Nobody is available to arbitrate disputes

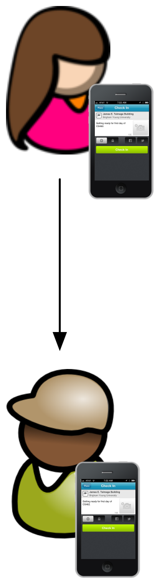
Bitcoin presents a classic consensus problem (i.e. the ledger) where some actors have impure motives.

Introducing YCoin

To understand how Bitcoin (and it’s underlying distributed ledger system called the *blockchain*) work, let’s develop a hypothetical cryptocurrency called YCoin.

Our development follows the arguments of *How the Bitcoin protocol actually works* by Michael Nielsen. We will approach the problem iteratively, solving just one or two problems at a time.

Problem 1: Sending Money



Suppose Alice wants to send one YCoin to Bob.

Alice can create a message that says “Alice sends on YCoin to Bob” and sign it with her private key.

As we learned in our lesson on Integrity, Confidentiality, and Non-Repudiation, this allows anyone with Alice’s public key to verify that

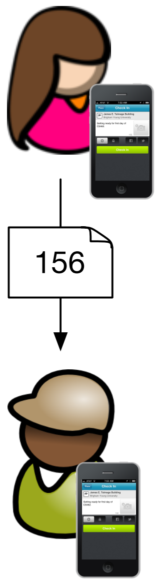
a) she (and only she) sent it and

b) it keeps Alice from claiming she didn’t.

These are two important properties.

But what if Bob gets two such messages? Did Alice send him two YCoins? Or was the message accidentally duplicated.

Problem 2: Uniquely Identifying Coins



Alice can send YCoin messages with a serial number.

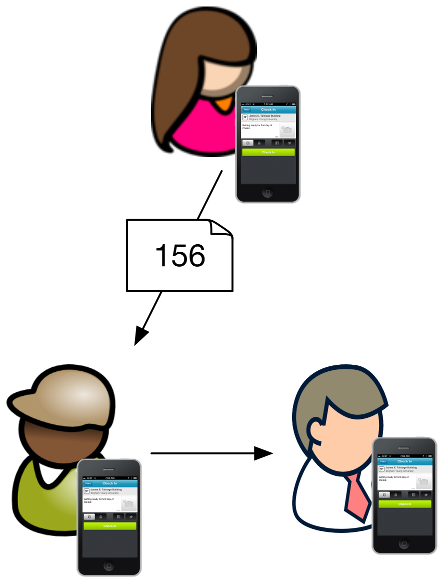
The problem of uniquely identifying coins is fairly simple to solve. Alice can ensure that each YCoin she sends has a unique serial number.

The typical way to do this is to create a *bank*. The bank issues serial numbers (i.e. transaction IDs) and keeps track of who owns which serial numbers. The bank is entrusted to ensure the ledger is up to date and accurate.

When Alice sends a YCoin with serial number 156 to Bob, he can contact the bank and ensure that the serial number is valid, Alice is the current holder of the YCoin with serial number 156, and that no one else is claiming it.

When he accepts it, the bank updates its ledger to show that Bob now owns YCoin 156.

Problem 3: Distributing the Bank



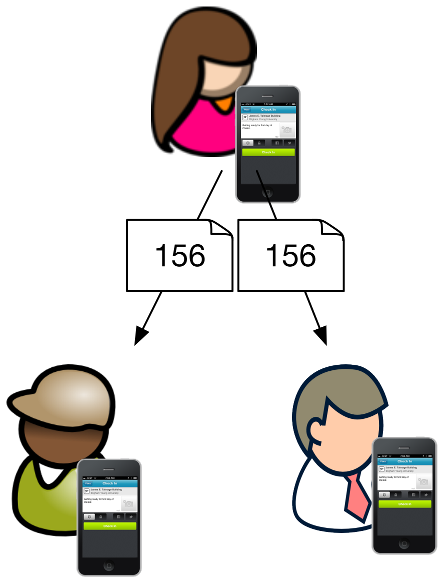
But we want a solution with no central bank, so we need to do something more ambitious.

Suppose that instead of checking with the bank when Alice sends him a YCoin, Bob checks with everyone else who has YCoins. Everyone is keeping track of who owns which YCoins in a shared, public ledger called the *blockchain*.

When Bob receives the message, he can check his copy of the blockchain to ensure Alice really owns YCoin 156. If she does, then he broadcasts Alice's message and his acceptance of the YCoin to the entire network.

Upon receiving Bob's message, the everyone in the network updates their copy of the ledger to remove YCoin 156 from Alice's balance and add it to Bob's.

Problem 4: Preventing Double Spending



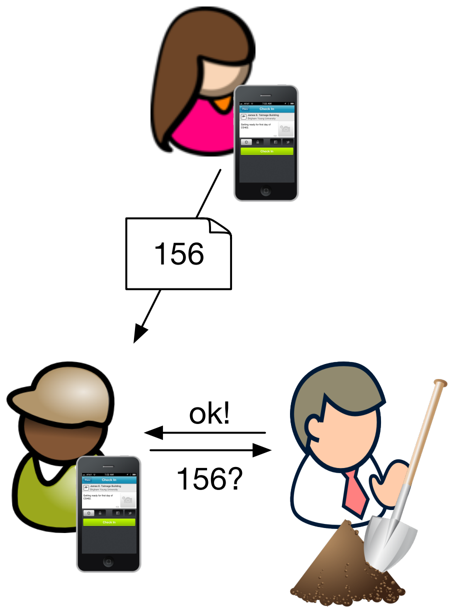
This all works great, unless Alice is dishonest (e.g. byzantine).

What if Alice tries to spend the same YCoin with Bob and Charlie? You might think this is difficulty since everyone would notice, but what if Alice takes advantage of a network partition or network latency? She could convince both Bob and Charlie that they own YCoin 156 and collect whatever she's exchanging it for before the other notices there's a problem.

The answer, not surprisingly, is a two-phase algorithm. Rather than merely accepting Alice's transfer at face value, both Bob and Charlie broadcast the transaction to the entire network for verification. Other members of the network check to see if Alice has the coin and hasn't spent it.

In the double spending case, other network members would see both Bob and Charlie's verification requests and signal a problem.

Problem 5: Stopping Network Hijacking



Alice can still double spend using the previous solution if she can take over the network by adding her own nodes that propagate her lies.

Alice can succeed because validation is based on a voting system. Anyone with a node on the YCoin network can vote. Virtual machines, and proxies make adding nodes cheap—cheaper than the money Alice can make with fraudulent YCoin transactions.

We can make cheating more expensive by requiring validation to only happen after the validator has expended a certain amount of effort—called *proof-of-work*—to show they aren't just a small virtual machine or network proxy.

Because Alice has to do real work to validate her fraudulent transaction, she can only do so if she contains more than half of the computing power of the entire YCoin network. Provided there are many participants, this is impractical because it is too expensive.

Problem 5 Continued: The Nature of Work

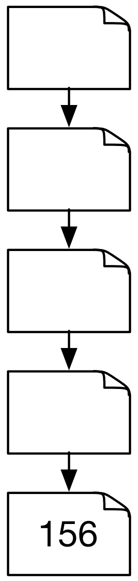
Before a node on the network can validate a transaction, we require that they solve a cryptographic puzzle.

The puzzle involves finding a value called a number called a *nonce* such that when the nonce is appended to the YCoin message and hashed, the first *n* digital of the result are all zeroes.

This sounds arbitrary but it has the property of allowing the amount of work to be adjusted according to the needs of the network at any given time by varying n. Larger values of n will require more computation to find a suitable hash.

Recall that hashing is irreversible. Consequently, the only way to search for a hash with a particular property is to try every possible computation.

Problem 6: Ordering Transactions



The YCoin network should agree on the ordering of transactions or we don't know, at any given time, who owns what, exactly.

Rather than merely validating individual transactions, the network can group them into blocks and as each block is validated, add it to a chain so that we have a precise order or blocks.

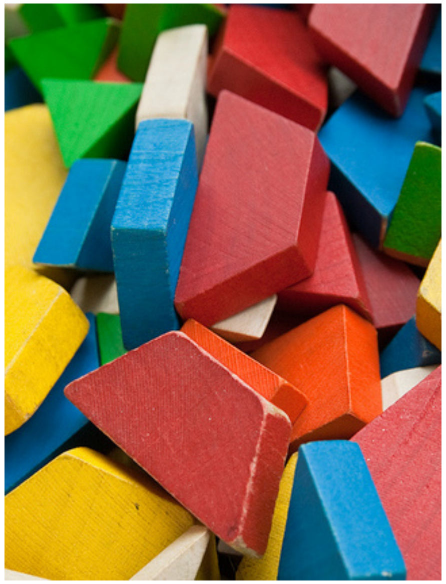
Under the right circumstances, the chain can be extended by two blocks at the same time, creating a fork and causing the problem we're trying to avoid.

The network can deal with forks by keeping track of them, but agreeing to only extend the longest chain. Because of the probabilistic nature of the proof-of-work problem, once fork will get longer and become the winning chain, ensuring there's only one, linear chain that completely orders all transactions.

Why The Blockchain Matters

The actual blockchain is a sophisticated computational system, not merely a simple public ledger. Consequently, it can be used to record all sorts of information that would generally require a centralized ledger.

The world is full of directories, registries, and ledgers—mappings from keys to values. We have traditionally relied on some central authority (whoever owns the ledger) to ensure its consistency and availability. Blockchain is a global-scale, practical ledger system that demonstrates consistency and availability without a central authority or owner. This is why blockchain matters.



Conclusion

Summary & Review

Credits

Summary and Review

Consensus is a fundamental concept in distributed computing. Most problems rely on cooperating actors agreeing on values.

In this lesson we've covered a broad range of consensus options from ensuring a specific set of actions succeed in a relatively local system by means of transactions.

We're also seen that transactions aren't usually practical for more decentralized systems and other strategies for handling failure are needed. These strategies include writing off the failure, trying the failed action, or undoing a successful action.

Both of these are aimed at preserving system cnosistency in the face of failure.

More generally, distributed systems often need to reach consensus to solve a problem. This is also a consistency issue, but among possibly decentralized processes.

We learned about Paxos as an example of algorithms that achieve consensus in the face of failure and without a centralized control point. Paxos succeeds event when processes fail.

Finally, we looked at byzantine failures—where failure is caused by malicious actors—and explored the blockchain as a system for ensuring consensus regardless of how failure might occur.

Credits

Photos and Diagrams:

- A Cruise Ship off the coast of Belize (https://commons.wikimedia.org/wiki/File:A_Cruise_Ship_off_the_coast_of_Belize.jpg), CC BY-SA 3.0
- Ken Arnold (https://commons.wikimedia.org/wiki/File:Ken_Arnold_-_25_december_2006.jpg), public domain
- Starbucks Baristas (https://commons.wikimedia.org/wiki/File:Baristas_first_starbucks.jpg), public domain
- Anna was her Starbucks name (https://www.flickr.com/photos/allaboutgeorge/14580567785), CC BY-ND 2.0
- A magnetic keyed padlock (https://en.wikipedia.org/wiki/Magnetic_keyed_lock), CC BY-SA 3.0
- Leslie Lamport courtesy of lamport.org
- Georgios Anemogiannis (https://commons.wikimedia.org/wiki/File:Georgios_Anemogiannis.jpg), CC BY-SA 2.0

Credits (continued)

Photos and Diagrams:

- Justinian (https://commons.wikimedia.org/wiki/File:Justinian.jpg), public domain
- Bitcoin (https://commons.wikimedia.org/wiki/File:Bitcoin.png), public domain
- Colorful Wooden Blocks (https://www.flickr.com/photos/stevendepolo/5644838033), CC BY 2.0

All other photos and diagrams are either commonly available logos or property of the author.

The Travel Agent example is from Enterprise Java Beans 3.0 by Richard Monson-Haefel