Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

Introduction

At the core of Radix lies its unique consensus protocol, Cerberus. Cerberus will have two distinct features that no other decentralized Distributed Ledger Technology (DLT) has yet to achieve: (1) practically infinite linear scalability; while preserving (2) cross-shard atomic composability.

Cerberus achieves this by starting with a unique data structure that pre-shards the ledger into 2^{256} shards [1]. This is large enough to fit every thousandth atom in the observable universe into its own shard; or for another comparison, all possible combinations of Bitcoin addresses could fit in the Cerberus "shardspace" 79 billion billion billion times. Every change to the ledger, or "substate", is deterministically allocated its own shard based upon its hash.

When a transaction occurs, Cerberus' novel consensus design allows nodes to temporarily "braid" consensus across the shards of relevant substates together. Related substates can thus be composed into atomic transactions when needed, and unrelated substates can be processed completely in parallel. Each node is only required to serve a subset of shards - no global state or ledger is maintained by any one node.

Because of this, as nodes are added to the network, transaction throughput increases linearly without practical limit; transactions reach settlement finality in less than five seconds; transaction fees will always be tiny; nodes can always be run on simple hardware; and the ability to compose transactions atomically across the global ledger will never be sacrificed.

By the end of this infographic series you will understand what this means and how it works. A more in depth and technical description of Cerberus can be found in the Cerberus whitepaper here; and an independent validation of the safety and liveness of Cerberus by the University of California Davis can be found here.

Unique Features

Practically Infinite Linear Scalability
Allows Radix to always scale to meet the needs of network usage – just like the internet, there is no ceiling

Cross-Shard Atomic Composability
Allows transactions to be composed across shards - either settling all together, or not at all - across the global ledger

[1] Shards in Cerberus: 115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

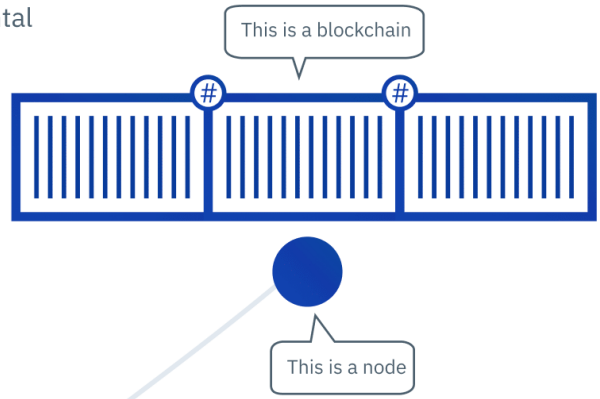
Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

Summary

Why Blockchains Can't Scale - Episode 3

We begin by outlining some core concepts, such as what nodes and blockchains are.

We then explore why blockchains can't scale, at a fundamental level.



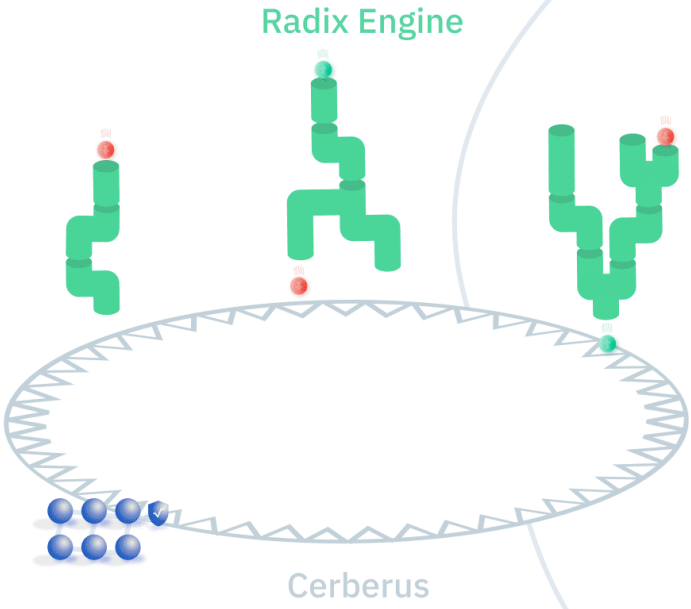
Every available scaling solution tried to date, such as:

- pushing more transactions down the same chain;
- "sharding" or "Layer 2" solutions;
- centralizing aspects of processing;

either reaches a scalability limit, or sacrifices, to varying extents: decentralization, security, or composability - the ability to settle transactions seamlessly across the ledger.

Blockchains are therefore not the technology that will allow Decentralized Finance ("DeFi") to scale to fulfill the future needs of billions of people.

A new paradigm is needed.



What is Radix - Episode 4

Radix is a "Layer 1" protocol. This means it's the foundational software run by nodes that together form the Radix network.

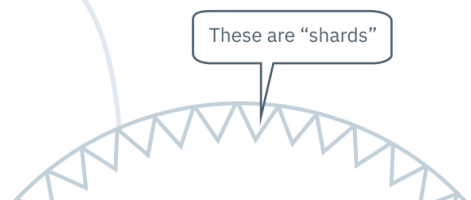
Radix is not a blockchain. Its design is radically different from all other decentralized networks. Radix is unique.

Radix is composed of two layers:

- Radix Engine - for applications;
- Cerberus - for consensus.

Anyone can program rules using the Radix Engine. Those rules govern how assets such as tokens, or decentralized applications, work.

This infographic series is focused on Cerberus, Radix's consensus layer.



The Shardspace and Validator Sets - Episode 5

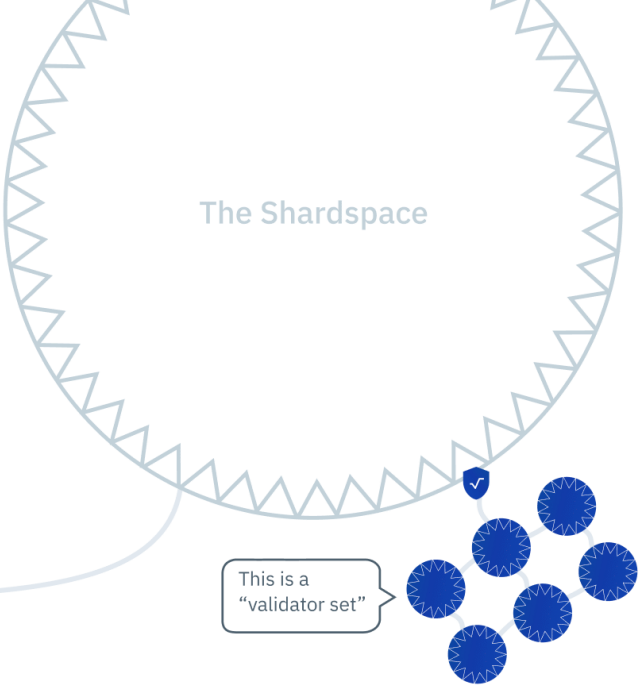
Radix's use of Cerberus as its consensus layer starts by presharding

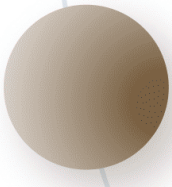
its ledger into 2^{256} shards.

Those shards are fixed, and will never change. All shards put together become the “Shardspace”.

Each individual shard is served by a collection of computers - nodes - called a “Validator Set”.

A validator set is responsible for verifying, voting on, and maintaining a record of transactions for shards they serve.





Substate - Episode 6

Every change to the ledger is recorded in a unit of data called a “substate”.

Substates are similar to Unspent Transaction Outputs (UTXOs) in Bitcoin, but substates can represent any kind of data, from tokens to smart contracts.

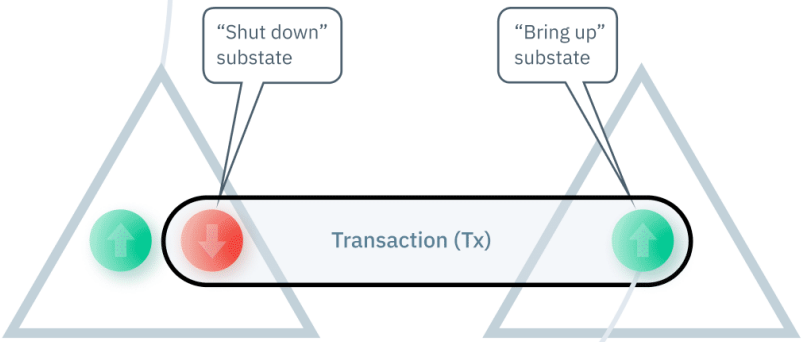
Substate and Transactions - Episode 7

Substate is created in transactions.

Transactions chain together two types of substate: “bring up” and “shut down” substates, across shards.

Bring up substates record changes to the ledger. Shut down substates record that bring up substates can’t be used again - preventing “double spends”.

Each bring up substate has its own shard.



256 bit #

Substate ID: 144,826,489,488,222,149,009,878

Shard Allocation - Episode 8

By hashing the data of a bring up substate, you get a unique ID.

That unique ID maps perfectly to a shard in the shardspace.

Therefore every bring up substate has its own deterministically allocated shard.

Transaction (Tx)

Shard 1: *Shut down Alice's Token A substate*

Shard 2: *Bring up Bob's Token A substate [Index: 0]*

Shard 3: *Shut down Bob's Token B substate*

Shard 4: *Bring up Carol's Token B substate [Index: 1]*

Shard 5: *Shut down Carol's Token C substate*

Shard 6: *Bring up Alice's Token C substate [Index: 2]*

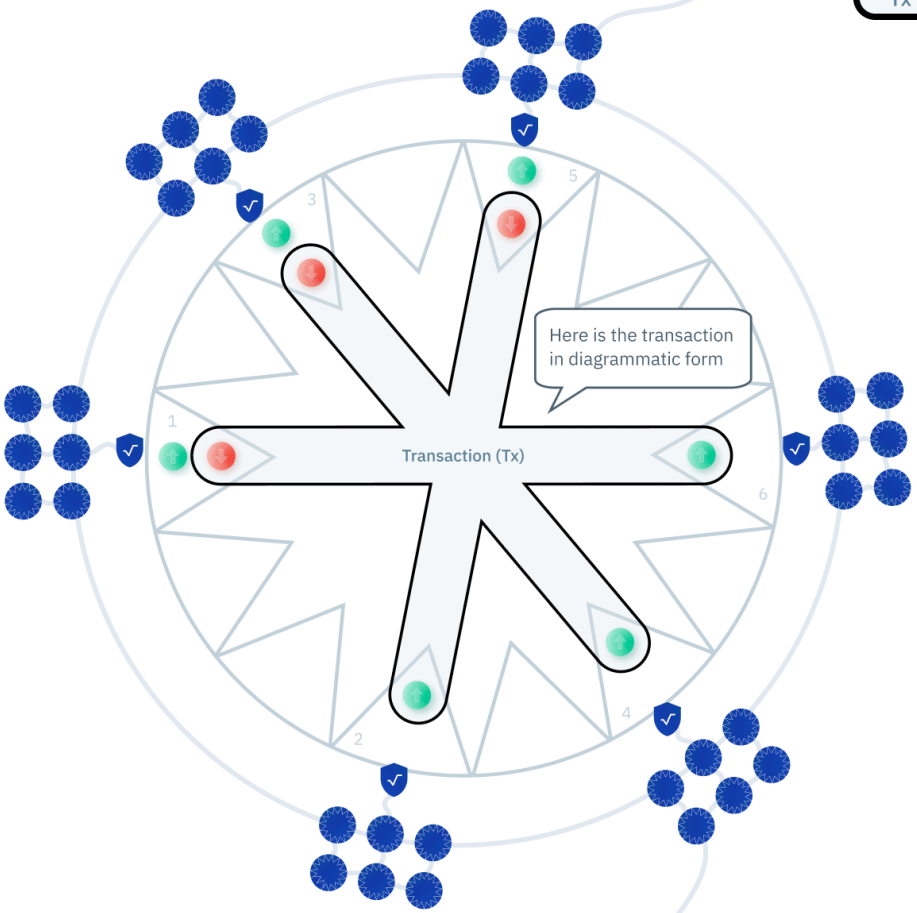
Signed: *Alice, Bob, Carol*

Tx ID: ...979

Transactions - Episode 9

Transactions are instructions sent to nodes to tell them what to write to the ledger.

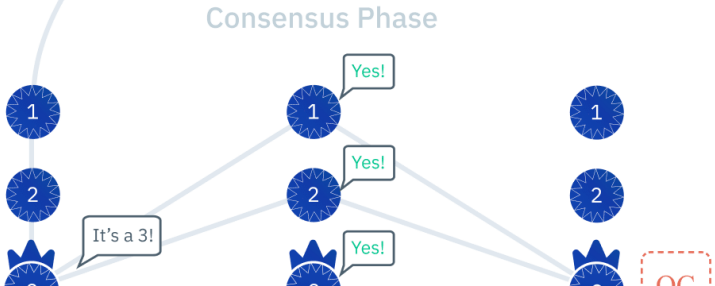
They contain all the details necessary to write substates to individual shards.



But nodes need a way to agree with one another on whether the transactions that are sent to them are valid, before they can commit the transaction to the ledger.

Nakamoto vs BFT-style Consensus - Episode 10

Nodes use "consensus protocols" to communicate and agree with one another. Bitcoin and Ethereum use a "probabilistic" class of consensus protocol named after



Satoshi Nakamoto.

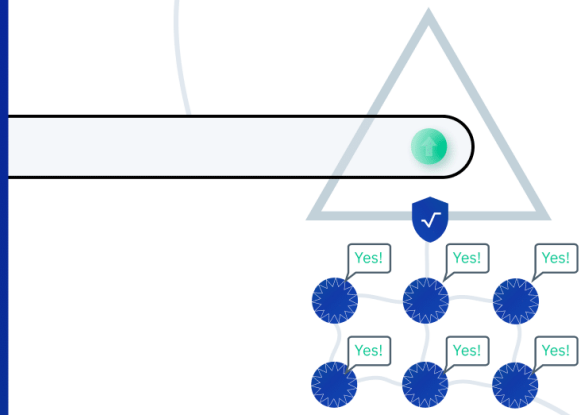
Instead of Nakamoto consensus, Cerberus uses a class of "Byzantine Fault Tolerant" style consensus that can be considered more "deterministic".

BFT-style consensus involves communication between nodes in "phases", an example of which is depicted to the right.

A phase is composed of three steps:

- a leader broadcasts a message
- nodes vote on whether they agree with the message
- if enough vote weight is in agreement, the leader creates a Quorum Certificate, QC, which provides cryptographic proof of the outcome of the vote.

Local Cerberus



Consensus - Local Cerberus - Episode 11

Cerberus' method of reaching consensus is in two tightly integrated parts - the first being local Cerberus, which is how nodes come to consensus with each other, for a single shard, within a validator set.

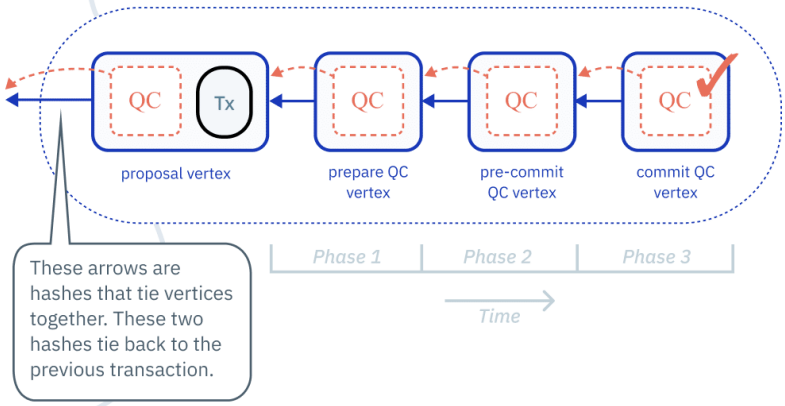
Consensus in local Cerberus forms a “3-chain”.

A 3-chain is proof that the nodes of a single shard's validator set approved their part of a transaction over three phase of voting.

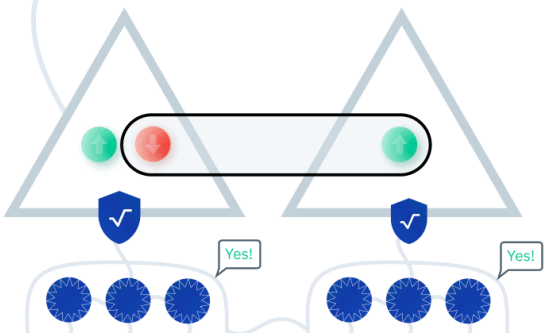
Proof of the vote for each phase is contained in the QC, and the QC is packaged into a data structure called a “vertex”.

There are three “chains” because each vertex contains hashes that chain it back to the previous vertex.

Local Cerberus 3-chain



Emergent Cerberus



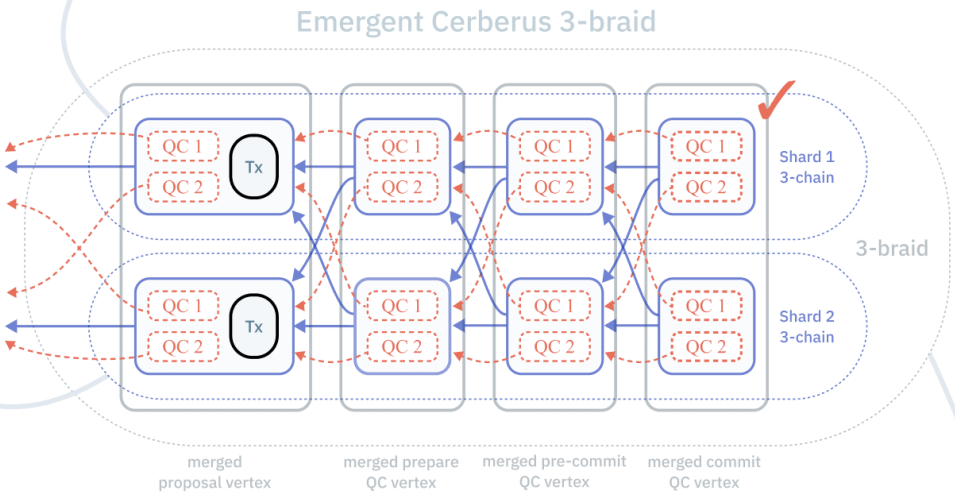
Consensus - Emergent Cerberus - Episode 12

The second part of Cerberus is Emergent Cerberus, which is how nodes come to consensus across shards, between validator sets.

Emergent Cerberus “braids” consensus across shards.

Multiple 3-chains braided together form a

Multiple 3-chains braided together form a "3-braid".



In emergent Cerberus, the QC for each shard is shared with the validator set for every other shard. We can see above that the QC for both shards 1 and 2 are contained within every vertex for every shard.

This gives each validator set visibility of the outcome of voting for every other shard, and allows validator sets to commit the transaction either all together, or not at all.

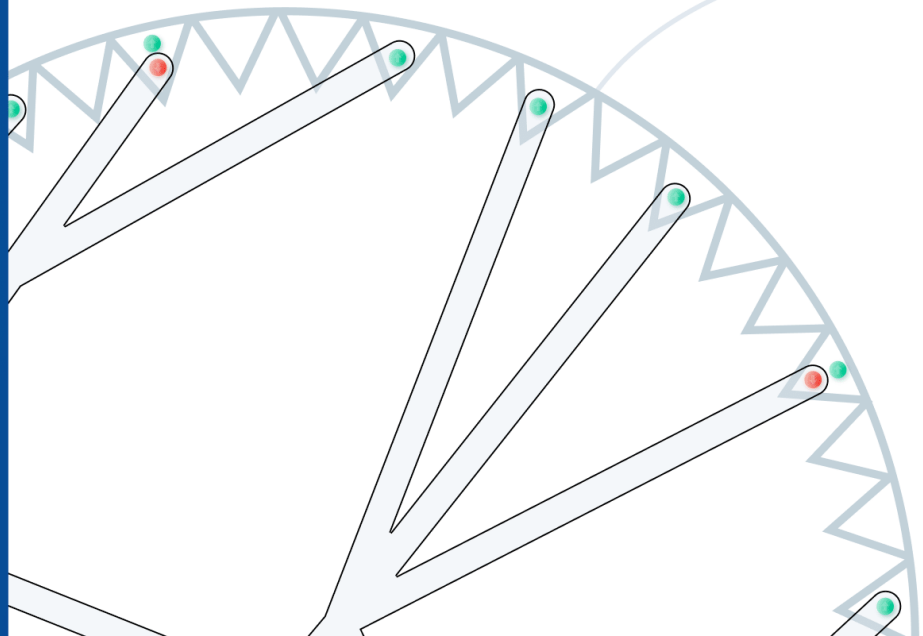
This is the key innovation of Cerberus and uniquely provides Cerberus the ability to settle transactions atomically across shards, also known as “cross-shard atomic composability”.

But to take full advantage of this braided consensus for scalability, we need our application layer to provide "partial ordering" in transactions.

Partial Ordering - Episode 13

Nearly all blockchain ledgers enforce global ordering, which is when all transactions are ordered on the same global timeline.

This limits the transaction throughput of those networks as unrelated transactions have to wait for other unrelated transactions before they can be settled.

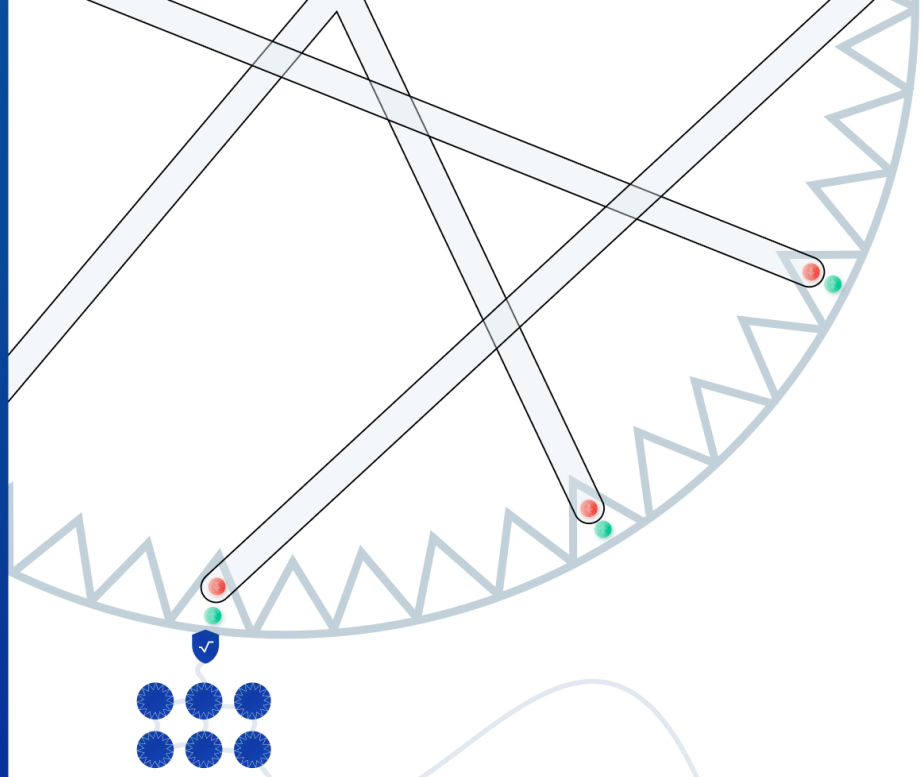


One of Cerberus' innovations is enabling a ledger that can be "partially ordered”.

Where substates are related, they are composed and ordered within the same atomic transaction.

Where substates are unrelated, they can be processed by transactions completely in parallel, across the vastness of the shardspace.

Those unrelated transactions don’t ever have to be “aware” of the others’ existence, as there is no global timeline that they have to be



placed on. As all transactions happen in parallel, no transaction ever has to wait for another transaction as it just gets processed immediately.

This allows Cerberus to scale transaction throughput “linearly”, as every additional node added to the network adds more computation and bandwidth, as the same amount of shardspace is now served by more nodes.

Because the shardspace is practically infinite in size, the linear scalability offered by Cerberus will never hit a ceiling. That’s how Radix achieves “practically infinite linear scalability”.

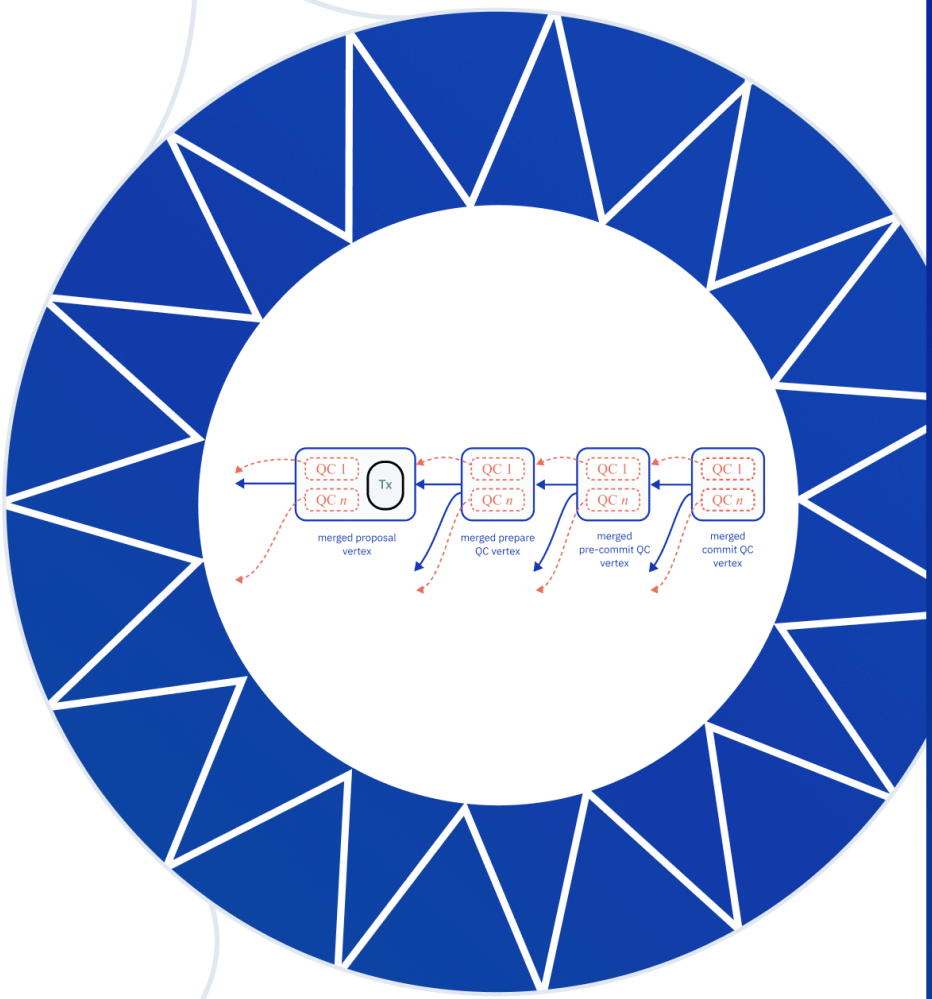
Maintaining a Record of Transactions - Episode 14

One of nodes’ key responsibilities is to maintain a record of transactions.

Radix nodes store a copy of the whole transaction, which includes all the substates across all relevant shards.

They also store proof that a ~67% majority (“2f+1”) of vote weight approved the transaction across every shard, for every phase of consensus through the QCs in each vertex.

Each vertex is chained to the previous vertex via hashes - cryptographically tying every phase of consensus together.



Sybil Resistance Through Proof of Stake - Episode 15

Radix uses Delegated Proof of Stake as a

means of weighting votes to protect the network against a type of attack called a Sybil attack.

Conclusion - Episode 16

We finish by revisiting why blockchains can't scale, and reflect on how Radix's unique design makes it sui generis: a new paradigm in decentralized DLT.

As the only DLT that achieves practically infinite linear scalability while preserving cross-shard atomic composability, Radix is the only decentralized network that will allow DeFi to scale seamlessly to fulfill the future needs of billions of people.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

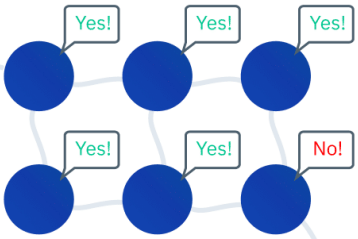
Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

Why Blockchains Can't Scale

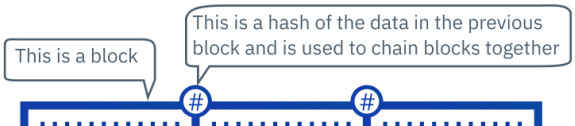
This is a node.

Nodes have two jobs:
- to verify transactions
- to maintain a record of transactions.

Nodes do this by talking to one another and voting. If enough of them agree, then they are in "consensus".



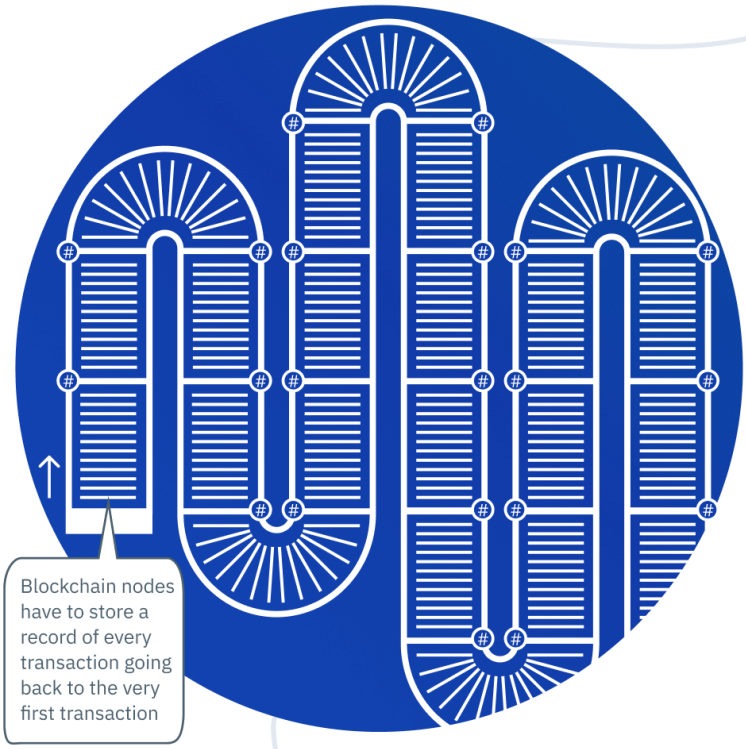
DLTs need a majority of nodes to agree with one another in order to achieve consensus, with votes weighted in a variety of ways.

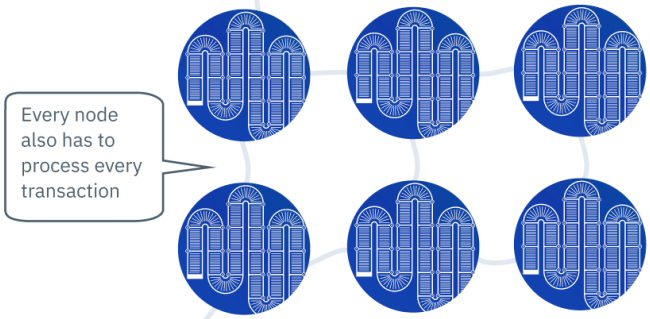


Blockchains, the most common form of DLT data structure, use consensus to agree on a sequence of “blocks” that contain transactions. Blocks are connected together into a “chain” with cryptographic proofs called hashes.

A hash is a derivative of certain data. For example, if your data is 10, a very silly hashing algorithm could be to divide by 5, so the hash is 2. If the 10 changes at all, e.g. to 11, the hash won’t match, as 11 ÷ 5 ≠ 2. So if a hash of some data matches, you know your data hasn’t been tampered with.

If you were to zoom into a Bitcoin transaction, you would see that it would be composed of what are called ‘unspent transaction outputs’, or UTXOs. A UTXO is a part of a previous transaction that hasn’t been spent yet.



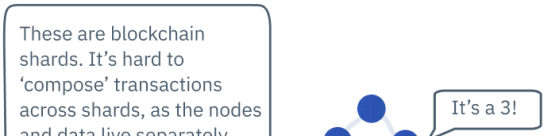


Why simple blockchains don’t scale

When a node verifies whether a transaction is valid, it checks to see that the UTXOs proposed in the transaction have not been spent in the past, in order to prevent a double spend. To do this, nodes require a record of every transaction going all the way back to the very first transaction – the entire blockchain – otherwise they wouldn’t be able to perform a complete search. Over time, this increases the burden on nodes, as they have to store more and more data.

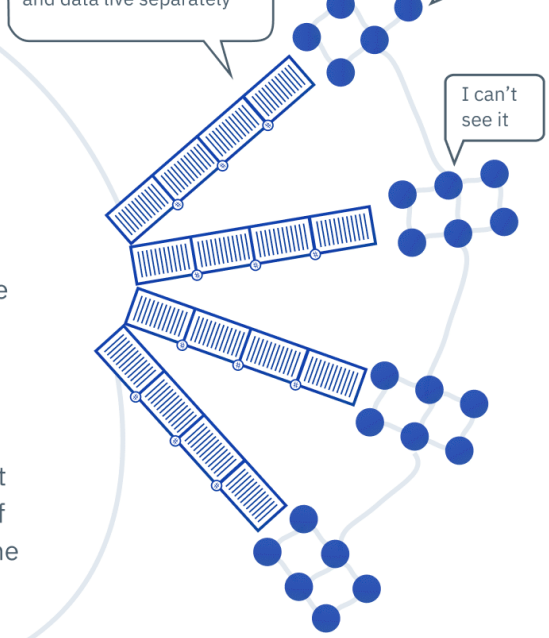
Additionally, every node also has to process every transaction, as there is only a single global ledger state that every node agrees on. For example, if you want to execute an Ethereum smart contract, to come to consensus on that contract execution, every single Ethereum node in the world is asked to process that transaction. This means a blockchain designed in this way cannot process more transactions than a single node can.

This is already causing major bottlenecks. Imagine the entire world’s traffic running through a single lane! The Bitcoin blockchain for example can only process around 7 transactions per second (TPS) in total. Even the fastest single lane (or single shard) blockchains today, can “only” process ~65,000 TPS. For a single DLT to fulfill all the world’s transaction needs for the entire 21st century: serving as the foundation for a trustless internet and decentralized global financial system by processing every single human and machine transaction alike; 65,000 TPS is too low by several orders of magnitude.

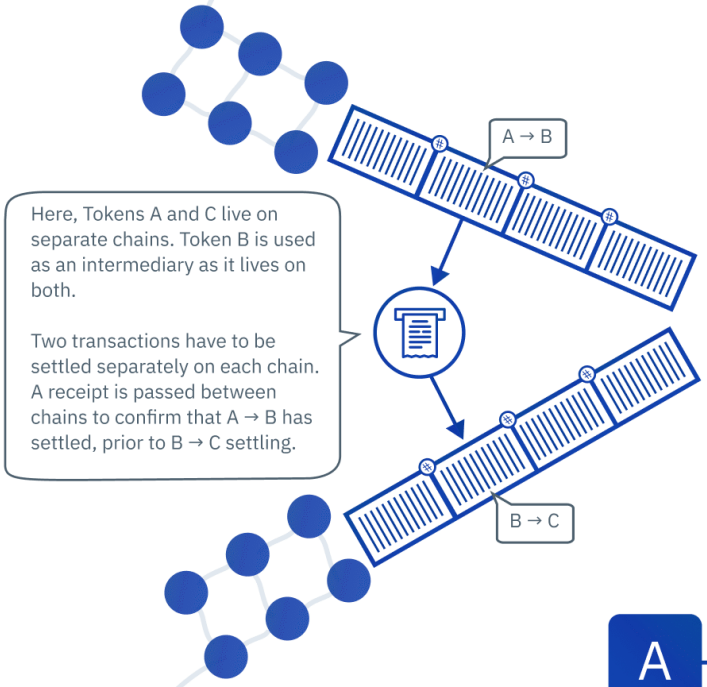


Why sharded blockchains don't scale

In order to solve this, some networks have implemented "sharding". This is where the ledger is broken into multiple parts running in parallel, such as on separate blockchains. Each of these parts has its own set of nodes. This is kind of like adding new roads that run alongside one another, so more traffic, or transactions, can be processed. This also allows nodes to split up the data so each node doesn't have to store as much.



However this introduces a new problem. If a transaction needs to be executed across multiple shards, how do the nodes arrive at consensus across those shards, if they can't see the other nodes' data? They would have to send a lot of data back and forth, possibly in multiple steps, defeating the point of sharding by slowing things down again.



Sharding blockchains breaks atomic composability

To expand on this problem further, sharding the ledger creates the issue of "non-composability".

On a single shard, "composing" a transaction across multiple tokens or decentralized applications (dApps) is easy, as all the nodes have all the data, and can easily come to consensus. Across multiple shards however, nodes lose the ability to come to consensus as they lose sight of each other's data. What was a single transaction on one shard has to be broken into multiple separate transactions across multiple shards.

Without atomic composability, this transaction failed half way through, "trapping" the user with Token B.

This causes major problems - particularly so in decentralized finance ("DeFi"). In DeFi, dApps and assets like tokens have the most utility when they can be connected together – composed – in single transactions. dApps or tokens living on separate shards makes DeFi slow and cumbersome, and opportunities for efficient capital allocation may be lost.

For example, if you want to sell Token A and buy Token C, and the only way to do so is to buy Token B as an intermediary step, you wouldn't want the A → B transaction to settle, unless you knew that the B → C transaction would also settle, otherwise you might get stuck owning B and you can no longer get C.

Ideally, you would want to be able to compose a single transaction that settles once across all shards, e.g. A → B → C all in one go, guaranteeing you get C. This is what "cross-shard atomic composability" enables.

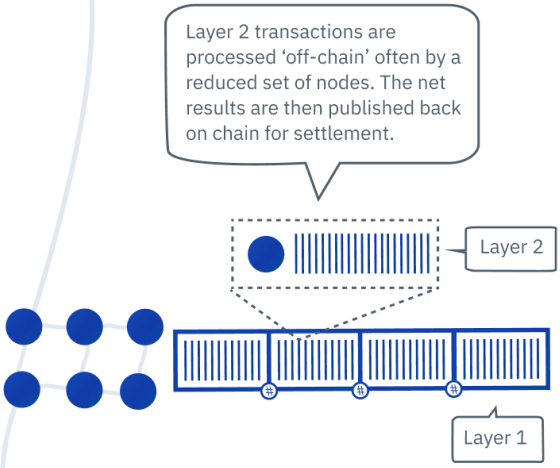
Some networks have implemented what's called "state locking" to solve this, which locks states across shards prior to settlement. This would allow you to back out of the A → B transaction if the B → C transaction were to fail. However, this coordination takes time, and allows high frequency arbitrageurs to easily front-run any trades. It's therefore only a partial solution.

How ‘Layer 2’ solutions solve scalability, but sacrifice decentralization and atomic composability

Another solution is to execute transactions “off-chain”, such as on the Bitcoin Lightning Network; or through “rollups” or “sidechains”, which are now being used on Ethereum. These are collectively referred to as Layer 2 (L2) scaling solutions because transaction processing is taken off the primary network, the Layer 1 (L1).

Assets, such as tokens, can then be ported between the L1 and L2 networks; either individually, or in batches, which is more efficient.

But L2 solutions create silos of transaction processing which are often more centralized and less secure than the primary network. Additionally, with many different types of L2 solution being implemented, composing a transaction across many different L2s quickly makes things very complicated. L2 solutions thus reduce interoperability and the ability to settle transactions atomically. They thus also break “atomic composability”.



Tradeoffs in blockchain system design

As we can see, there appear to be tradeoffs when designing blockchain systems. Those tradeoffs are commonly referred to as the “Blockchain Trilemma”, which posits that there is a three-way compromise between:

- scalability;
- decentralization; and
- security.

Improving one tends to come at the expense of the others.

However, it’s a bit more nuanced than that:

- decentralization isn’t useful at all without security, and so security is an uncompromisable feature; and

- scalability on its own also isn’t very useful.

  Imagine “spinning up” 1m Bitcoin blockchains. You’d have a set of systems that could process 7m transactions per second, but they wouldn’t be very useful, as those 1m blockchains couldn’t interoperate particularly well and you’d have silos of transaction processing and liquidity.

  The opposite of such fracturing is for a system to exhibit “atomic composability”, which is the ability to structure a single transaction across any section of the ledger and for all parts of the transaction to settle atomically - either all together or not at all.

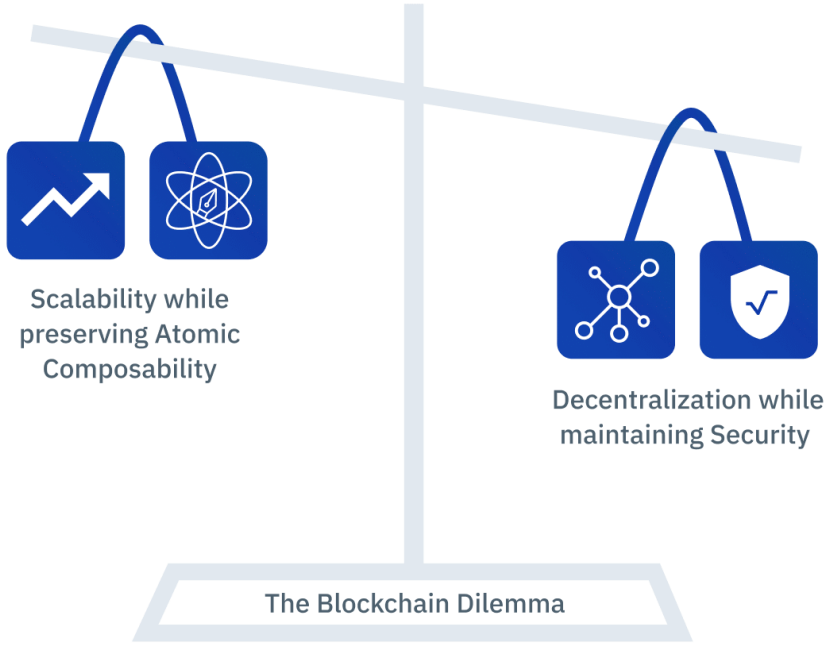
  For DeFi to seriously differentiate itself and outcompete the rest of

finance, it will require unlimited atomic composability.

Scalability is therefore only useful when it comes with composability.

The Blockchain Trilemma could therefore be reimagined as the "Blockchain Dilemma", with a more refined two-way compromise between:

- scalability while preserving atomic composability; and
- decentralization while maintaining security.



Through the lens of the Blockchain Dilemma, we can see that sharding or L2 solutions sacrifice atomic composability, making them unsuitable for DeFi.

Current Solutions	Achieves	Sacrifices
Central control of nodes	Increase in scalability	Decentralization
Sharding or L2	Increase in scalability	Atomic Composability

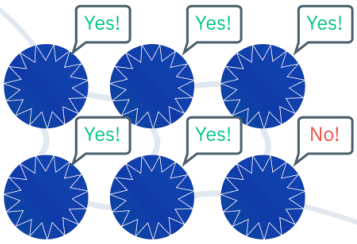
Given the Blockchain Dilemma, a radically different paradigm is needed if DeFi is to ever scale to fulfill the future needs of billions of people.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

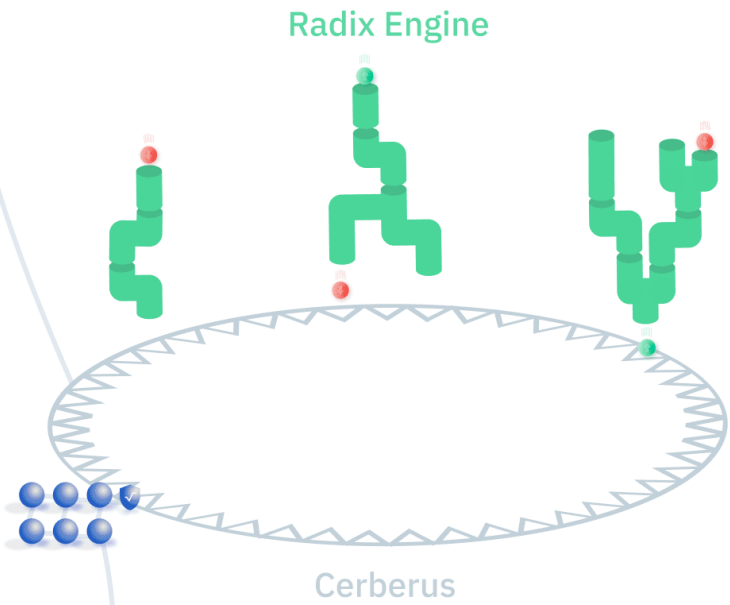
v1.0 · June 2021

What is Radix?

In essence, Radix is software for nodes.



When many nodes run the Radix software, they form a network. That network is capable of agreeing or disagreeing - coming to consensus - on data. Put together, that data creates what we call a "ledger".



The Radix node software is composed of two main parts:

- Radix Engine - which is Radix's "application layer". It includes the set of rules that nodes use to **determine what can and can't be written to the ledger**. Developers can use Radix Engine to write their own rules that combined together, can form decentralized applications.
- Cerberus - which is Radix's "consensus layer". It includes the set of rules that nodes use to **agree with one another** about what should be committed to the ledger. They do this by applying the rules defined in the Radix Engine.

The two are inextricably linked. The Radix Engine or Cerberus don't do very much on their own - Radix needs both parts working together.

Radix Engine allows developers to create "components" of smart contract logic that can be snapped together just like lego bricks.

This allows developers to quickly, easily, and safely build decentralized applications: “dApps”.

Because of this, Radix can be considered to be a “Layer 1” protocol, as it’s the base layer that other applications and protocols are built on top of.

But this infographic series isn’t meant to describe how Radix Engine works - that’s for another time.

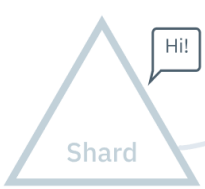
For now, we cover Cerberus. Specifically, the fully sharded version of Cerberus that is scheduled to launch as part of the Radix Xi’an release. For more information on the Radix roadmap, please visit https://www.radixdlt.com/

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi’an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

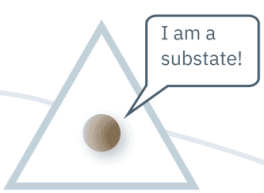
Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

The Shardspace and Validator Sets



This is a shard in Cerberus.

Shards in Cerberus work very differently to the blockchain shards we covered



In Cerberus, shards are a container for “substates”.

Substates are units of data that record changes

before, although the basic goal is the same - to break up data and allow parallel processing.

to the ledger.

Each substate has its own shard. We cover substates in detail in the next episode.

Every shard has an address. It starts with 1, although in reality it's a bit more complicated.

Then 2, and so on....

All the way to shard 2^{256}.

2^{256}

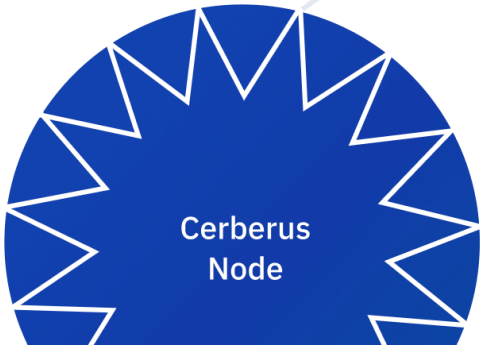
115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936

Combined, all the shards arranged together become the "shardspace".

We can find any shard in the shardspace by its address, as the address is always fixed.

There is no "start" or "end" to the shardspace – only an ordering that loops like a clock.

Every shard has two neighbours - one above, and one below. The neighbours of shard 1 are actually 2 and 2^{256}!

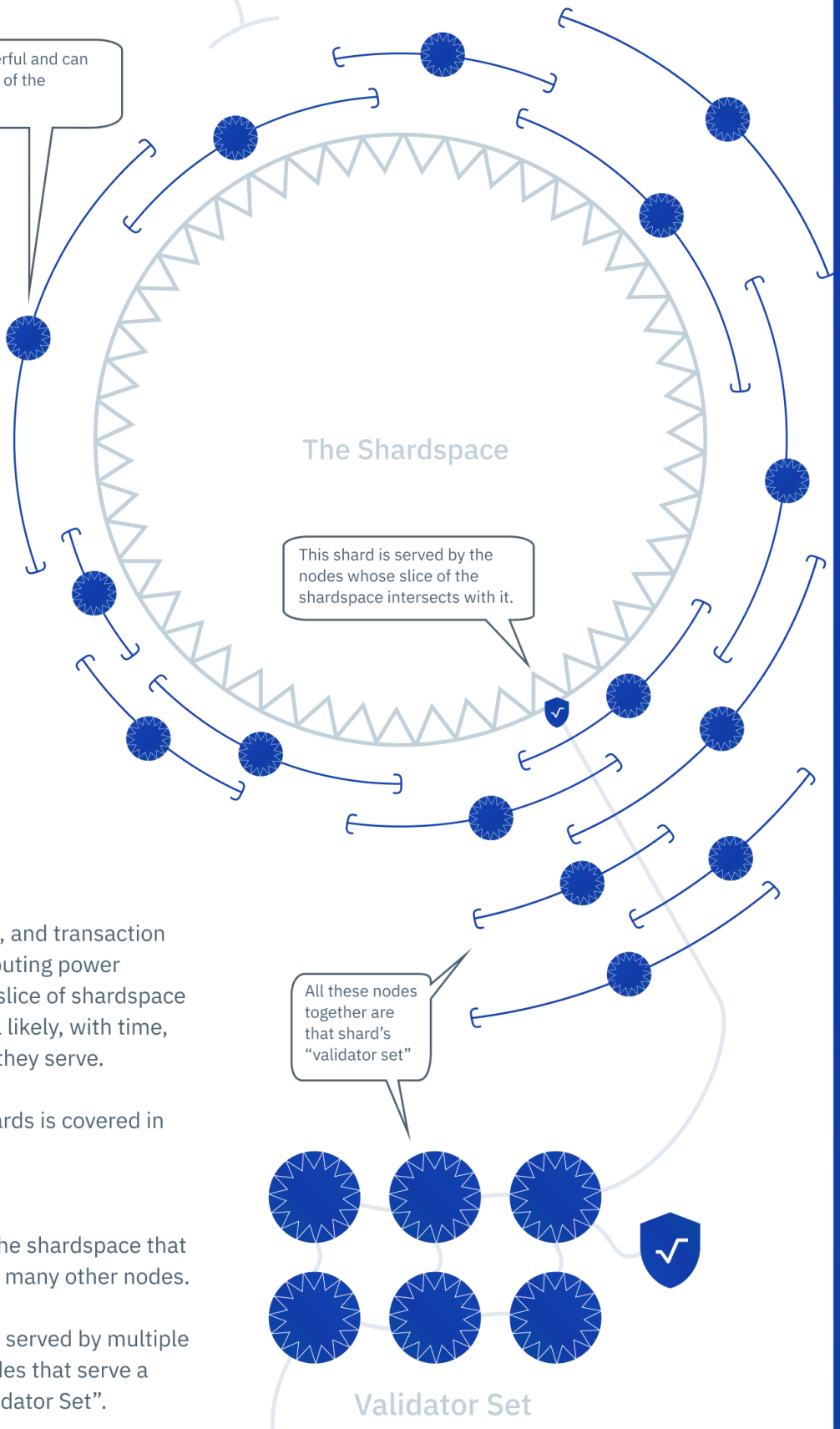


This is a node in Cerberus. All Cerberus nodes contain instructions on how to use the shardspace.

Nodes in Cerberus have the same job as any other blockchain node - to verify, vote on, and maintain a record of transactions.

However, instead of a blockchain, nodes in Cerberus are responsible for serving a slice of the shardspace. They

don t serve the entire shardspace as that would be too large for any one computer to handle, and we'd be back in the same position as with an unsharded blockchain - with every node verifying and maintaining a record of every global transaction!



Nodes serve a "slice" of the shardspace
Starting from an individual shard, the slice each node serves extends outwards in both directions. The width of that slice depends on the node's hardware capabilities. More powerful nodes can serve more shards.

By serving a shard, a node verifies, votes on, and keeps a record of the transactions that take place across that shard.

Between them, all the nodes across the whole of Cerberus serve the entire shardspace.

As the Radix network grows, and transaction volume increases, the computing power required to serve the same slice of shardspace will increase, and nodes will likely, with time, reduce the width of the arc they serve.

How data is allocated to shards is covered in episode 8.

Validator Sets
As we can see, the slice of the shardspace that a node serves overlaps with many other nodes.

Any given shard is therefore served by multiple nodes. The collection of nodes that serve a single shard is called a "Validator Set".

Additionally, each node will be a member of multiple different validator sets - one for each shard that that node serves.

The membership of any given validator set shuffles periodically as nodes join and leave the network, or when the slice of shardspace

So now we know what the shardspace looks like; we know that nodes each serve a slice of

that nodes serve changes.

The period of time that determines when this shuffling takes place is called an “epoch”.

the shardspace depending on their hardware capabilities; and we know that shards are served by a collection of nodes called a validator set.

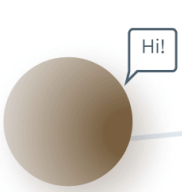
Let’s now explore what data looks like in Cerberus - which means understanding what a substate is, and how substates and the shardspace interrelate.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi’an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

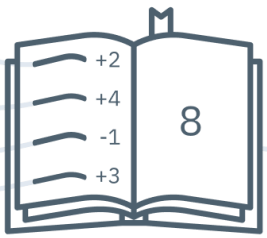
Substate



This is a substate.

A substate is a unit of data that represents a change to the state of the ledger. It does not represent the state itself.

Kind of like entries in a set of accounts, you have to add up all the entries to get your balance.



So in this example there are four substates. Adding them together gives you the state, which in this case is 8.

The state of the entire Radix ledger will be composed of millions upon millions of substates scattered across millions upon millions of shards.

Once a substate is created, it is there forever and cannot be changed - it’s immutable. To alter the effect of a substate on the ledger, you would have to create a new substate.

Additionally, a substate doesn’t have to just represent a number. It can represent *anything*.

From something as simple as the average temperature on a certain day.

To the ownership of a single token in a single address.

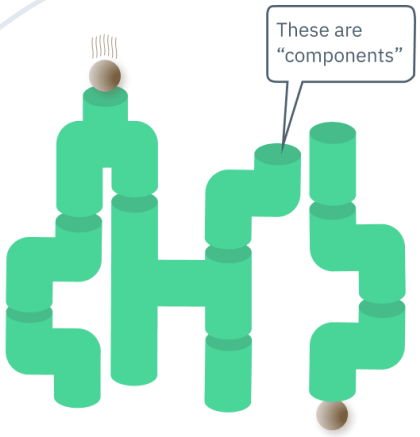
Or multiple tokens in a single address.

Or be as complex as the deployment of self-executing computer code - smart contract “components” - on-ledger.

Or thousands of different tokens across thousands of addresses.

A substate can therefore be as simple or as complex as needed - from a single token, to an entire dApp *and* all of its users’ tokens, anything can be be contained in a single substate, as a substate is just data.

However, Cerberus can’t tell the difference between a substate that represents the temperature, or a substate that represents a token - Cerberus only helps nodes come to agreement about which substates should be created and become part of the ledger.



Radix Engine

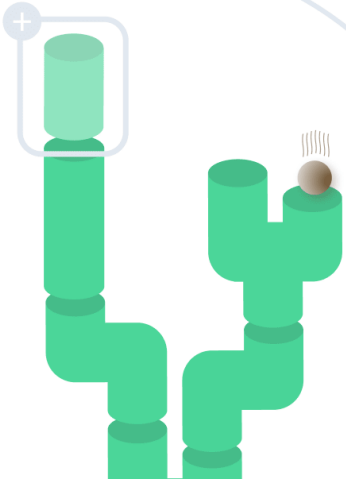
The part of Radix that helps nodes make sense of what a substate represents is Radix’s application layer, the Radix Engine.

The Radix Engine allows developers to create “components”, which is code (similar to a smart contract) that defines rules that tells nodes what to do with certain substates. These components can be snapped together in any way shape or form to create decentralized applications “dApps”.

So for example, the rules that govern a substate that represents the temperature - for example in a decentralized agricultural derivatives application - will be different from the rules that govern the substate that represents a token.

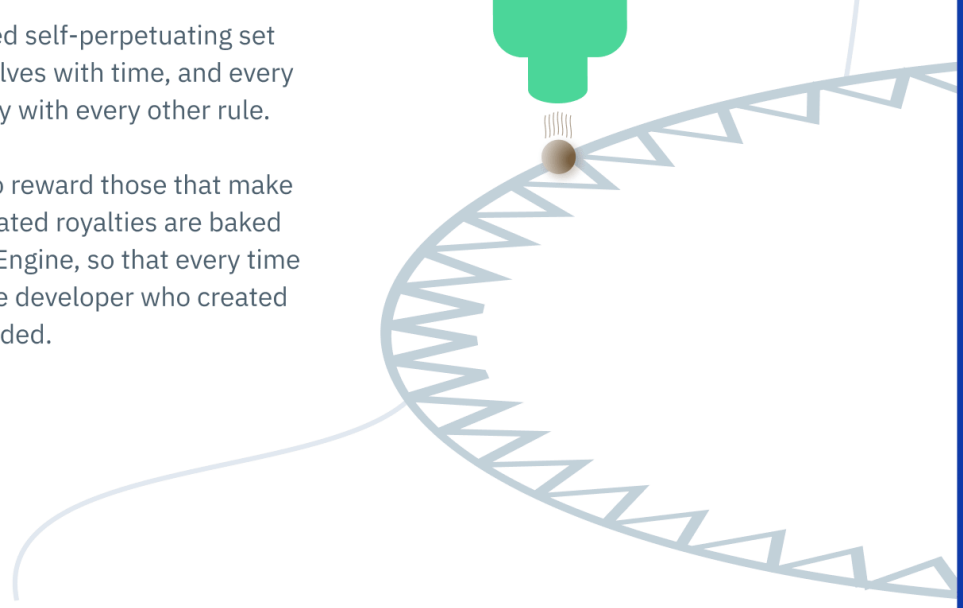
Because Radix is decentralized and permissionless anyone can build their own component. Because components are open-source, anyone can iterate on anyone else’s.

A process akin to natural selection will result in the best components having the most liquidity and use. With time, countless layers of components will be built, each extending and innovating on what came before, and they will all be atomically composable.



Radix is thus a decentralized self-perpetuating set of rules that grows and evolves with time, and every rule can interact seamlessly with every other rule.

A cornerstone of Radix is to reward those that make it better. That's why automated royalties are baked into the heart of the Radix Engine, so that every time a component gets used, the developer who created that component gets rewarded.



But now back to Cerberus - how is a substate created?

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

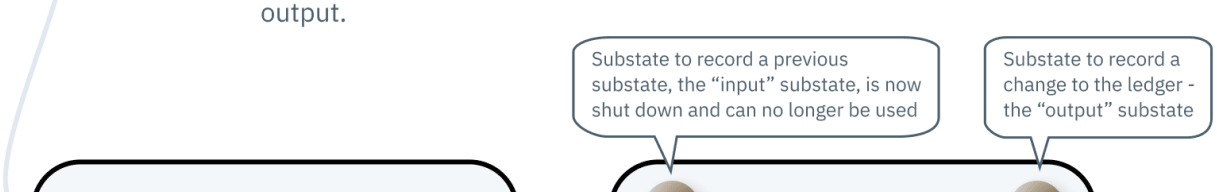
Substate and Transactions

So if you think back to how Bitcoin works, transactions are formed out of Unspent Transaction Outputs (UTXOs).

The Bitcoin transaction to the left consists of three UTXOs, where each one is "spending" the output from a previous Bitcoin transaction. Once spent, a UTXO cannot be used again in another transaction. You can trace any fraction of a Bitcoin through a sequence of UTXOs all the way back to when it was first "mined".

Radix takes a similar approach, but instead of just simple token transfers such as in Bitcoin, substate can represent anything - such as smart contracts.

To make a change to the ledger, a transaction must take one or more substates as input, shut down those substate(s), and then bring up one or more substate(s) as output.



This is a transaction in Radix.

A transaction is a unit of data that nodes keep a record of.

Transactions contain substates. There's a minimum of two types of substate that must be created for a transaction to work. On the left, a substate to record that the substate from a previous transaction, the "input" substate, has been shut down. On the right, an "output" substate to bring up a change to the ledger.

If the transaction is complex, it could involve many different substates of either type.

As these two types of substate do different things, and have their own unique properties, they have their own names, these are: "shut down" and "bring up".

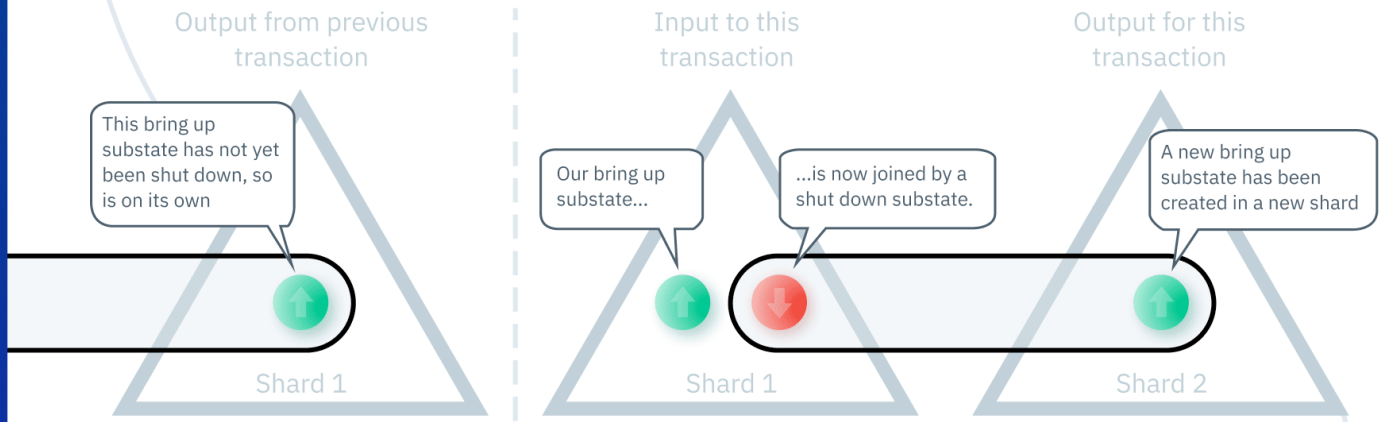
Shut down substates record that an input substate can no longer be used in another transaction, preventing "double spends".

Bring up substates are where the change to the ledger is recorded, e.g. the transfer of token ownership from one wallet to another. This is the output of the transaction.

So at the beginning of the shardspace and validator sets episode, we mentioned that every substate has its own shard. However, to be more precise, it is every bring up substate that has its own shard; and when that bring up substate is shut down, it is joined by a corresponding shut down substate in that same shard. A shard will thus over its lifecycle:

- begin empty
- contain a bring up substate
- contain a bring up substate and a corresponding shut down substate.

To illustrate, we walk through an example transaction below.



This shows Shard 1, with a bring up substate within it.

This is the output substate from a previous transaction. However, this will be the input substate for our example transaction to the

To make a change to the ledger, the transaction does two things simultaneously, or not at all:

Record that the input substate in Shard 1 has been shut down. A shut down substate is created in the same shard to record this.

Record the output for this transaction in a new bring up substate in a new shard - Shard 2.

The new bring up substate is

example transaction to the right.

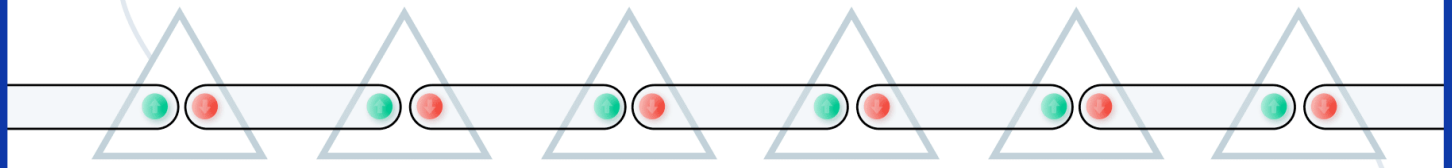
As this bring up substate has not yet been shut down, it has the shard to itself.

The transaction links the shut down substate in Shard 1 with the bring up substate in Shard 2. This is so that anyone looking at the output can see the related input that made it possible.

The substate in Shard 1 can never be used again in another transaction, as there is now a shut down substate in that shard.

The new bring up substate is linked back to Shard 1 by the transaction, immutably tying the two shards together – for that transaction only. Anyone who looks at either shard can thus follow the transaction in either direction.

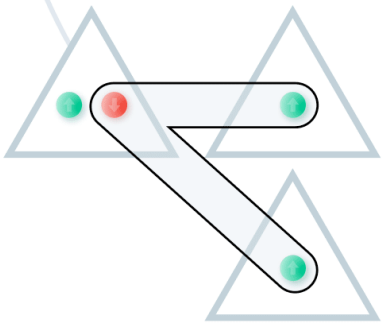
At some point in the future, this bring up substate will itself be shut down, and the cycle continues...



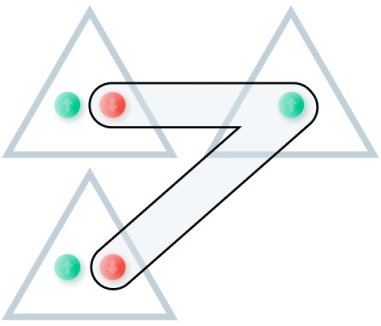
...forming a chain of bring up and shut down substates across many different shards, across many different transactions through time. These substates are connected cryptographically as inputs and outputs – similar to a blockchain, but on a much more individual scale rather than a single global ledger chain of blocks. You can follow any substate on the ledger all the way back to the genesis of the Radix ledger.

Furthermore, the example transactions above only concern shutting down one substate in one shard, and bringing up one new substate in a new shard.

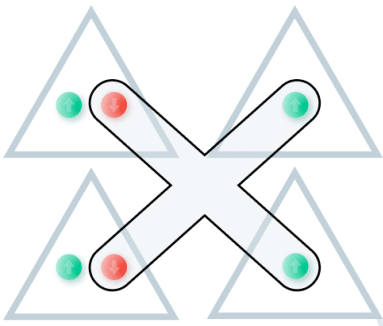
Transactions however can be much more complex. To illustrate...



A transaction could take a single bring up substate as input, shutting it down to create two or more bring up substates as output.



A transaction could take two or more bring up substates as input, shutting both of them down to create a single bring up substate as output.



Or the transaction could be any combination of any input or output substates, limited only by computing power and network capacity.

As a final thought, it's worth emphasizing that all transactions are cross-shard - it's not possible to have a transaction within a single shard.

The transaction thus serves as a wrapper, binding all the substates inside a single operation, so that they must be accepted either all together, or not at all. This is what “atomic” means.

So how are substates allocated shards?

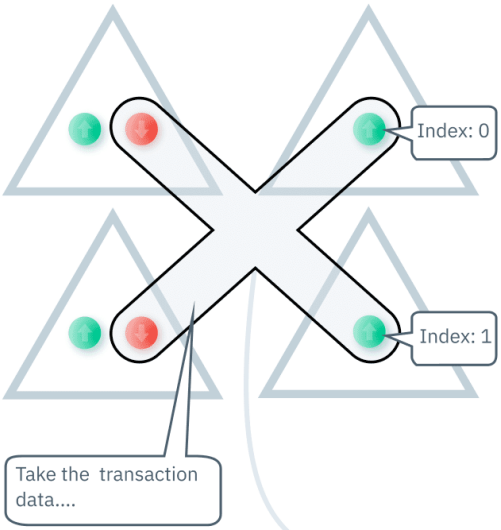
The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi’an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

Shard Allocation

So we’ve previously mentioned that each substate has its own unique shard in the shardspace. So how does this work? How does a substate get allocated its own unique shard?

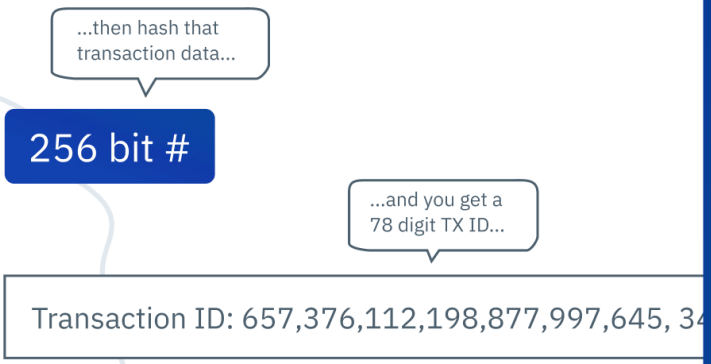


The transaction to the left is composed of two shut down substates and two bring up substates. The transaction will also include data such as a header and footer, and cryptographic signatures authorizing the transaction, e.g. from the people who “own” the tokens that the substate represents. We cover transactions in more depth in the next episode.

Each of the bring up substates within the transaction has an index position. The first bring up substate is Index: 0, the second - Index: 1, and so forth.

Each of these bring up substates will need their own address in the shardspace. Shut down substates don't need a new shard address as they live with a bring up substate from a previous transaction.

So to start with, we must first generate a unique

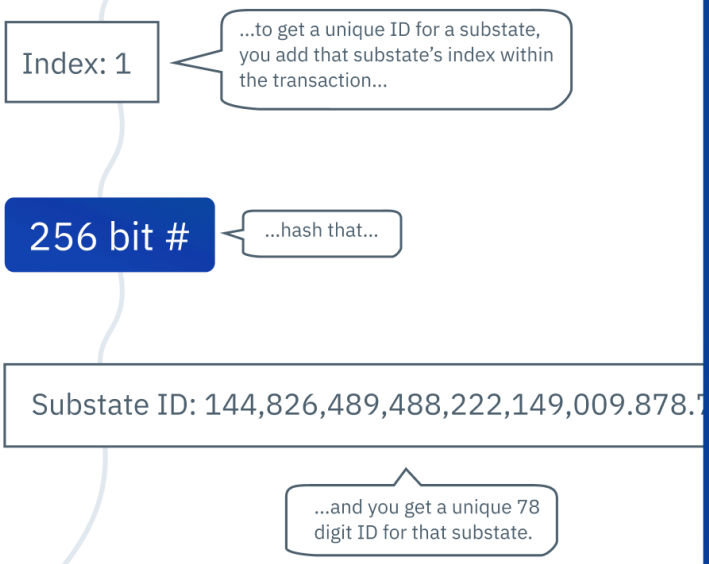


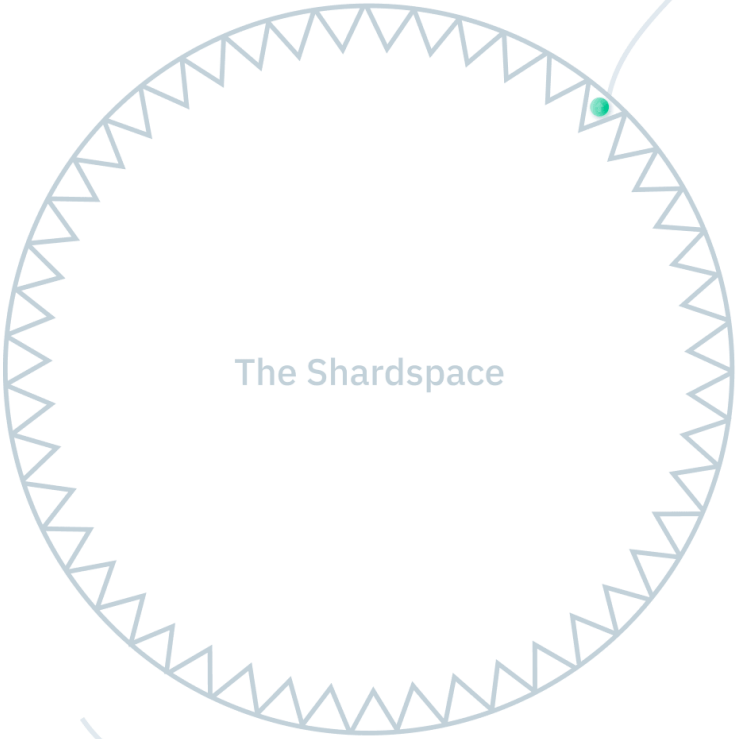
identfier for the transaction. This is done by taking the transaction data, and hashing it. The output is a hash that is 256 bits (78 digits) long. This number serves as that transaction's unique identifier - its Transaction ID.

We can then add the index number of the bring up substate to the Transaction ID, which creates a number that is unique to that substate. So in this example, the substate has Index: 1.

We can then apply the hash again to the combined data, to generate a unique 256 bit substate ID.

Hashing algorithms are deterministic. Given the same input data, you will always get the same output data.





The number of possible substate IDs maps perfectly to the 2^{256} shardspace as they're both 256 bit. In other words, the substate ID *is* that substate's shard address. The two are inextricably linked.

This means that just by looking at a substate, any piece of software can instantly determine what shard it belongs in. The substate can't live anywhere else.

Substates are scattered across the shardspace as Substate IDs can be expected to be nearly uniformly distributed across the shardspace. This "load balances" the computational requirements across the shardspace.

Because the shardspace is so large, the chances of there being a "collision" is practically zero. Even at a trillion transactions per second, we would have entered the "degenerate" era of the universe (1,000 trillion years from now); where only brown dwarfs, white dwarfs, neutron stars, and black holes remain; prior to there being a ~0.0000001% chance of a single shard collision during that entire period [1] . And in the extremely unlikely event that there is a collision, the transaction would just fail.

Additionally, the potential for attack or disruption by an adversary taking over any single shard is low, as a single shard only ever contains a single substate.

Finally, as the shardspace is fixed-size, static, and well-known ahead of time, no index is ever required, as every substate has its own fixed shard. This removes the need for complex dynamic sharding procedures that other sharded networks have, as when they try and scale by adding new shards (such as new blockchains that run in parallel), lots of communication is often required redefining where everything lives.

This unique approach is a key part of what enables Cerberus to process an almost unlimited number of parallel transactions, as Cerberus will never run out of shards.

So now we know what substates are, how they're created, and how they're allocated a shard address, but what does an actual transaction look like?

[1] Trillion = 10^12; seconds in year = 3 × 10^7; degenerate era = 10^15 years

Therefore (10^12) × (3*10^7) × (10^15) = 3 × 10^34 total transactions at a trillion TPS until the degenerate era of the universe

Using a "birthday attack", ~0.0000001% (10^-9) odds for a collision in a 2^256 shardspace occur after 1.5 × 10^34 hashes

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

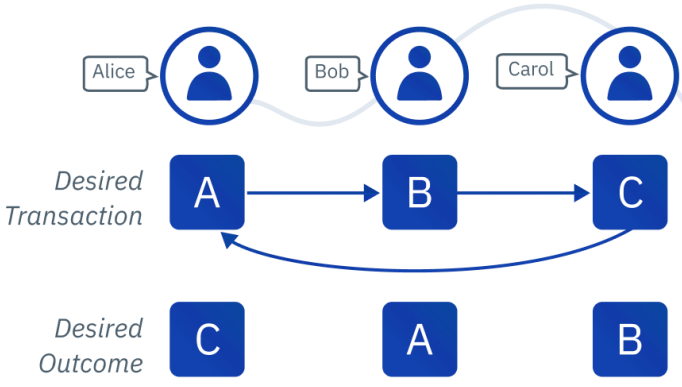
Transactions

So let's walk through how someone would go about exchanging Token A for Token C, using Token B as an intermediary step, all in a single transaction. We will use this transaction for the rest of this infographic series.

In this example Alice owns Token A, Bob Token B, and Carol Token C.

Alice wants Token C, Bob wants Token A, and Carol wants token B.

It's not possible for this transaction to work with just a two way swap. A three way swap is needed.



The transaction shown here is exactly the same as the transaction shown below. Here it is presented as text, there it is presented diagrammatically.

Every node in every relevant validator set stores a copy of the whole transaction, including all the substates.

So the first thing that's needed is for Alice, Bob, and Carol to agree on the transaction and that they will all be

Transaction (Tx)

Shard 1: *Shut down Alice's Token A substate*

willing to sign it. This can be done offline.

Once agreed, they use a client, such as a “wallet”, to create the transaction. The client software will generate the substates that need to be written to the ledger, and the shards those substates belong in. We covered how substates get allocated shards in the previous episode.

So in this example, three shut down substates are created to record that the tokens, as per the old ownership, can no longer be spent; and three bring up substates are created to record the new owner for each token.

Alice, Bob and Carol then cryptographically sign the transaction proving that they have the authority to send the tokens and that they approve the transaction.

Shard 1: *Shut down Alice’s Token A substate*

Shard 2: *Bring up Bob’s Token A substate [Index: 0]*

Shard 3: *Shut down Bob’s Token B substate*

Shard 4: *Bring up Carol’s Token B substate [Index: 1]*

Shard 5: *Shut down Carol’s Token C substate*

Shard 6: *Bring up Alice’s Token C substate [Index: 2]*

Signed: *Alice, Bob, Carol*

Tx ID: ...979

The client thus performs all the computation necessary to define exactly what needs to be written to the ledger - everything is contained in the transaction.

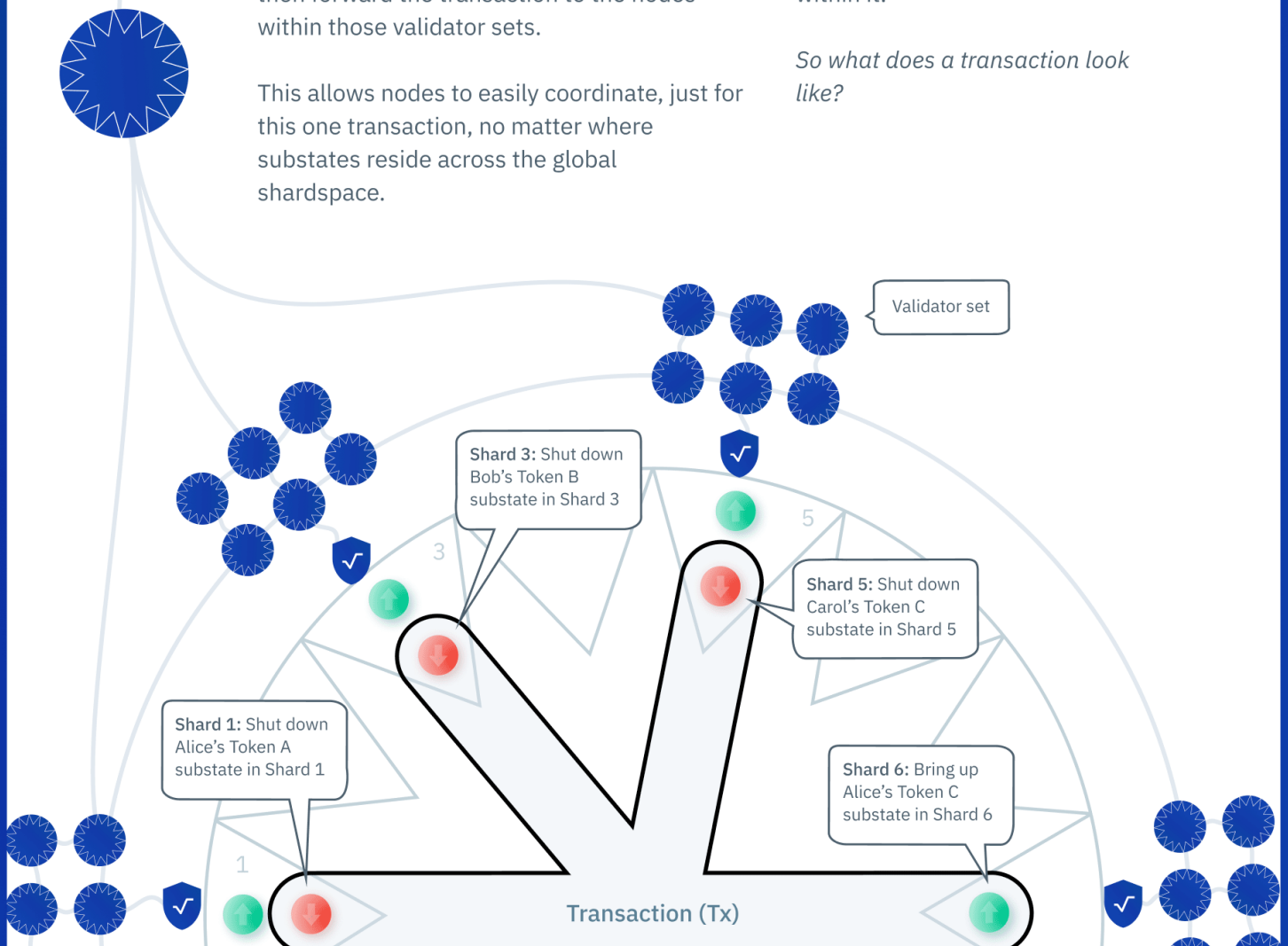
The client then submits the transaction to any node on the network.

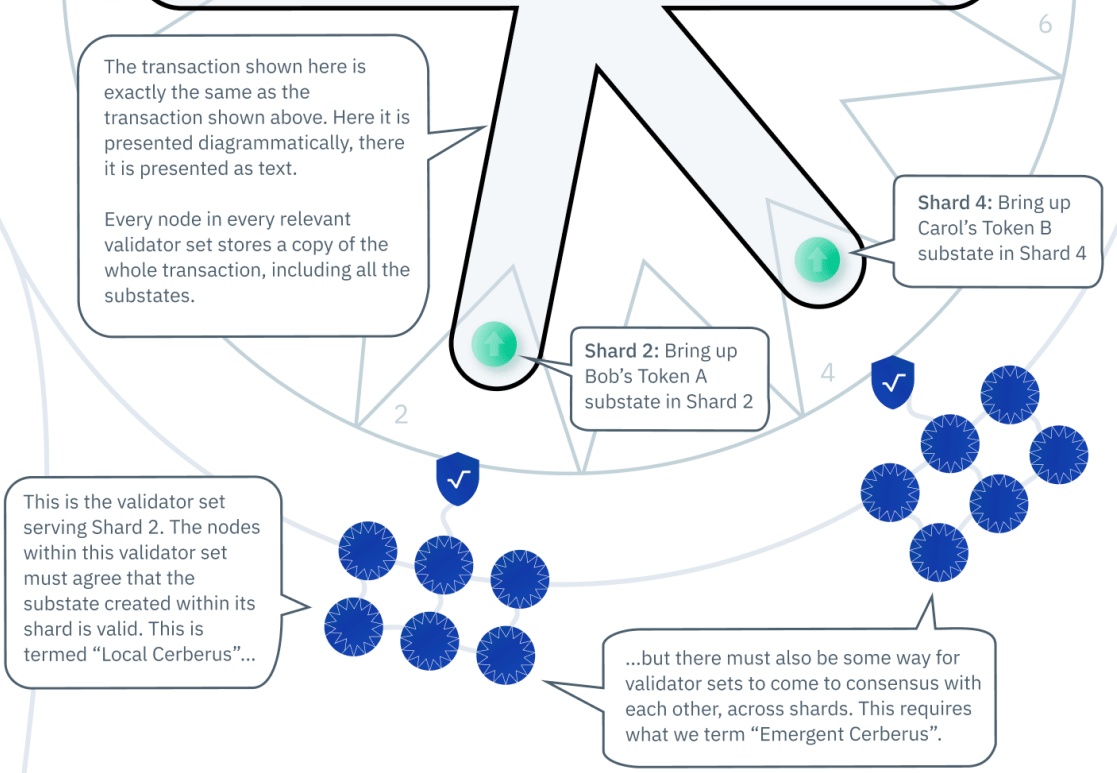
Once received by a node, the node can quickly look up the validator sets that serve the shards defined in the transaction, as this information is stored on-ledger. The node can then forward the transaction to the nodes within those validator sets.

This allows nodes to easily coordinate, just for this one transaction, no matter where substates reside across the global shardspace.

The job of a validator set is to verify transactions for the shard it serves, and store a permanent record of the transaction and the substates within it.

So what does a transaction look like?





The above diagram shows six substates being created by six validator sets across six shards, all in one transaction. It contains:

- three shut down substates created to record that the input substates can never be used again in a transaction
- three bring up substates created to record the new owners of the tokens.

But all we've shown here so far is the result of the transaction. We've not described how the nodes within those validator sets communicate with one another and agree that the transaction is valid, in consensus.

To do that, Cerberus' method of reaching consensus is actually composed of two tightly integrated parts that are both required for the transaction to work.

Those two parts are:

(1) Between the nodes within a validator set for a single shard - this is "Local Cerberus".

(2) Between the validator sets across shards - this is "Emergent Cerberus".

It's important to note that as all transactions are cross-shard, it is actually emergent Cerberus that nodes use to come to consensus. Local Cerberus is just a theoretical stepping stone to help us understand emergent Cerberus.

We start by providing an overview of consensus, and then move on to both Local Cerberus and Emergent Cerberus.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

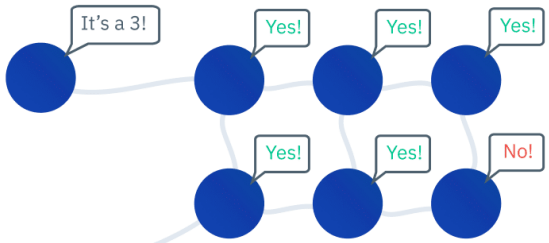
v1.0 · June 2021

Nakamoto vs BFT-style Consensus

So before jumping into Cerberus, we go back a step and first provide a little background on consensus.

Consensus is a problem of coordination between nodes - getting them to either agree or reject something, together.

This means the nodes need a protocol - some established set of procedures and rules, that governs how they communicate with one another and come to a decision.



Nakamoto Consensus

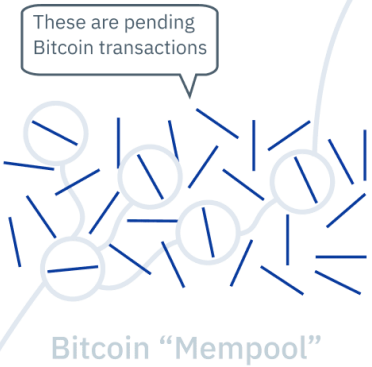
Bitcoin uses a consensus protocol some people call "Nakamoto" consensus.

In Bitcoin, nodes select transactions that are pending from users, and attempt to solve a cryptographic puzzle - performing lots of computation to find a special kind of hash derived from the data of those selected transactions.

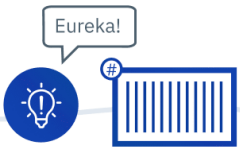
The first node to find that hash then writes a block with those selected transactions and proposes this to the network. If other nodes agree that the block and hash are valid, then those nodes also write that block to their own copy of the blockchain, and that block propagates throughout the network.

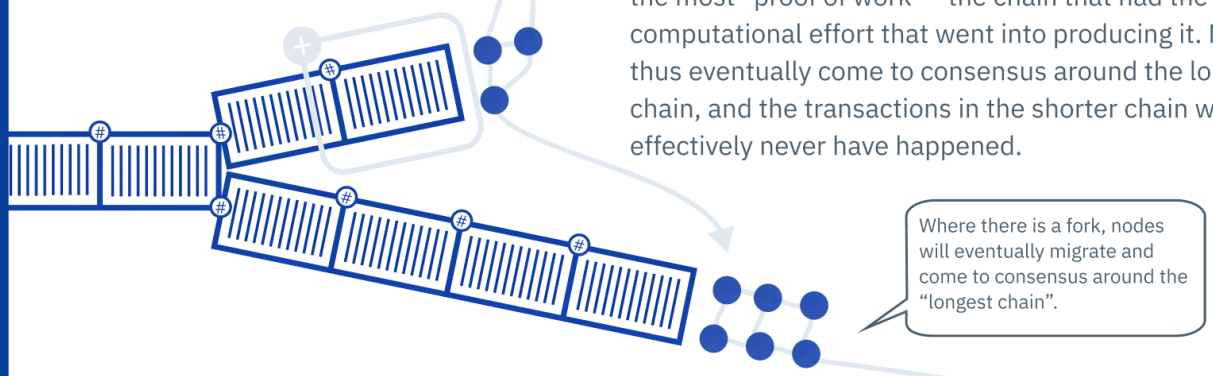
If nodes don't agree, we have what's called a "fork" such as that depicted to the bottom left. This is where some nodes maintain a record of one set of blocks, and other nodes keep a record of another set.

However, with time, nodes will migrate over to the "longest chain" as that is the chain that can demonstrate the most "proof of work" - the chain that had the most computational effort that went into producing it. Nodes thus eventually come to consensus around the longest chain, and the transactions in the shorter chain will effectively never have happened.



Bitcoin "Mempool"





Because of this, Nakamato consensus is considered "probabilistic", as the node that proposes the transactions that go into a particular block is not

However, over time, as a chain gets longer and longer, the chances of there being another chain with more proof of work effectively becomes so small, that

predetermined; and you can never be 100% sure when a transaction is “final”, as there might always be a longer chain out there.

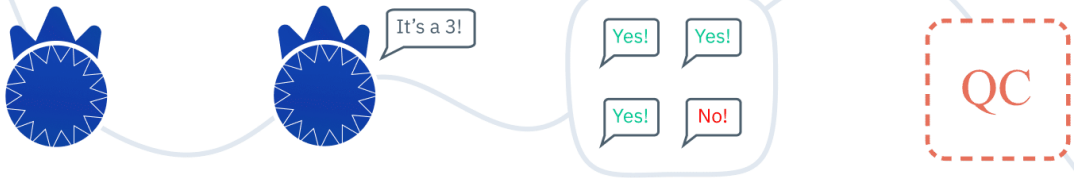
you can be almost certain that your transaction is on the longest chain, making it effectively final. This is why some exchanges require you to wait 60 minutes, even though the Bitcoin block time is roughly 10 minutes.

BFT-style Consensus

Cerberus falls under a different class of consensus protocol - one that can be referred to as “Byzantine Fault Tolerant style”, and can be considered to be “deterministic”.

This is because this style of consensus has clear rules about who can vote on a transaction and exactly how many votes are needed before everyone can agree the transaction is 100% final. Once a transaction is finalized, there is no way of there being a "longer chain" out there, as there is no uncertainty in the process.

In this class of consensus algorithm, the node that gets to propose the next transaction is called the "leader" of that particular consensus round.



This is a leader.

The role of leader is to...

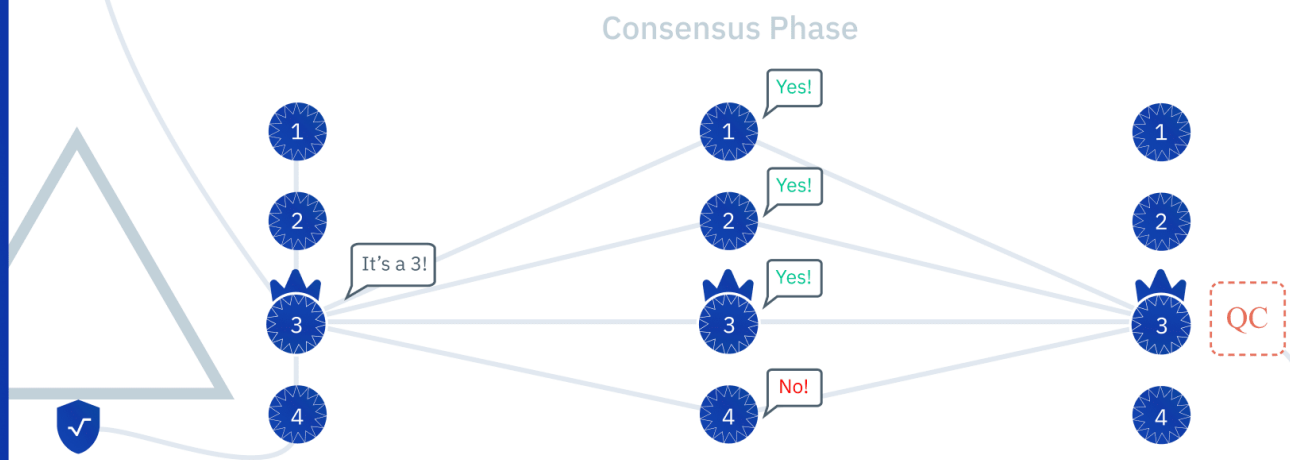
...submit messages to the nodes within their validator set and to drive progress through “phases”...

...collect the votes from nodes on whether they agree with the message...

...and to then broadcast the aggregation of those votes back to the nodes, proving that enough nodes agreed, in what’s called a Quorum Certificate.

Phase

A phase is a series of steps that a collection of nodes undertake within consensus. Here is what a typical consensus phase looks like for a hypothetical validator set with just four nodes (in reality there would be many more). A phase is composed of three steps:



1. Broadcast

Here we can see the leader broadcasting a message to the other nodes in the

2. Vote

All nodes then vote on whether the message is valid. They do this by

3. Create Quorum Certificate

The leader then gathers the votes and creates a Quorum

validator set.

The message in the first phase is a proposal. In later phases, the message is the QC from the previous phase (we explain what this means later).

cryptographically signing their vote, and sending their vote to the leader.

Certificate - cryptographic proof that enough votes were in agreement with the message.

If there aren't enough votes, no valid QC can be produced.

Choosing a leader

In Cerberus, the leader within each validator set shuffles periodically, after every "consensus round".

The selection process is deterministic, and is agreed between nodes in a similar way to the consensus phase described above. Nodes therefore know, at any given point in time, who the leader of that validator set is. In consensus lingo, when a leader changes, this is called a "view change".

So those are the basics. Let's now move on to local Cerberus.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

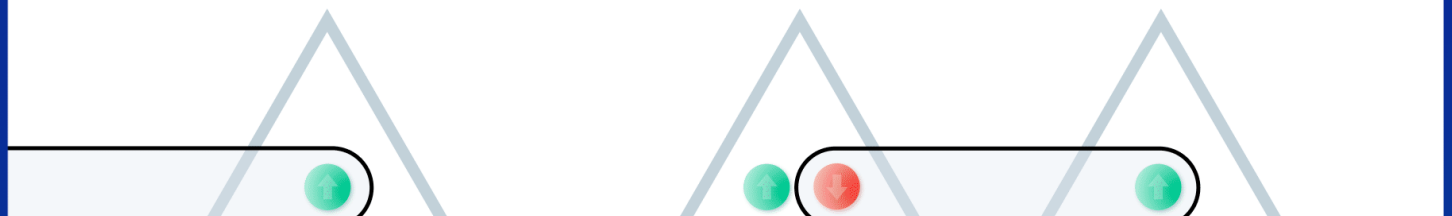
Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

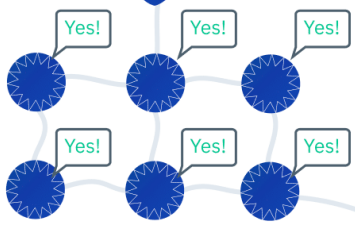
Consensus - Local Cerberus

So now on to Cerberus. As previously mentioned, Cerberus' method of reaching consensus is in two tightly integrated parts:

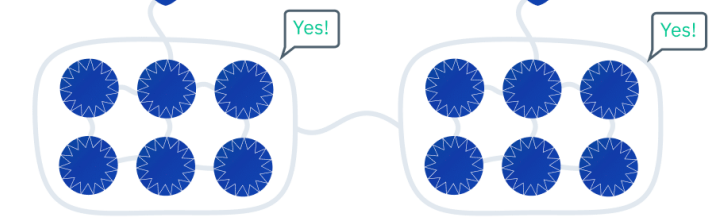
Local Cerberus

Emergent Cerberus





Local Cerberus, which is how nodes come to consensus with each other, for a single shard, within a validator set.



Emergent Cerberus, which is how nodes come to consensus across shards, between validator sets.

We start with local Cerberus.

This is the same transaction as in episode 9.

So as before, a client submits a transaction to the network.

Once a node picks up the request, it forwards the transaction to the "leader" of each relevant validator set.

We can see that as this transaction covers six shards, the node that picks up the transaction forwards the transaction to six different leaders - one for each validator set serving each shard.

Transaction (Tx)

Shard 1: *Shut down Alice's Token A substate*

Shard 2: *Bring up Bob's Token A substate [Index: 0]*

Shard 3: *Shut down Bob's Token B substate*

Shard 4: *Bring up Carol's Token B substate [Index: 1]*

Shard 5: *Shut down Carol's Token C substate*

Shard 6: *Bring up Alice's Token C substate [Index: 2]*

Signed: *Alice, Bob, Carol*

Tx ID: ...979

As we are covering local Cerberus here only, let's see what happens just for:

"Shard 1: Shut down Alice's Token A substate".

This is the final vertex from the previous transaction. Vertices are stored by nodes alongside every transaction, as proof that

Proposal Vertex

When the leader of the validator set serving Shard 1 receives the transaction, it first verifies that the transaction is valid. Once verified, it broadcasts the transaction to all the other nodes in the validator set. As this is the first message, this is the "proposal".

Each node then verifies the transaction and votes - either agreeing or disagreeing that the transaction is valid. The nodes then cryptographically sign their votes and send their votes to the

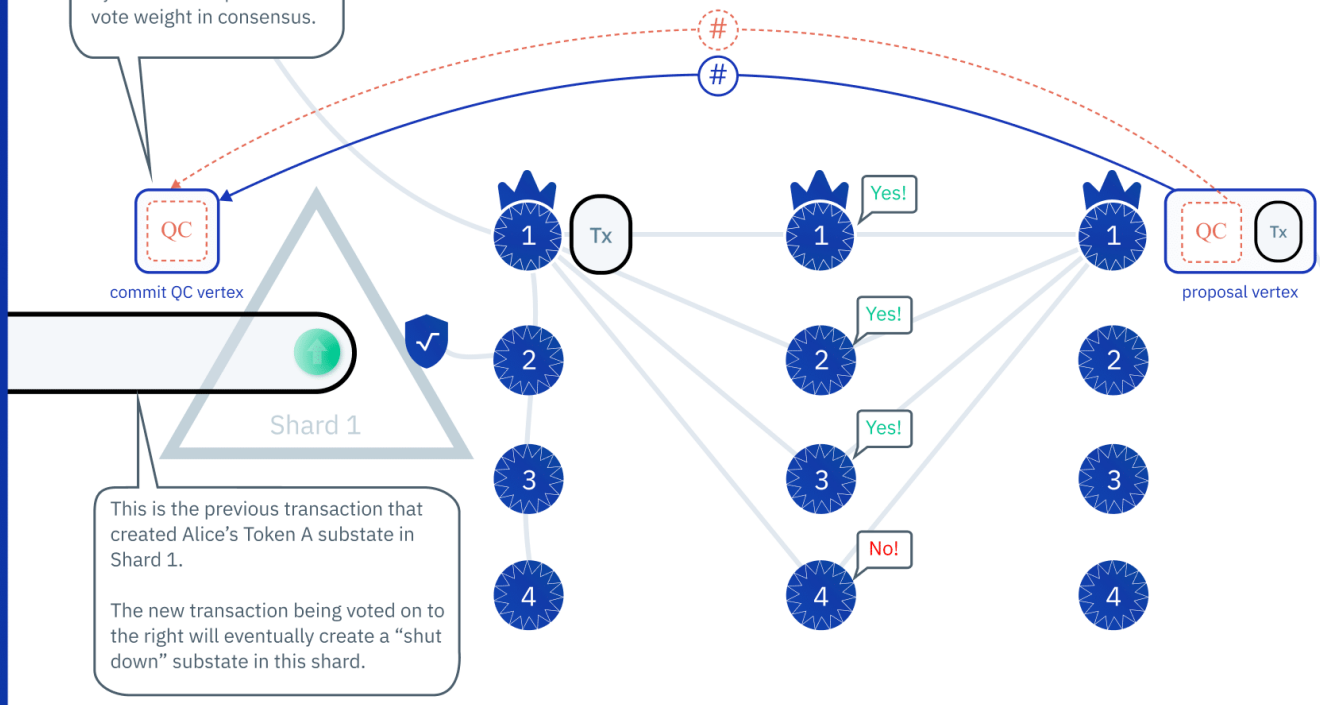
As the nodes in this example only have visibility of Shard 1, when they sign, they're only signing that they're happy with the transaction as it relates to Shard 1.

The leader then collects all the signed votes. If enough nodes agree with one another, we have a quorum.

In Cerberus, it's not just a simple case of one node one vote - achieving a quorum is based on

The nodes then cryptographically sign their votes and send their votes to the leader.

case of one node one vote - achieving a quorum is based on "vote weight".



Cerberus weights votes made by nodes by how many tokens are "staked" to them. We cover how this works in episode 15 - Sybil Resistance Through Proof of Stake. For now we'll just assume one node one vote.

In Cerberus, you need "2f+1" of vote weight, or just over 2/3rds, to achieve a quorum. "f" means the maximum tolerated number of "faulty" nodes, so a threshold of 2f+1 means that just over 2/3rds of vote weight has to be non-faulty, or trustworthy.

If a quorum is achieved, the leader takes all the signed votes, and issues what's called a Quorum Certificate (QC). The QC provides cryptographic proof that a 2f+1 majority of vote weight in the validator set approved the transaction.

But the leader doesn't just create a QC on its own - it creates a data structure called a vertex.

The vertex contains:

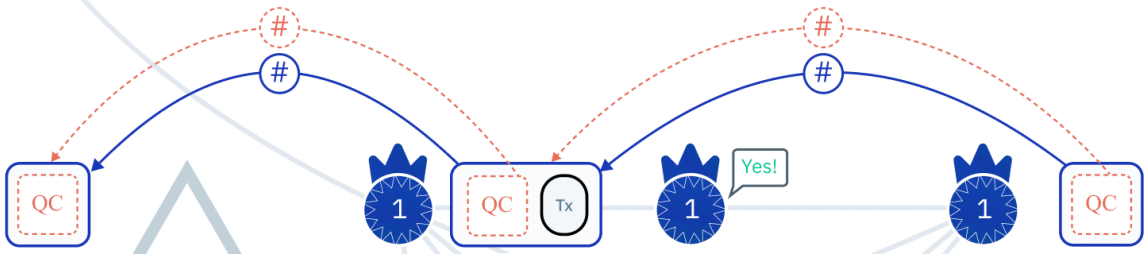
- the transaction
- the QC proving the transaction was agreed by 2f+1 of vote weight
- hashes tying both the QC and the proposal vertex back to the final vertex from the transaction that created the input substate.

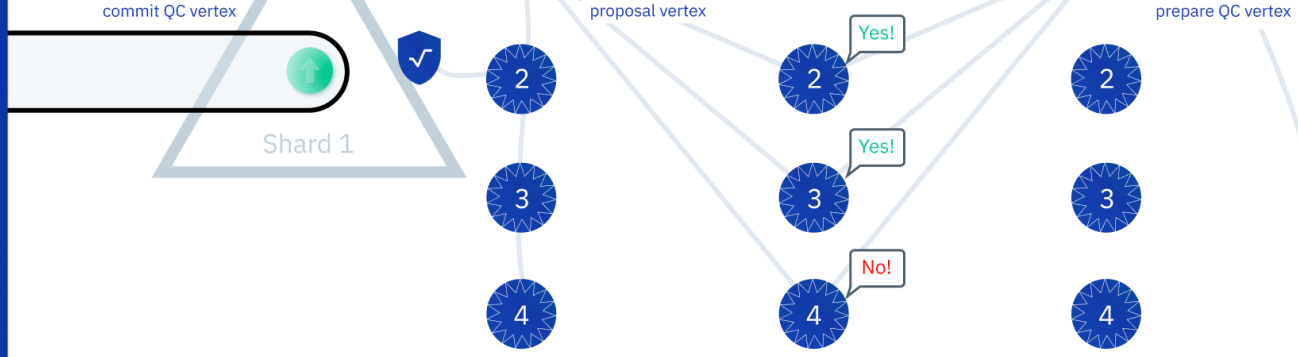
As this vertex contains proof that the nodes agreed on the proposal, it's called the "proposal vertex".

Prepare QC Vertex

We then move on to the next phase, where the process repeats. The leader now broadcasts the proposal vertex to relevant nodes, and the nodes then vote on whether the proposal vertex is valid.

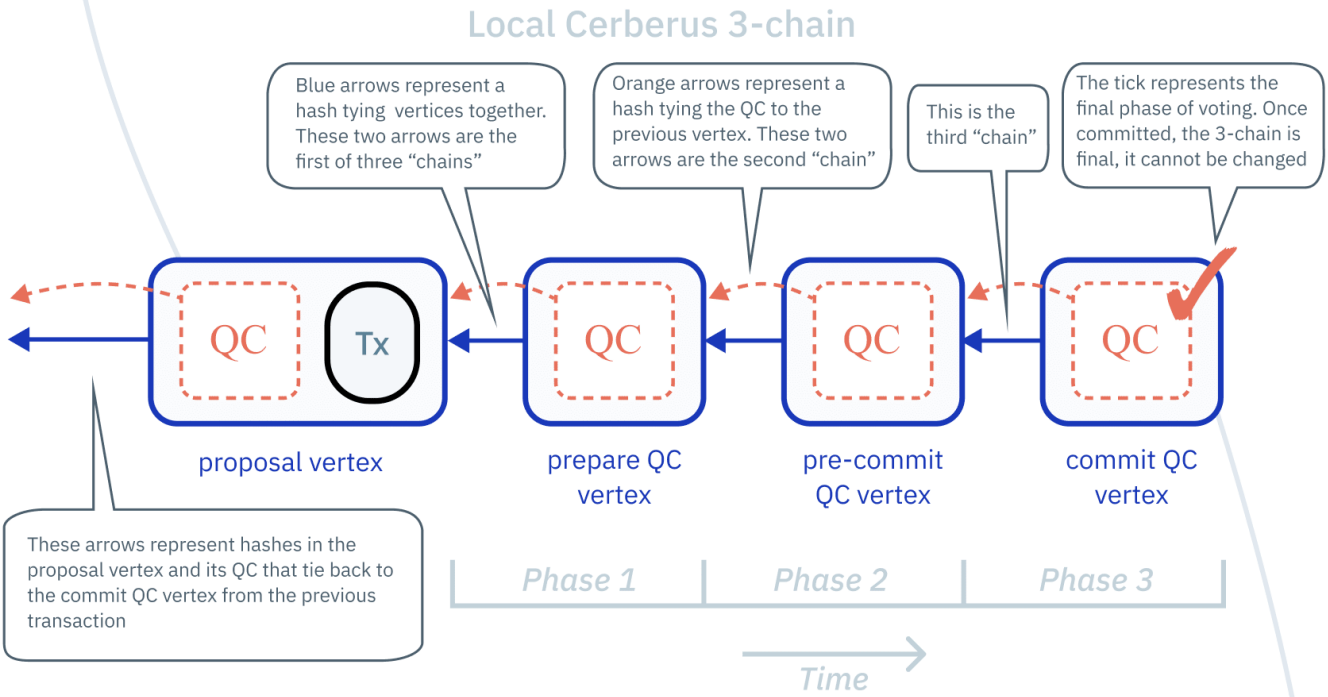
If 2f+1 of vote weight agrees with the proposal vertex, then the leader packages up proof of this in another QC, in another vertex, with hashes back to the proposal vertex. This next vertex is called the "Prepare QC Vertex".





Pre-commit QC Vertex and Commit QC Vertex
Two more phases of voting then take place, for a total of three phases after the proposal vertex. This is why Cerberus is described as being a “Three Phase Byzantine Fault Tolerant” consensus protocol.

If we zoom out on all these rounds of voting, the process looks like this:



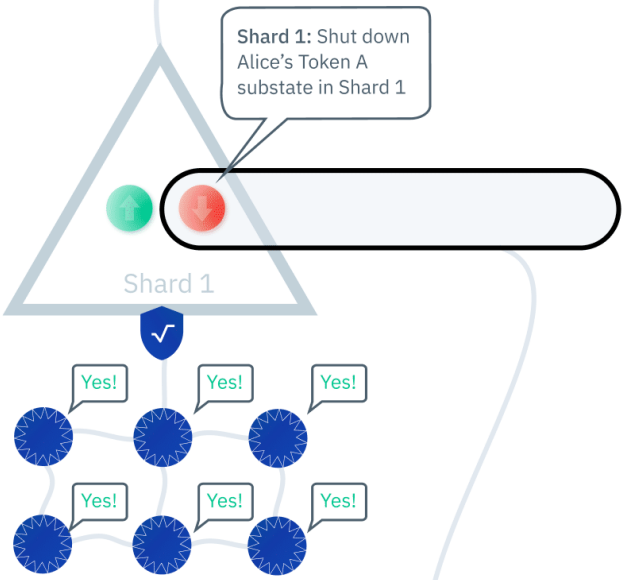
Following the creation of the proposal vertex, you can see that there are three phases of voting before the transaction is final. Progression through each phase is marked by the creation of a vertex.

So that’s a lot of voting. Why go through the trouble of all these repeated phases?

The answer is that two phases of voting is the minimum that’s required to allow nodes sufficient opportunity to be sure that they are all in agreement, but still allow them to safely fail the transaction at any phase if a quorum of vote weight cannot be achieved. Two phases of voting is common in other BFT-style consensus protocols such as “Tendermint”.

However, one of the drawbacks of just two phases of voting is that nodes must wait a defined period of time for each round of consensus to finish. This limits the throughput of the network and slows things down.

Having three phases of voting allows consensus to proceed "optimistically". This means that consensus rounds can be completed as fast as the nodes can exchange enough votes – there is no longer a need for a fixed waiting period. This is the key innovation of a BFT-style consensus protocol called "HotStuff" that local Cerberus draws heavily on.



To close the loop, we can see that as the nodes in the validator set serving Shard 1 have come to consensus - evidence of which is contained in the commit QC vertex - they are now able to create the shut down substate in Shard 1 as per the transaction.

So that's how local Cerberus allows nodes within a validator set to come to consensus.

But local Cerberus on its own wouldn't allow a single transaction to complete, as all transactions are cross-shard.

To understand how cross-shard transactions work, we now cover emergent Cerberus.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

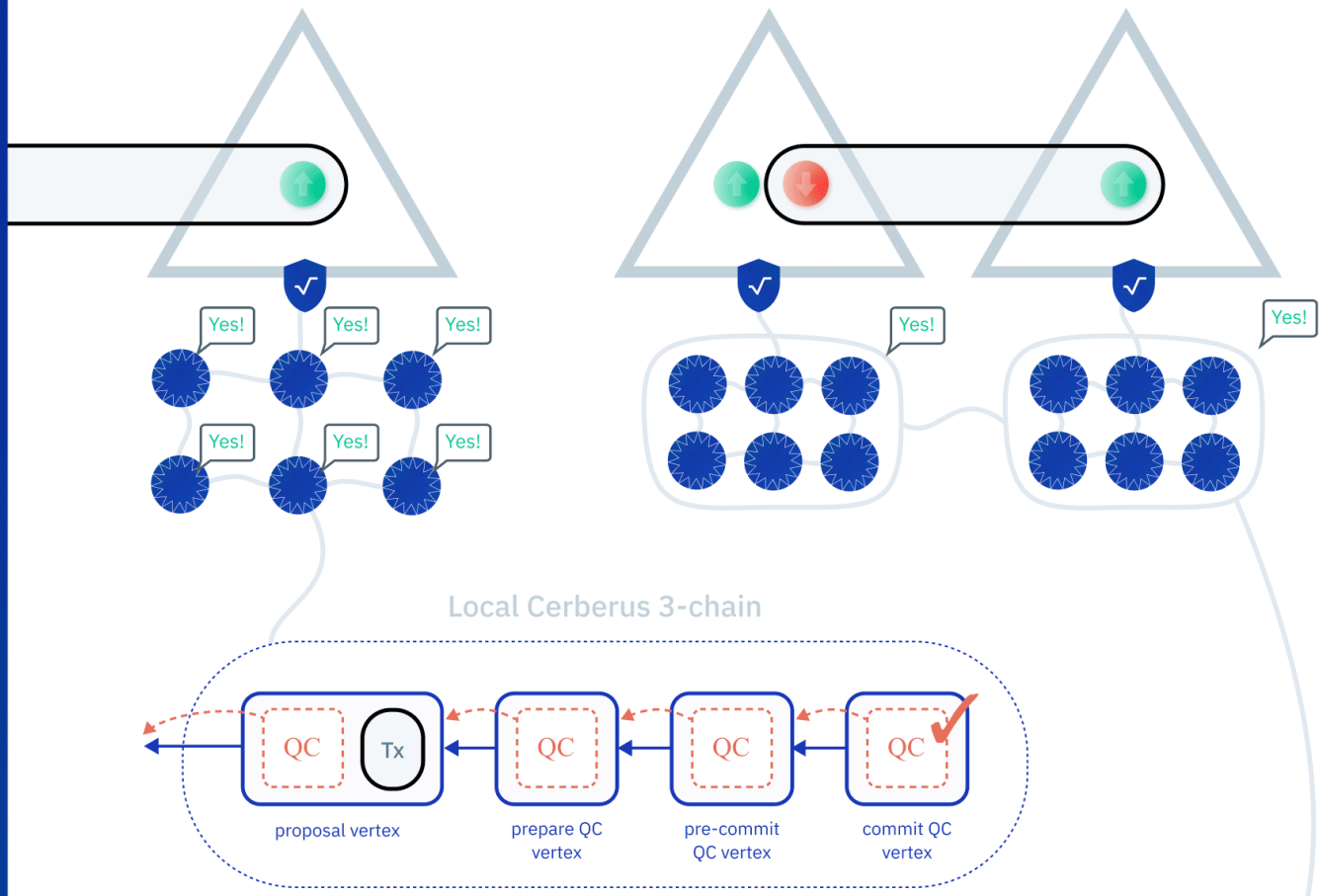
while preserving atomic composability

Consensus - Emergent Cerberus

So far we've covered local Cerberus - how nodes come to consensus within a single shard. But as you know, all transactions are cross-shard. There must therefore be a way for validator sets to come to consensus with other validator sets, across multiple different shards.

Local Cerberus

Emergent Cerberus

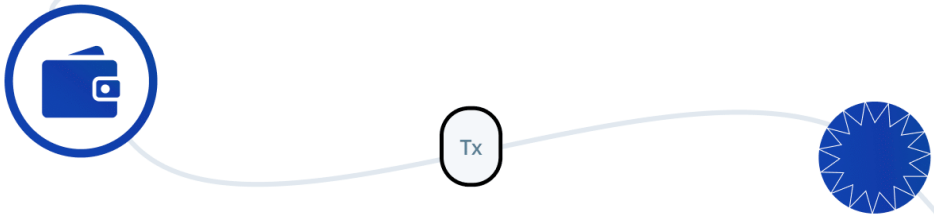


So we previously explained how nodes following local Cerberus form a “3-chain”.

Each “chain” represents the hashes that tie the vertices together.

The vertices provide cryptographic proof that 2f+1 of vote weight had voted on and agreed with the previous vertex.

We now cover emergent Cerberus.



So revisting our transaction, a wallet submits our transaction to a node, and that node then forwards that transaction to the leaders for each relevant validator set.

As we are covering emergent Cerberus, let’s see what happens for two shards:

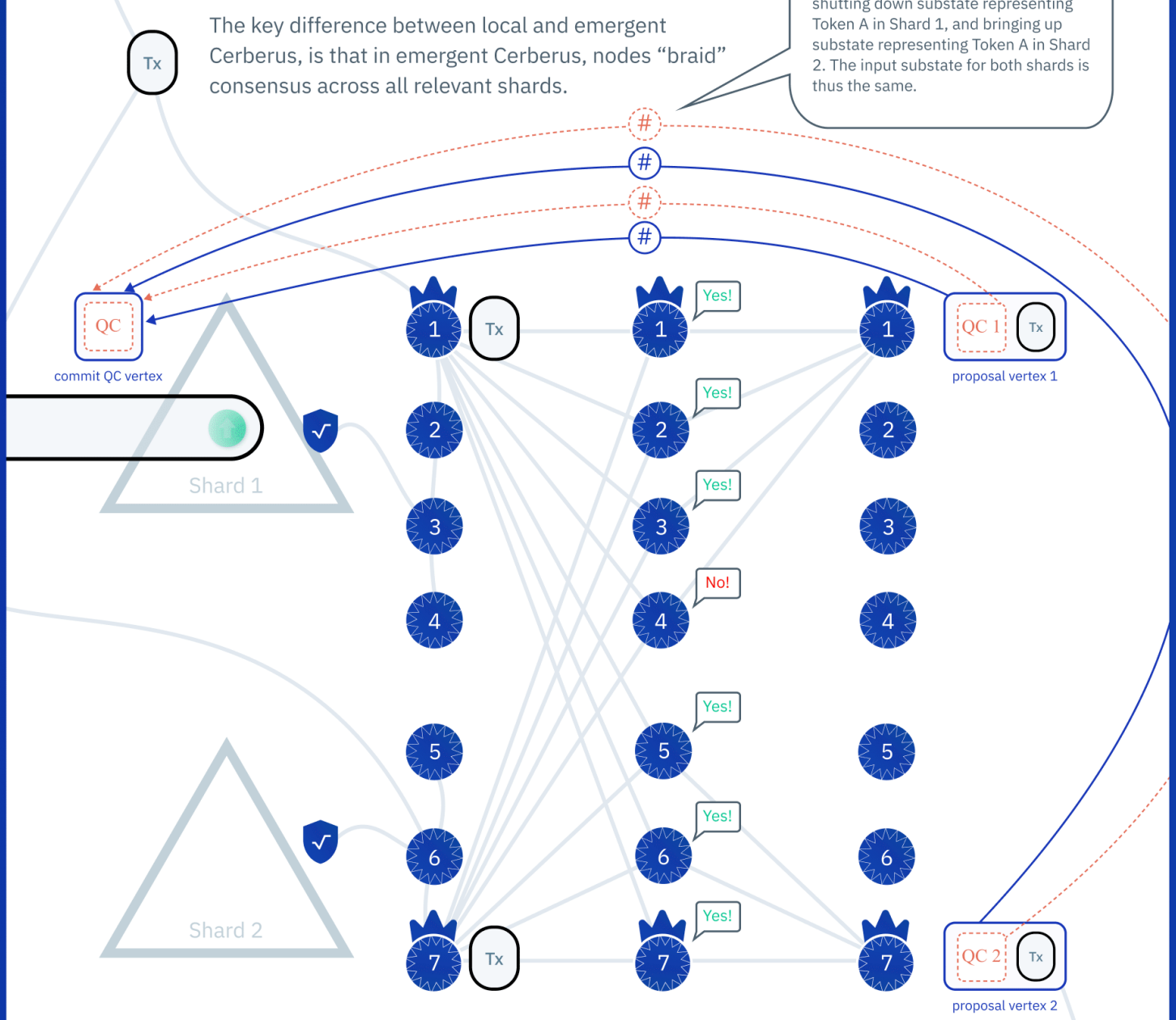
“Shard 1: Shut down Alice’s Token A substate”.
”Shard 2: Bring up Bob’s Token A substate [Index: 0]”

Proposal Vertex

So in the example below, we show the same four nodes for Shard 1 (nodes 1-4); and three nodes for Shard 2 (nodes 5-7). The leaders of each respective shard are nodes 1 and 7.

In this example, the hashes for the proposal vertices for both Shard 1 and Shard 2 tie back to the same commit QC vertex from the previous transaction.

This is because in our example, we are



1. Broadcast
So for the broadcast step, instead of broadcasting the transaction to only nodes within its validator set, each leader broadcasts the transaction to all nodes across all relevant validator sets.

2. Vote
Then, when nodes vote, they are voting on whether the transaction:

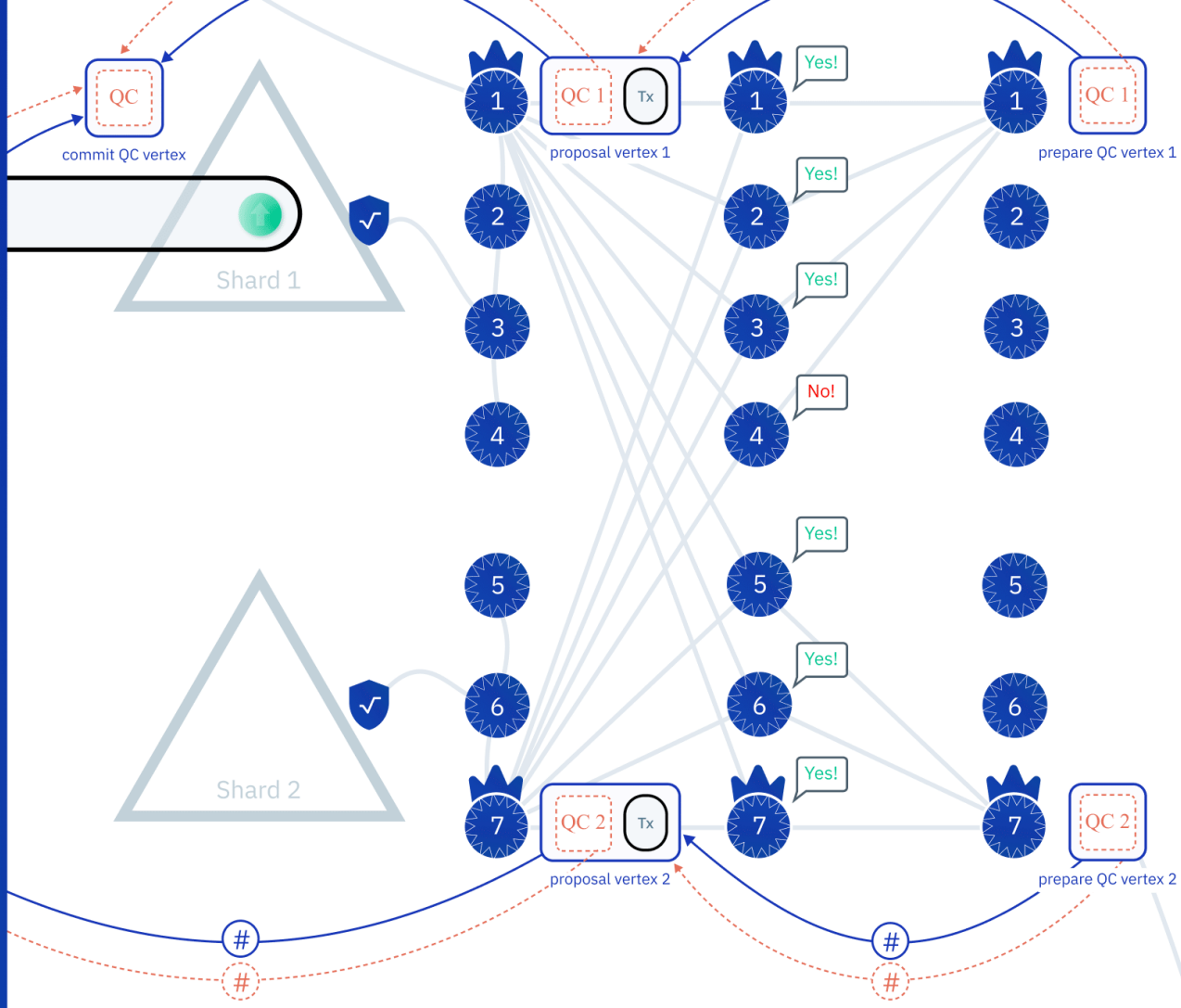
(i) sent by both leaders is identical
(ii) is valid as it relates to only their own shard.

This is because nodes only have data for their own shard, and so they send their vote back to only the leader for their shard.

3. Proposal Vertices
The leaders of each validator set then collect the votes for their own respective shards, and create two separate proposal vertices - Proposal Vertex 1 for Shard 1, and Proposal Vertex 2 for Shard 2. While the transaction is the same in these vertices, the QCs are different, as they represent the votes for different shards.

These vertices both tie back to the same commit QC vertex from the previous transaction, as these are the shut down and bring up substates for the transfer of ownership for Token A.

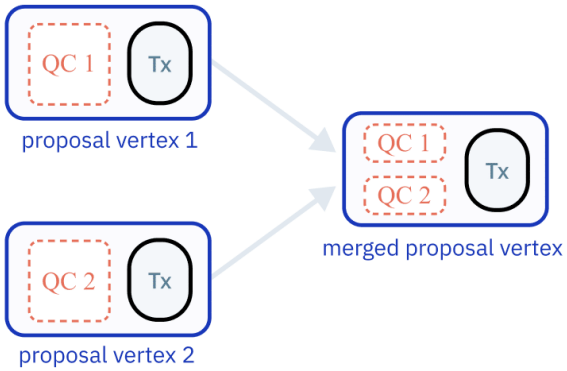
Prepare QC Vertex
For the next phase, the process repeats: the leader of each validator set now broadcasts their proposal vertex to all nodes across all validator sets.



This means that every node across every relevant validator set has a copy of both Proposal Vertex 1 and Proposal Vertex 2 - they have the QCs for both shards.

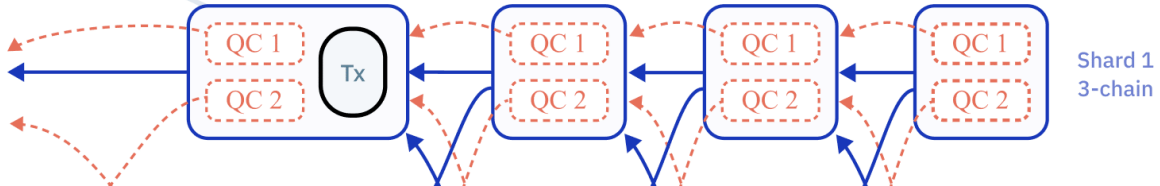
Each node can then merge the proposal vertices and create a “merged proposal vertex”.

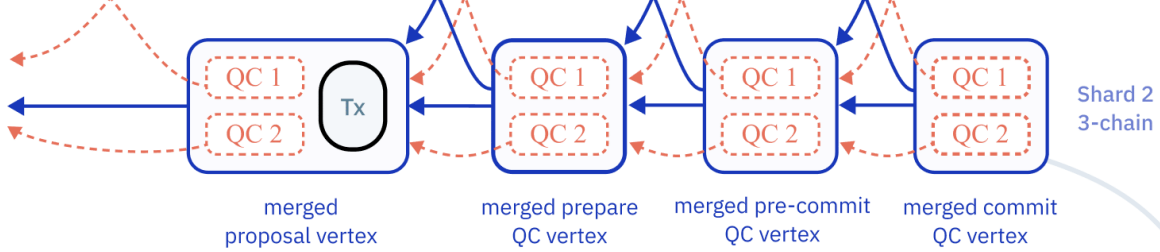
When nodes vote, they’re then actually voting on whether the merged proposal vertex is valid. They then send this vote back to their local leader, who packages those votes into a Prepare QC Vertex - one for each shard.



Pre-commit QC Vertex and Commit QC Vertex

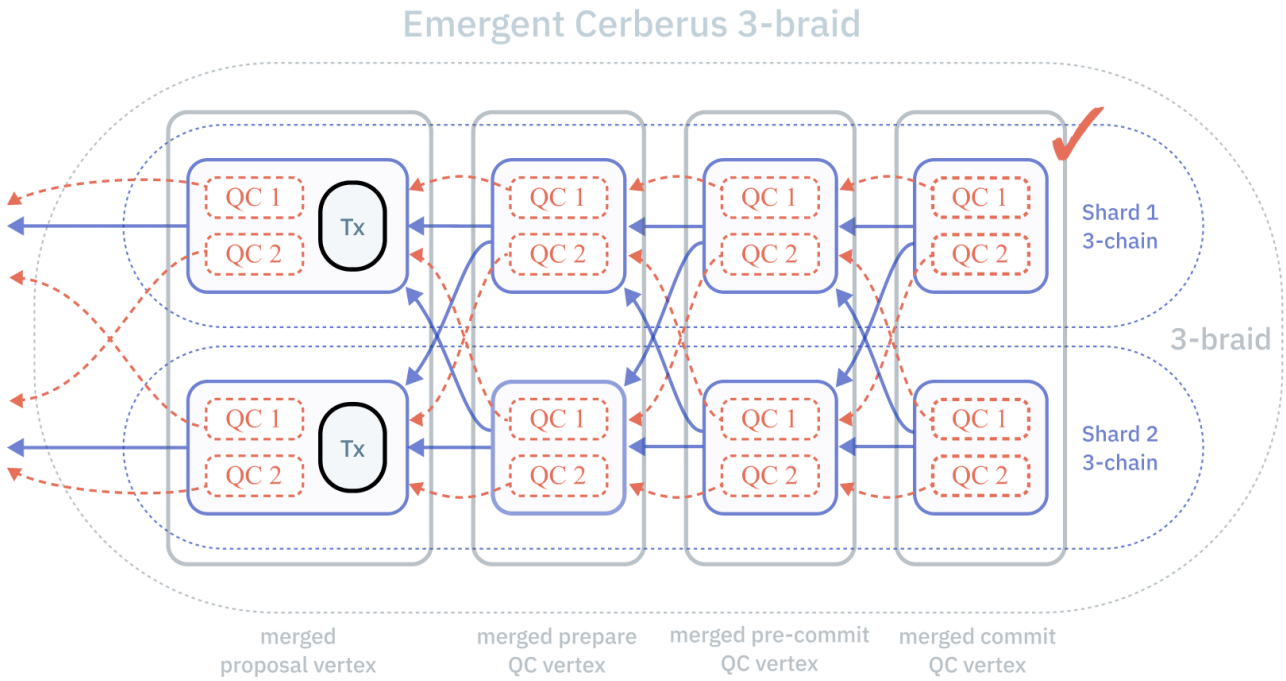
The exact same process takes place for the last two phases, where vertices are merged across every phase of consensus.





As you can see, instead of a “3-chain”, we get a “3-braid”, where QCs and hashes are braided between every shard.

If we then draw shapes around the vertices, we can see how the merged vertices intersect with each of the 3-chains. And together, a 3-braid forms.

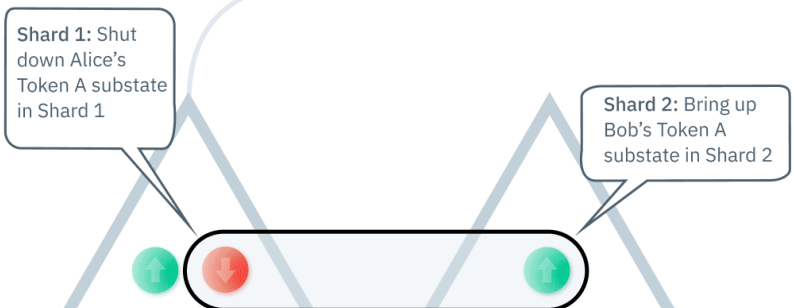


We can see that within each 3-chain above, every node has a copy of every QC for both Shard 1 and Shard 2, and that every vertex and every QC is chained to all other previous QCs and vertices across each shard.

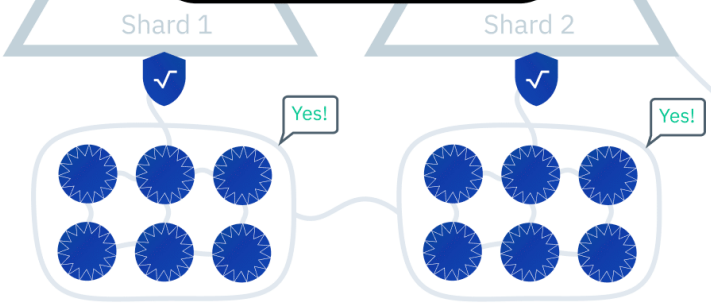
Progress through each phase therefore only happens when the leader of every shard is able to obtain QCs from all shards. Only QCs need to be shared - the data in each shard does not need to be passed around.

Every node in every validator set therefore has proof that every other validator set had obtained and approved the previous vertex, for each phase.

Validator sets can thus come to consensus with other validator sets and they write the transaction to relevant shards in the ledger together, atomically. If any validator set doesn’t agree with the transaction, at any phase, the whole transaction is able to safely fail.



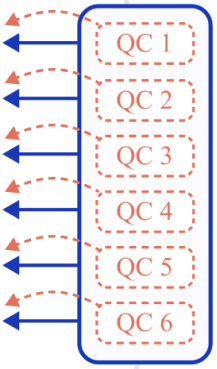
To close the loop, we can now see that the nodes across both validator sets are able to create the relevant substates across both Shard 1 and Shard 2 in a single atomic transaction, as consensus was braided.



However, our example transaction doesn't just involve two shards. It involves six!

To braid consensus across two or more shards, all you do is braid more 3-chains together.

So if you were to braid six shards together, each merged vertex would contain six QCs, as depicted to the right.



The cross-shard braiding of consensus in emergent Cerberus is perhaps its single most important innovation. It's what allows transactions to be composed across any number of shards across the entire shardspace, containing any combination of substates, with validator sets committing or rejecting all the substates within those transactions atomically - either all together, or not at all.

Given that all transactions are composed and settled across shards, so long as there are enough shards and nodes to serve them, there will never be a limit to how many transactions the network can process in parallel. The lack of composability that results from sharding blockchains is just not a thing in Radix, as the network has an almost unlimited number of atomically composable shards right from the very start. Radix therefore scales linearly with every additional node added to the network, without ever adding any friction.

Moreover, one of the key concepts that allows this to happen is that of parallel processing. In a future where Radix has scaled to fulfill the DeFi needs of billions of people, the vast majority of transactions will involve completely unrelated substates, shards, and validator sets. Those transactions will be processed entirely in parallel across the vastness of the shardspace without ever being "aware" that the other transactions exist.

But not everything can be processed in parallel. There must still be some logic that tells Cerberus when substates are related and *must* be composed in the same transaction.

We cover this in the next episode - Partial Ordering.

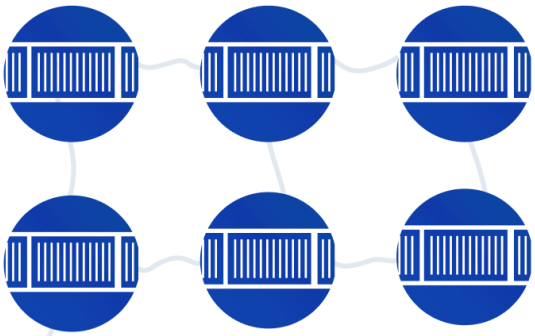
The version of Cerberus described in this infographic series is scheduled to launch as part of the fully

sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

Partial Ordering - Parallelization of Processing 13



If you remember how a blockchain works, such as Bitcoin, every node has to store a record of every transaction from the very first transaction. Then every new transaction, across the whole network, has to be processed by the entire network. The Bitcoin ledger is "globally ordered" (not to mention Ethereum and virtually every other blockchain in existence today).

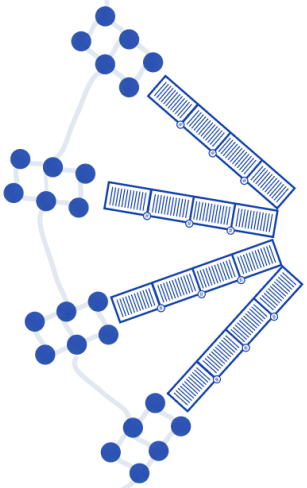
Bitcoin, Ethereum, and other blockchains do this by ordering the transactions within a block, and then ordering blocks in time - chaining each block to the next through hashes; all agreed by the network through consensus.

This severely constrains the number of transactions that can be processed. Imagine you wanted to buy a coffee, but you first had to wait for someone else's coffee on the other side of the world to be processed first! That's what a globally ordered ledger requires.

A more scalable solution would be to parallelize transaction processing. The blockchain shards we talked about in episode 3 are an example of parallel processing.

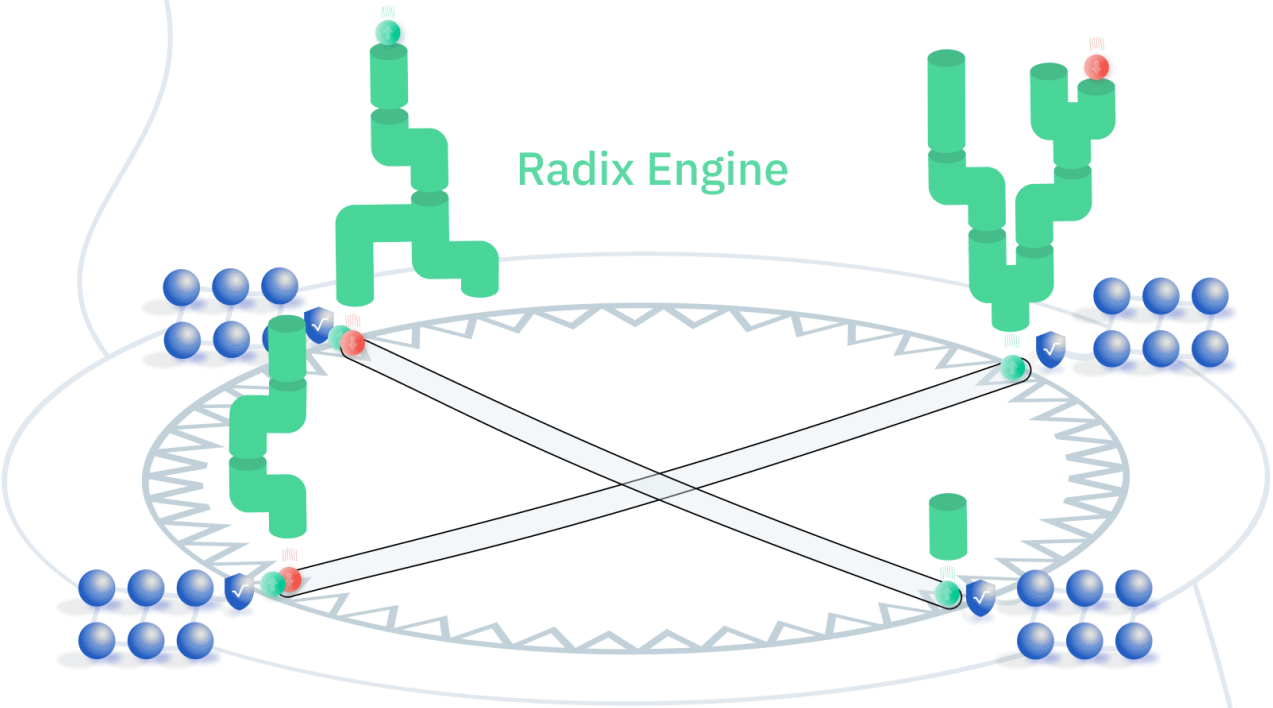
But on a blockchain, you still have to wait for other transactions as transactions are grouped into blocks. Additionally, as we described earlier, blockchain shards face difficulties communicating across shards - breaking "atomic composabilty".

To be truly scalable, what you want is for the network to process your transaction independently, irrespective of anybody else's transaction that is unrelated to yours. It's not waiting around for any other transaction so as to be ordered on a timeline - it just gets processed immediately.



But you also can't have no ordering whatsoever. There will be times when ordering is required because events are related and must execute in a certain order. Think back to the example of selling Token A to buy Token C, the swap from A → B

logically has to happen prior to B → C.



So ordering restricts the throughput of a system; but you do need ordering sometimes.

Therefore ideally, you have a network that's intelligent enough to only order when it's absolutely necessary, so that it can process as many transactions as possible in parallel. We therefore want a network that has "partial ordering".

To make this work, the network needs a layer of logic that specifies dependencies - so that it knows when ordering is required or not required.

For Radix, that layer of logic is the Radix Engine. It tells Cerberus which substates are related and must be ordered, and which ones are not related and do not need to be ordered and can thus be safely parallelized.

So how does the Radix Engine do this?

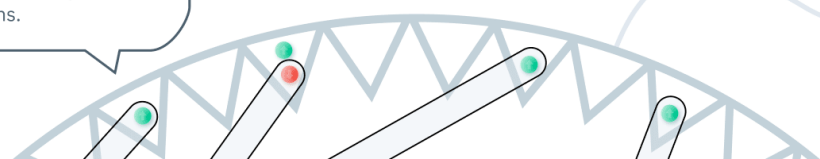
When a transaction is created, it must follow rules stipulated by the Radix Engine. Those rules stipulate that only required substates are used in a transaction, and that when new substates are created, that those are broken down into as many substates as possible, that still make logical sense. This is so that if future transactions need to use those substates, those transactions can be processed in parallel.

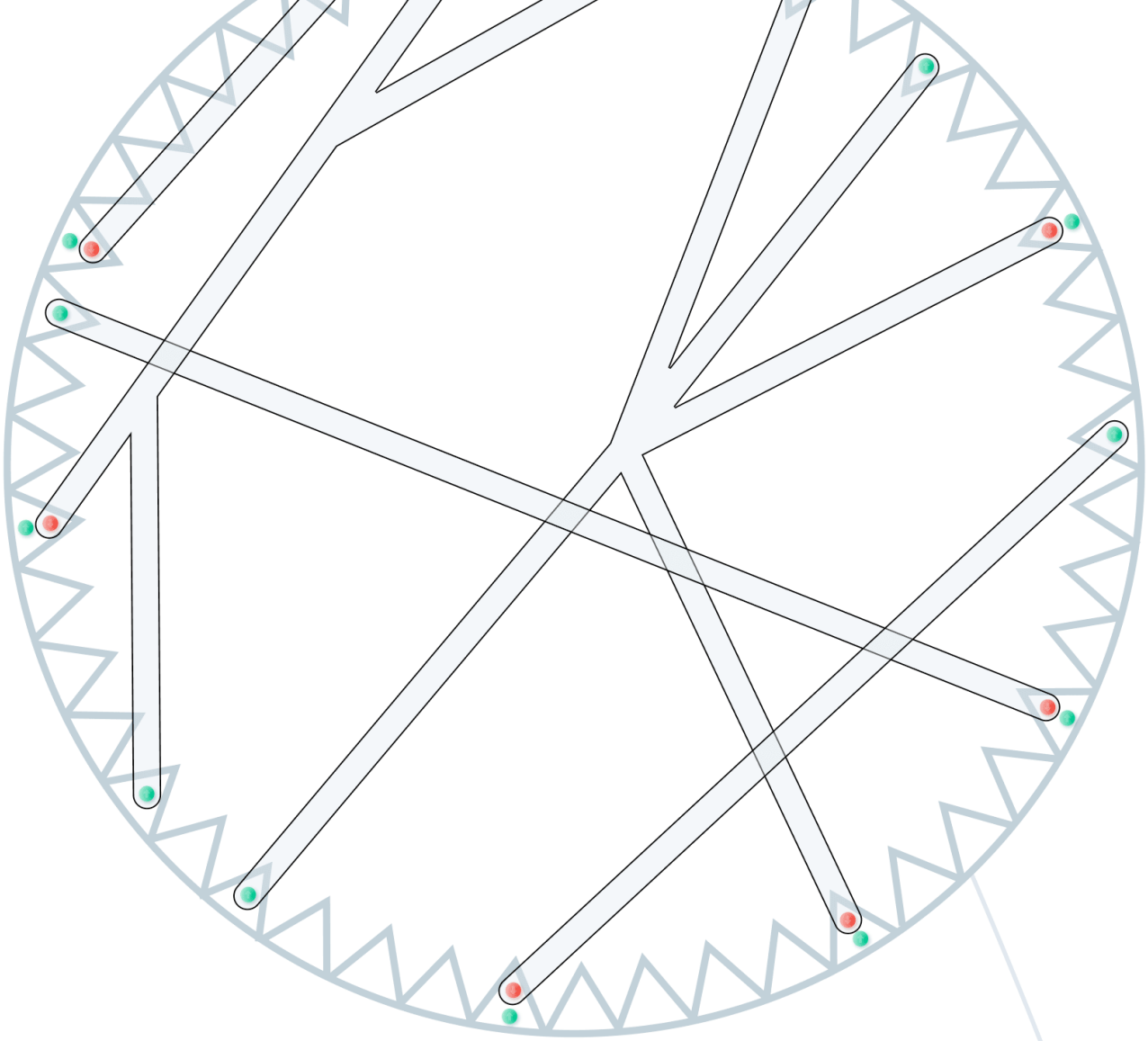
For example, the substate that represents Alice owning Token A and the substate that represents Bob owning Token B are two separate substates. This means that if Alice or Bob ever wanted to spend those tokens in future, those future transactions could be processed in parallel. It doesn't make logical sense to join them into one substate.

Because of the Radix Engine, nodes therefore only need to undertake consensus on the absolutely most minimal set of shards required.

If events are not related, they just get processed in parallel, across the entire shardspace, which is so large in size as to be practically infinite.

An almost infinite number of transactions can be processed across the global shardspace in parallel, so long as there are enough nodes. But where substates are related, Cerberus can compose these into atomic transactions.





As an example, the five transactions above are submitted to the network around the same time.

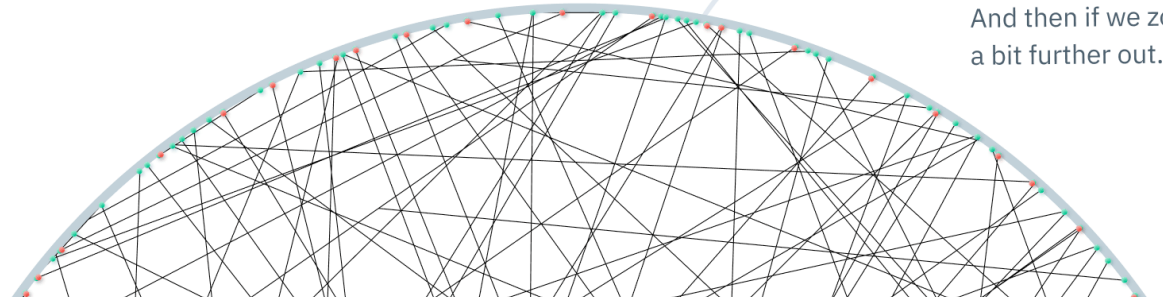
For each transaction, the Radix Engine has forced the transaction to specify the dependencies between substates. So within each transaction, events are ordered.

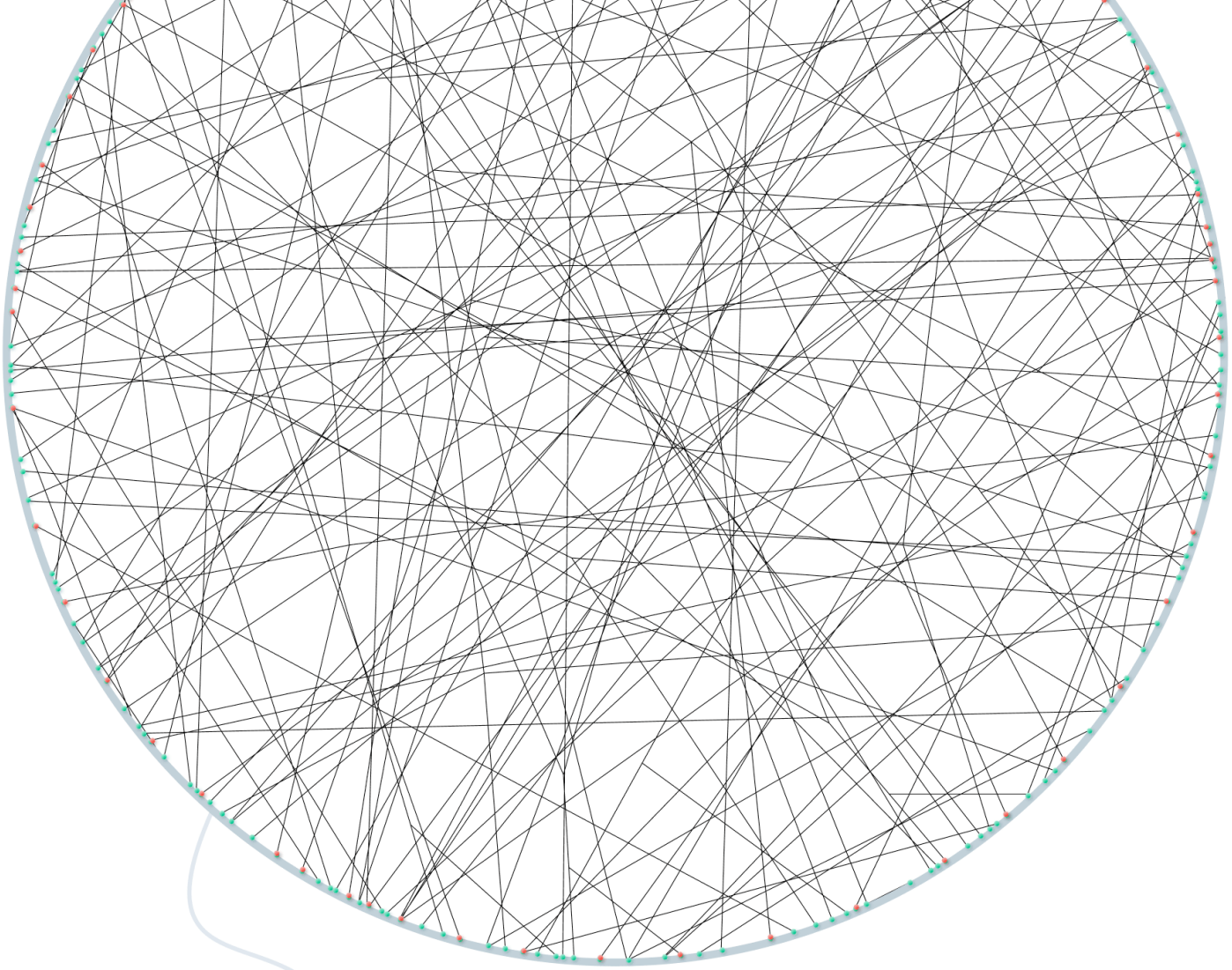
However, between them, the transactions are completely unrelated - think two different people buying coffees on opposite sides of the world. So at the transactional level, they can be processed safely in parallel.

As the shardspace effectively stretches out to infinity, to process more transactions, all you need to do is add more nodes so that you have more compute and network capability. This is what “linear scalability” means.

Additionally, if a single shard was included in two different transactions, those transactions would then be in conflict! The validator set serving that shard would see two different transactions that it was being asked to vote on, and would have to decide on only one of them. The other transaction would then be rejected.

And then if we zoom a bit further out...





...we can see there really isn't any limit to how many transactions can be processed across the practically infinite shardspace.

So let's now wrap up the final responsibility of a node - maintaining a record of transactions.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

while preserving atomic composability

Maintaining a Record of Transactions

To wrap up consensus and understand the data that nodes maintain, we revisit our example transaction.

Transaction (Tx)

Shard 1: *Shut down Alice's Token A substate*

Shard 2: *Bring up Bob's Token A substate [Index: 0]*

Shard 3: *Shut down Bob's Token B substate*

Shard 4: *Bring up Carol's Token B substate [Index: 1]*

Shard 5: *Shut down Carol's Token C substate*

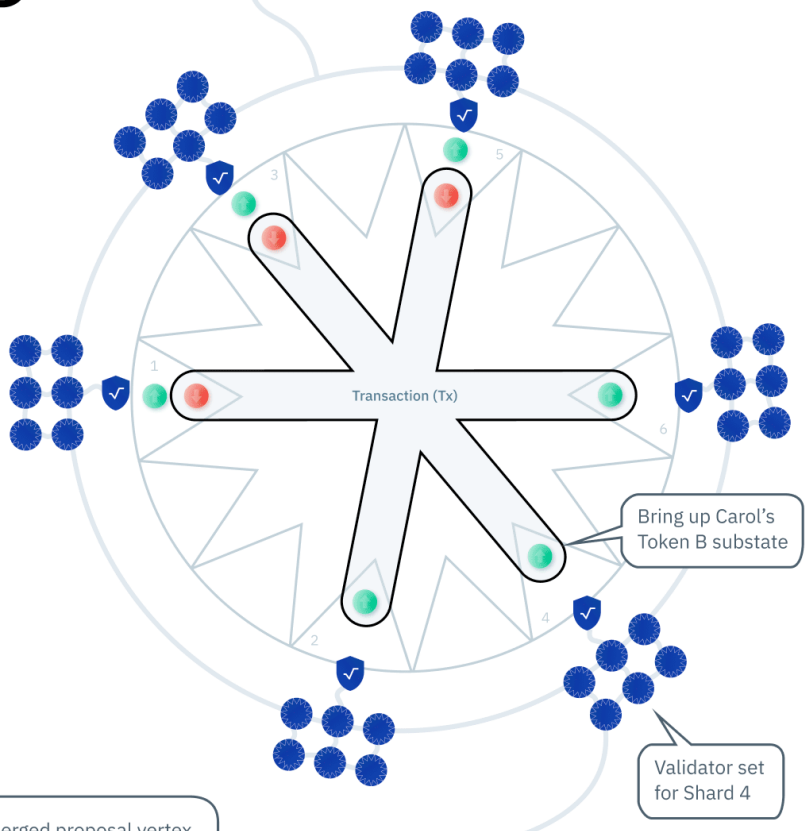
Shard 6: *Bring up Alice's Token C substate [Index: 2]*

Signed: *Alice, Bob, Carol*

Tx ID: ...979

We can see that the transaction contains all the details needed to create each substate in their respective shards...

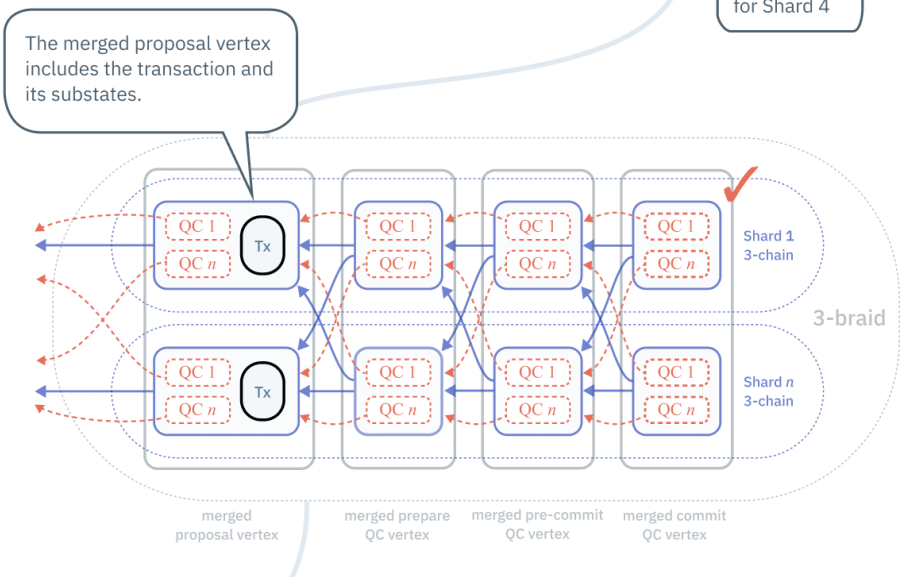
...and in order to write the transaction to the ledger, nodes need some way of coming to consensus between validator sets, across shards.



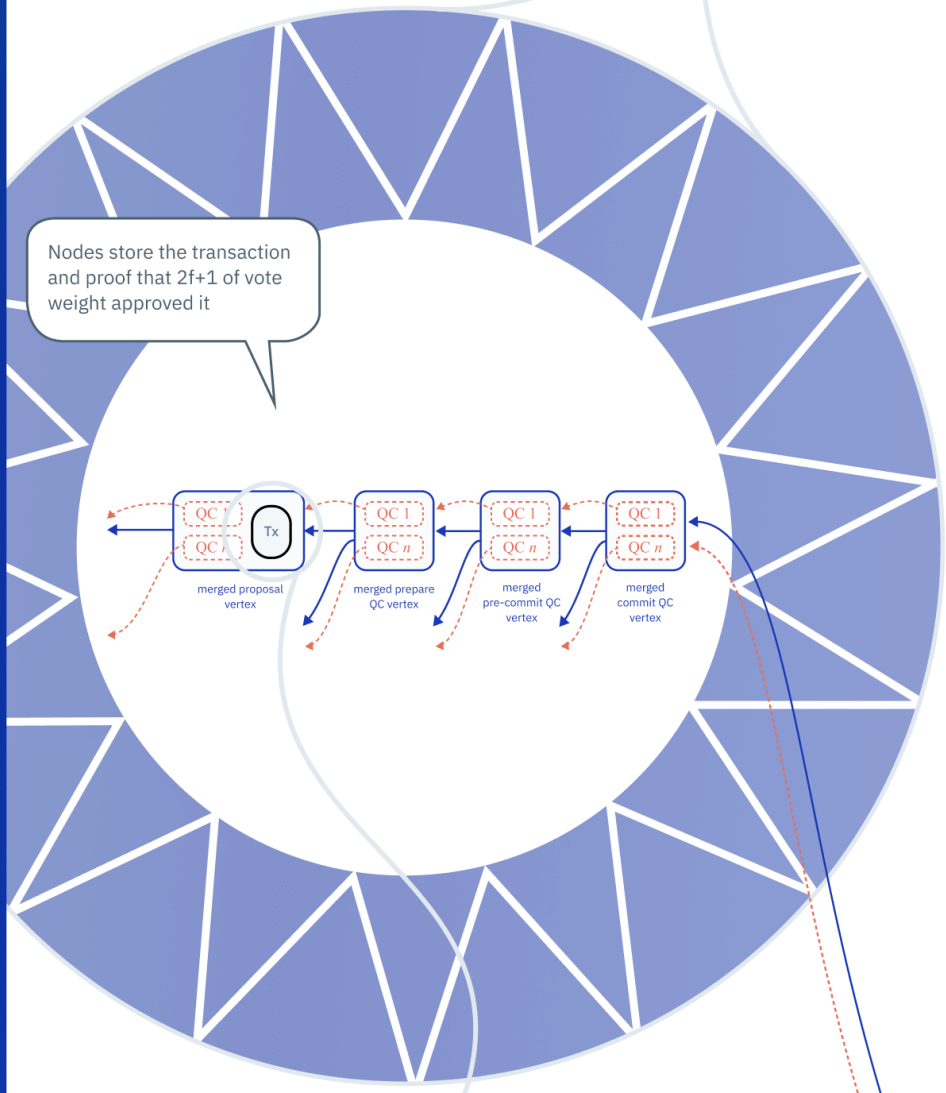
This is achieved through the unique "braiding" design of emergent Cerberus, where QCs and vertices for each shard are shared between all shards, and chained together with hashes.

Because of this, every node in every validator set has proof that 2f+1 of vote weight in every shard approved each phase of consensus.

This design uniquely allows



validator sets in Cerberus to come to consensus across shards - either all together, or not at all.



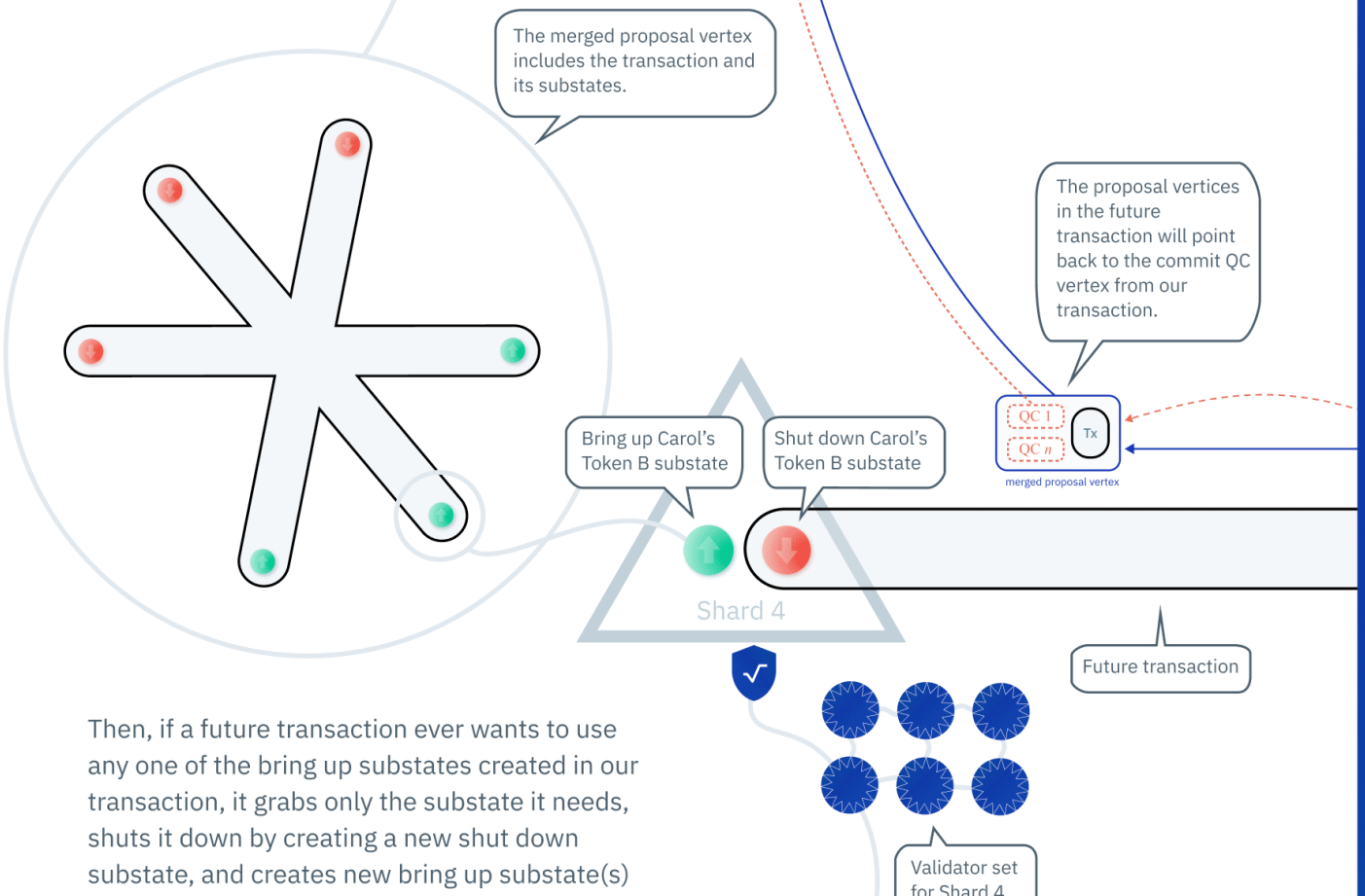
But so far we've only covered how nodes verify and vote on transactions. We haven't covered one of the most integral parts of their responsibilities - maintaining a record of transactions.

So what data do nodes have to keep, in order to fulfill this duty?

Nodes store a copy of the whole transaction, which includes all the substates across all relevant shards.

They also store proof that 2f+1 of vote weight approved the transaction across every shard, for every phase of consensus through the QCs in each vertex.

Each vertex is chained to the previous vertex via hashes - cryptographically tying every phase of consensus together.



Then, if a future transaction ever wants to use any one of the bring up substates created in our transaction, it grabs only the substate it needs, shuts it down by creating a new shut down substate, and creates new bring up substate(s)

in new shards.

Relevant validator sets then come to consensus on new proposal vertices that point back to the commit QC vertex from our transaction. The validator set serving Shard 4 depicted here has all the data it needs to verify the future transaction as it has a complete copy of our transaction, and a complete copy of the future transaction.

And the cycle continues...

So that's a wrap on consensus. Now just one last thing - Sybil Resistance Through Proof of Stake.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

v1.0 · June 2021

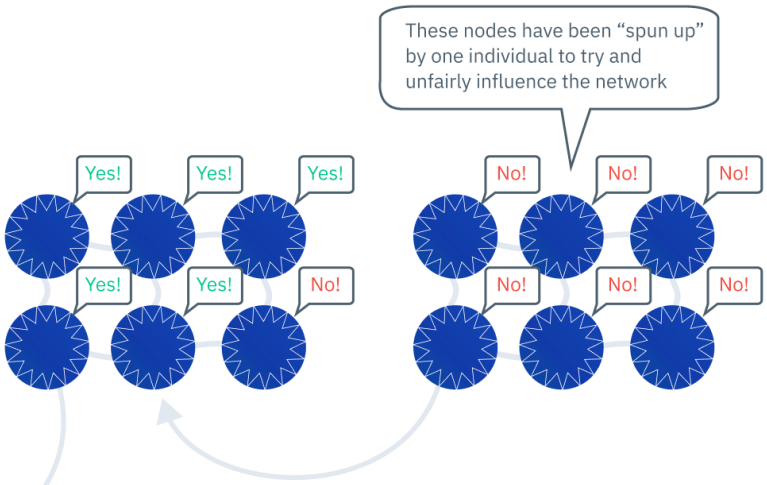
Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

Sybil Resistance Through Proof of Stake

We discussed before that nodes come to consensus to commit a transaction to the ledger by reaching a majority vote on it. But what if somebody were to create lots of nodes to get lots of votes? This is known as a Sybil attack.

To prevent this, public blockchains and DLTs need a smarter way to weight the votes.

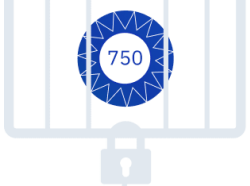
Radix uses a mechanism called "Proof of Stake" (PoS) to weight the votes of each node for Cerberus consensus.



In a PoS network, if a node wishes to participate in verifying

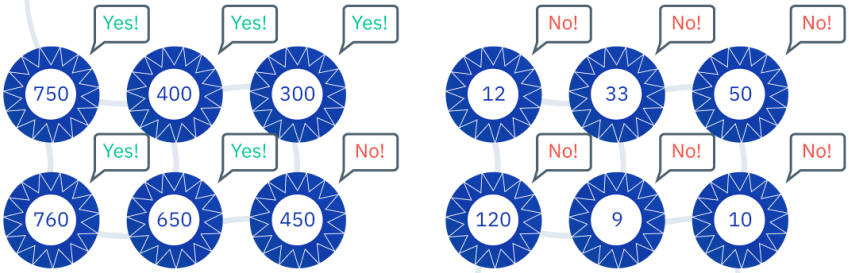
Radix's version of Proof of Stake is called Delegated Proof of Stake (DPoS)

transactions, it must put down a "stake" – that is, lock up some tokens and keep them locked as long as they want to verify transactions.



which means that token holders are incentivized to "delegate" their tokens to a node to earn a reward.

A node's voting strength is weighted proportional to the total amount of stake locked to it.



With votes weighted by stake, if someone wanted to execute a Sybil attack, it would be much more difficult, as they would have to spend significant resource purchasing stake in the network, and all they would be doing is harming a network they now have a significant stake in!

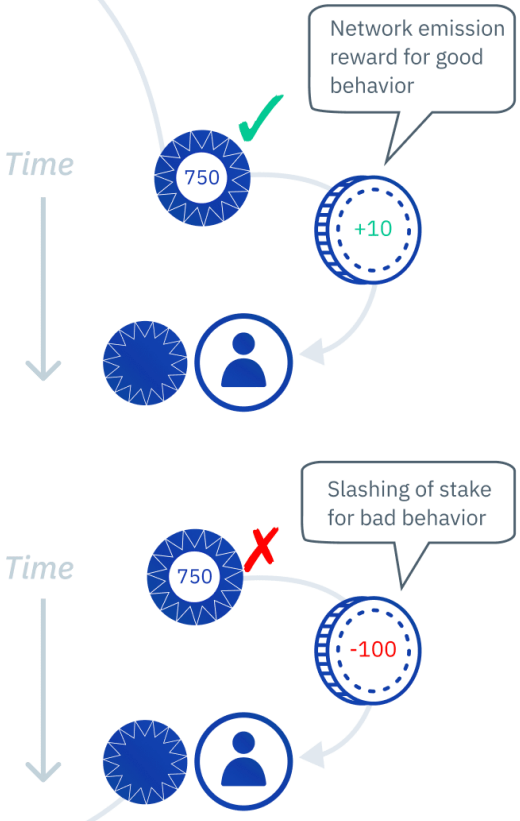
The Sybil attack depicted here would fail, as the malicious nodes don't have enough stake.

So why stake in the first place?

If nodes behave well, and validate transactions honestly, they and the token holders who delegate stake to them earn a reward that's proportional to the stake put in. This reward is called "network emission" and is similar to mining tokens on Bitcoin.

If a node is found to be provably malicious, then it will lose both the network emission reward, and some or all of the stake committed to it, incentivizing nodes to remain honest. Malicious behavior might be something like a "double spend": telling some nodes one vote and other nodes a different vote to try to convince the network to process two conflicting transactions.

While the network (majority of vote weight) will try to detect and punish malicious behavior, to successfully mount such an attack, an attacker would need to control greater than 33% of the total stake, making such attacks extremely expensive.



So that's it! Time to conclude.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.

Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

Conclusion

Before we wrap everything up, we go back to the very beginning:

> Cerberus - How Radix achieves infinite linear scalability while preserving atomic composability

To summarize, Cerberus' novel "braided" consensus design is what provides Radix the unique ability to compose transactions and settle them atomically across shards.

Cross-shard atomic composability, in turn, lets Cerberus take full advantage of partial ordering (as provided by the Radix Engine):

- If substates are unrelated, they can be processed in parallel across the vastness of the shardspace. The practically infinite shardspace allows Radix to process more transaction throughput just by adding more nodes - there will always be room for more!
- If substates are related, Radix Engine tells Cerberus they must be processed atomically in a single cross-shard round of consensus - thus preserving atomic composability.

So, Radix achieves practically infinite linear scalability *precisely because* cross-shard atomic composability is baked into the very fabric of every transaction. Atomic composability is not an afterthought to scalability - it is the key enabler!

And so it all comes back to braiding of consensus. Of all our technical advancements, *this* is Radix's most profound innovation. Braiding is the root of Radix's new paradigm in DLT design.

Unique Features

Practically Infinite Linear Scalability
Allows Radix to always scale to meet the needs of network usage – just like the internet, there is no ceiling

Cross-Shard Atomic Composability
Allows transactions to be composed across shards - either settling all together, or not at all - across the global ledger

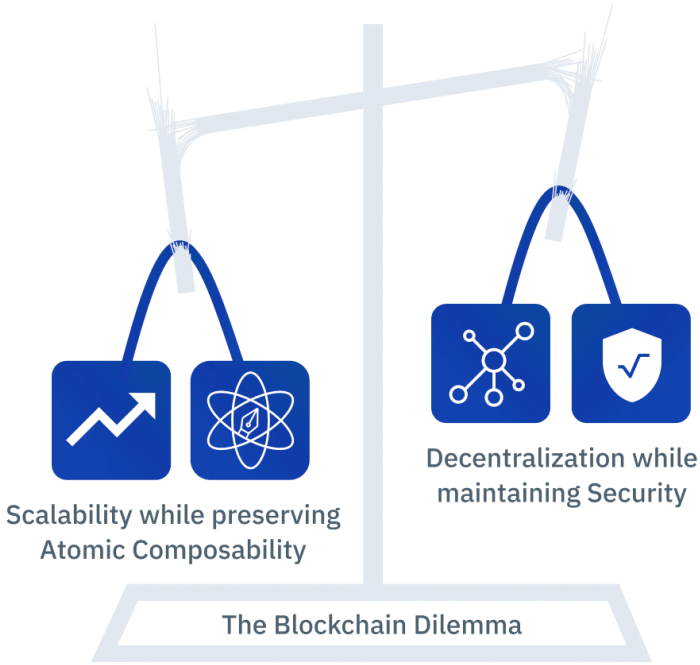
In final conclusion, we revisit the Blockchain Dilemma:

Scalability while preserving atomic composability

Radix offers practically infinite scalability while preserving atomic composability. No Layer 2 solutions will ever be necessary. Settlement finality can be achieved within 5 seconds because there is no need to wait to see if a longer chain emerges.

Decentralization while maintaining Security
Radix has no masternodes or coordinators; there are no relay, beacon, or runner chains. Radix exists as a single layer of programmable trust as all nodes are created equal. As there is no limit to how many nodes can validate on Radix, and that nodes can always be run on simple hardware, Radix is as decentralized as can be.

For security, Cerberus' unique braided consensus design has been publicly reviewed by UC Davis' ExpoLab consensus experts and others and shown to provide strong guarantees of safety and liveness. Radix uses Delegated Proof of Stake for Sybil protection and has the same resistance to Sybil attacks as any other DPoS protocol.



As a final thought, Radix rewrites the standard for what was previously thought impossible. Radix demonstrates that it *is* possible to achieve the scalability and atomic composability required to scale DeFi to billions of people *without* sacrificing decentralization and security.

The version of Cerberus described in this infographic series is scheduled to launch as part of the fully sharded Radix Xi'an release. Please visit www.radixdlt.com for details on the Radix roadmap.