Notoros: A Next Generation Framework for Distributed Ledger Applications
v1.0

Brendan Laiben,
Andrew Brick

March, 2021

Abstract

Notoros is a protocol for executing distributed applications in parallel using a decentralized ledger. It uses logical clock vectors to order resource accesses by transactions. The name Notoros is an amalgamation of the Latin words "nota", meaning note or record, and "hora", referencing "time", followed by "os" for operating system.

Notoros builds on many concepts already present within the distributed ledger industry as well as designs for general distributed applications. The protocol is inspired heavily from the works of Leslie Lamport[14], specifically the papers on the Bakery Algorithm[1] and the use of logical clocks in distributed systems[2]. By combining sharding, parallelization, and lazy execution, it provides the most complete set of distributed ledger features ever seen. The protocol is designed to allow any kind of decentralized application to be built using a common distributed ledger.

As currently implemented on a Cerberus[11] network, the protocol has proven to be capable of scaling the Ethereum Virtual Machine[4] to over 24,000 transactions per second, reaching parity with traditional payment processors. Additional progress in code optimization and message succinctness can allow the network to increase total throughput by over two magnitudes without protocol changes. The Notoros Protocol can also be recursively implemented as a Layer 2 solution to provide further scalability.

1 Introduction

Since the inception of the decentralized ledger industry, the community has sought a solution for the scalability of arbitrary distributed virtual machine models.

The Bitcoin Whitepaper represented a dramatic leap forward for the legitimacy and exchange of disintermediated digital assets. The Ethereum protocol extended this concept by leveraging Bitcoin's distributed consensus protocol to create a trustless distributed ledger complete with a Turing-complete virtual machine for executing smart contracts. These contracts were used to encode arbitrary state transition functions enabling a wide variety of applications like DAOs, Non-Fungible Tokens, and decentralized financial applications.

While these innovations have already left a multi-trillion dollar mark on the global economy, they remain incapable of scaling, narrowing their use cases and leading to egregiously expensive operational costs. State-channels, side-chains, relay networks, and cryptographic rollups have all been attempted as ways to mitigate current scalability issues, but each of these solutions retains major limitations with regard to security, smart contract composability, cost-effectiveness, or decentralization.

To resolve these obstacles and reveal entirely new technological and business opportunities, we present the Notoros Protocol: a generic framework for applications on distributed ledger networks providing a parallelizable resource allocation and sharing mechanism. Notoros maintains the ordering of requests made to the network using a distributed ledger version of Lamport's bakery algorithm. This allows virtual machines to be deployed to a UTXO-based consensus mechanism, where their state can be sharded down to the bit level.

2 Previous Work

Notoros builds on many concepts already present within the distributed ledger industry as well as designs for general distributed applications. All of them attempted to solve a specific problem, and the solutions that came from them are incorporated within Notoros.

2.1 Transaction Ordering Systems

There have been many attempts to create robust transaction ordering systems within the distributed ledger industry. There are already hundreds of consensus mechanisms to choose from, and most of them are small variations on a theme. Looking beyond individual consensus mechanisms, there are several core transaction models that are prevalent within the industry.

2.1.1 Unspent Transaction Output Model

In the Unspent Transaction Output (UTXO) Model, a transaction consumes a collection of states from previous transactions as inputs and transforms those to new outputs for later consumption. Each output can only be consumed once, ensuring atomicity of updates.

Bitcoin[3] uses the UTXO model to track fungible token transfers, creating a linked series of transactions that traces each back to the ledger's creation. Within the Bitcoin ledger, each transaction output has a unique identifier, an owner address, a number of tokens associated with the output, and a flag indicating if the output has been consumed by another transaction or not.

The sum of tokens in a transaction's outputs must be equal to the sum of tokens in its inputs, except for special cases related to consensus and mining. Additionally, all of the inputs must be owned by the transaction's sender. The UTXO model has been adopted by many distributed ledgers, but few of those implementations allowed any programmability. The primary use for UTXO ledgers remains currency tracking.

2.1.2 Atom Model

The Atom Model[11] provides a way to extend the functionality of UTXOs to allow multiple kinds of token systems to be tracked in the same ledger.

The base inputs and outputs within the Atom Model are Particles, which have a unique identifier used to track them. Custom classes of Particles can specify the transitions needed for them to be accepted by the ledger. To allow for multiple actions within a single Atom, Particles are organized into Particle Groups, and all of the Particle transitions within a Particle Group must execute simultaneously to be valid. Particle Groups are then collected into Atoms, where each Particle Group must execute properly in sequence for the Atom to be valid. Finally, Atoms are passed through the network by consensus.

This model allows more fine-grained control over high-level ledger operations than typical UTXO transactions alone.

2.1.3 Account Nonce Ordering Model

The UTXO models is limited by its inability to perform complex logic within the transactions due to the restriction on inputs. As an alternative, the Global State Model uses a virtual machine to execute arbitrary transaction inputs. In the Account Nonce Ordering Model, each account submits transactions in the order of their respective account nonces, and those transactions must be executed in order of ascending nonce. Inter-account ordering is typically informed by block production rather than any specific properties on the transactions.

2.2 Distributed System Scaling

The design concepts and hurdles for distributed systems are not confined to the blockchain space; they have been around since the first computers. Most of the distributed ledger industry relies on papers and technology from before the dot-com rush. The trustlessness features of these new networks that allow them to be decentralized are the primary difference from the pre-Internet networks.

Distributed systems allow hardware to be horizontally scaled using many low-end devices rather than one mainframe device. Architects can provide multiple configurations of the system so companies can upgrade their systems on an as-needed basis and reduce up-front infrastructure costs. Distributed systems become most cost effective when the amount of data being processed or stored by the network is maximized per device and the cost of devices is minimized.

2.2.1 Sharding

One of the ways that distributed systems reduce device costs is by sharding the network. When a network is sharded, only activity pertinent to a device is recorded and processed by that device. All non-relevant activity is ignored or, ideally, not even delivered to the device. This helps reduce network messaging load, storage costs, processor requirements, and power consumption.

Modern database systems use sharding to provide massive amounts of storage capabilities with reduced query and lookup times. Popular examples include MongoDB[15] and Redis[17].

Sharding is a highly desired feature in decentralized systems to support higher levels of transaction throughput. Sharding was not considered for the original designs of Bitcoin and Ethereum, and few blockchains have been able to implement a trustless sharding protocol that is able to scale effectively. The Radix network protocol includes sharding as a design requirement, allowing it to reach much higher levels of throughput than contemporary decentralized ledgers. Optimized consensus mechanisms around sharding trustless networks is still an area of active research since the use of the shards highly influences protocol design.

2.2.2 Parallelization

Execution parallelization is one of the most important features in today's desktop operating systems and programs. Parallelization allows applications to run on multiple concurrent processor threads, greatly reducing computation time. It also allows operating systems to control multiple applications at the same time, and distributed applications to use shared

resources without worrying about conflicts. Within distributed systems, many of Leslie Lamport's papers[14] are foundational works.

A variety of approaches to parallelizing trustless networks have been developed recently. Polkadot[8] and Skale[13] use a collection of side-chains to provide parallelization of environments. Optimism[16] uses optimistic rollups to allow off-chain computation to parallelize virtual machine execution. Cerberus[11] relies on Lamport's logical clock vectors to parallelize transaction validation. Most of these concepts are complimentary, not exclusionary, since they can be applied throughout the decentralized network stack.

2.2.3 Lazy Execution

In contrast to parallelization, which scales systems by allowing many instructions to execute at the same time, lazy execution scales systems by doing only the minimal amount of processing needed to determine the outcomes at a later time. This can be thought of as a queueing system, where instructions can be accepted into the queue without needing to be immediately processed. Lazy execution is widely used by applications dealing with databases where processing costs can be spread out during delays from asynchronous processes.

Lazy execution is a powerful concept within the distributed ledger industry because all network operations are asynchronous. However, lazy execution prevents transactions from being directly validated by Layer 1 consensus beyond simple formatting. This can be problematic for ledgers that use Layer 1 cryptocurrencies to record fees from smart contract execution, since the transaction's costs are not known up front. LazyLedger[10] uses lazy execution as part of its design to provide the minimal boilerplate infrastructure needed for decentralized networks.

3 Notoros Protocol

The Notoros Protocol is designed to scale the throughput of distributed virtual machines and decentralized applications using a trustless ledger. By combining sharding, parallelization, and lazy execution, it provides the most complete set of distributed ledger features ever seen. The protocol was designed to allow any kind of decentralized application to be built using a common ledger database.

The protocol is inspired heavily from the works of Leslie Lamport[14], specifically the papers on the Bakery Algorithm[1] and logical clocks[2]. It also draws from Dan Hughes's work on Tempo[12], a precursor to Cerberus[11], which relied on Lamport's works to achieve high throughput distributed consensus using the Atom Model. Like LazyLedger[10], it realizes that decentralized ledgers are simple recording tools and focuses on providing only the baseline functionality needed for distributed applications.

The Notoros Protocol on a Cerberus network is capable of scaling the Ethereum Virtual Machine[4] to over 24,000 transactions per second using today's technology. This throughput milestone is widely seen by the industry as the requirement for real mass adoption since it represents parity with traditional, mainstream payment providers. Additional progress in code optimization and message succinctness can allow the network to increase total throughput by over two magnitudes without protocol changes. The Notoros Protocol can also be recursively implemented as a Layer 2 solution to provide further scalability.

3.1 Notoros Components

The core design objective of Notoros is to allow any type of application to coordinate resource usage and chronologically order activity via a distributed ledger. It can be thought of as an abstract access control mechanism, and is simple to implement with few moving parts.

3.1.1 Messages

Transactions may include one or multiple messages to record in the ledger. Because the main purpose of the protocol is to determine a proper ordering of transactions, Notoros has little concern for the contents of the messages. It's intended that these messages be used as inputs to a virtual machine, like a stack of punchcards, though application designers can use them for any purpose that fits their needs. [1]

3.1.2 Signatures

Transactions may include an array of **signatures** to provide a proof of origin and implement various permissions, such as token transfers.

[1] While it's impossible to prevent *all* bad design practices, its worth noting here that application architects SHOULD NOT use the ledger for permanent storage of their application state. Instead, architects SHOULD use the ledger to record state proofs and other kinds of compact zero knowledge primitives that minimize the transmission and storage costs of the transaction. In controlled situations, such as private or consortium networks, this allows the network to push older transaction data to cold storage without impacting application availability.

3.1.3 Resources

Transactions specify a set of **resources** to operate on. Resources can be arbitrarily identified by applications, along with their state within the application. These definitions do not matter to Notoros, only the resource's unique identifier (usually in the form of a hash). The application state can be lazily computed by its users after consensus has determined the ordering of transactions.

Resources may be owned, preventing transactions from accessing them without the appropriate signatures. The determination of which resources are owned by which accounts is specific to the each implementation of the various consensus algorithms. Some networks may not allow for any owned resources, while others may only support owned resources.

Resource identifiers can be shared between applications, so each application must do some minimal sanitation checking on inputs from transactions to determine if they are applicable to the application's environment.

For example, an application may identify resources as belonging to Internet of Things message channels. In this case, each resource may be defined as either an input or output channel for a given device. Owned resources can be used to indicate outputs, enforcing that transactions have the device's signature. Unowned slots may represent the device's input channel, allowing the network to route transactions (and thereby messages) to a specific shard maintained by the device.

3.1.4 State Counters

Lamport's Bakery Algorithm[1] uses logical clocks to share resources in a distributed system. Logical clocks keep track of the number of events in a computer system. [2] The name of the Bakery Algorithm comes from the take-a-number queuing concept used in bakeries and delis, where customers receive a ticket with a number upon entry into the store and the lowest number is served first. The number on those tickets is analagous to a logical clock counter. Each time a resource is updated, its logical clock increases by one. Requests for that resource can be structured to allow concurrent reads between updates by referencing specific logical clock values.

Lamport's paper "Time, Clocks, and the Ordering of Events in a Distributed System"[2] laid the groundwork for Notoros by using logical clock vectors to establish an ordering of events. Logical clock vectors are groups of logical clock values corresponding to a set of resources.

Notoros uses logical clock vectors to order resource accesses by transactions. A resource's logical clock value, or **counter**, is updated when a transaction updates the resource. Combining a resource identifier and its logical clock counter exactly specifies a state in the Notoros ledger. We can think of this state identifier in the simple format **resource@counter**.

Each time a transaction accesses a resource it must also reference the transaction that last updated the clock value, so applications can easily query the ledger for any transactions that reference a given resource. This transaction chaining is provided by both the UTXO and Atom Models.

[2]Models like the Stellar Consensus Procotol validators use consensus about a logical clock value updated with every account's sent transaction in the ledger.[6]

3.1.5 Notoros Policies

For each state identifier (**resource@counter**) referenced in a transaction, the transaction must also specify an operation on the identified state. These operations are called **transition policies**.

In the starting case where a resource entry has not been initiliazed, a transaction may do so by refencing the *null* clock value and updating the new clock value to 0, representing the `CREATE` operation. Once a resource is initialized, transactions can either increment the counter by one or maintain the current value. These operations represent `WRITE` and `READ` changes within applications, respectively.

Note that these operations represent three of the four CRUD operations that can be performed for persistent storage. Because the nature of distributed ledgers is to be immutable, and thus retain all data, there is no delete operation. Applications may prune their state as needed, however, since the ledger will always be able to bootstrap it from history.

The general format of a policy can be thought of as **stateIdentifier:operation**, which expands to **resource@counter:operation**. The operation can be a simple binary value, with 1 representing a non-READ operation based on the counter value.[3].

The formal definition of these transitions is outlined in Table 1 and diagrammed in Figure 1.

Action (n)	Prior Counter	Updated Counter
Create State	*null*	0
Read State	$C(n-1)$	$C(n-1)$
Write State	$C(n-1)$	$C(n-1)+1$

Table 1: Policy Transitions

[3]The binary counting property of Notoros policies lends itself well to zero knowledge proof systems. Additional research is needed to identify opportunities in this area

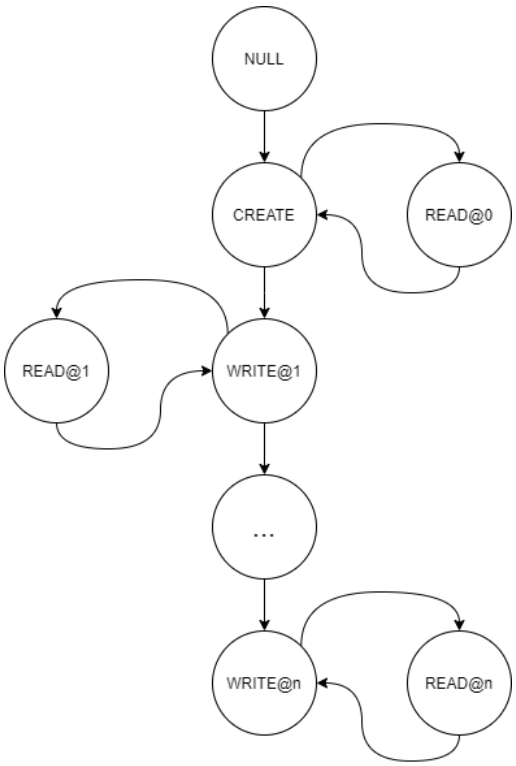

Figure 1: Finite State Machine Transitions in Notoros

3.1.6 Example Ledger States

The Notoros ledger is simple to internally maintain. Figure 2 shows an abstract representation of how the ledger records transaction history against resources. In the example, the ledger records four transactions against a random resource with an initially undefined logical clock counter. The first transaction creates the 0 state. The next two transactions reference the state in READ operations, which are parallelized. The last transaction updates the logical clock counter to 1, creating the newest head state.

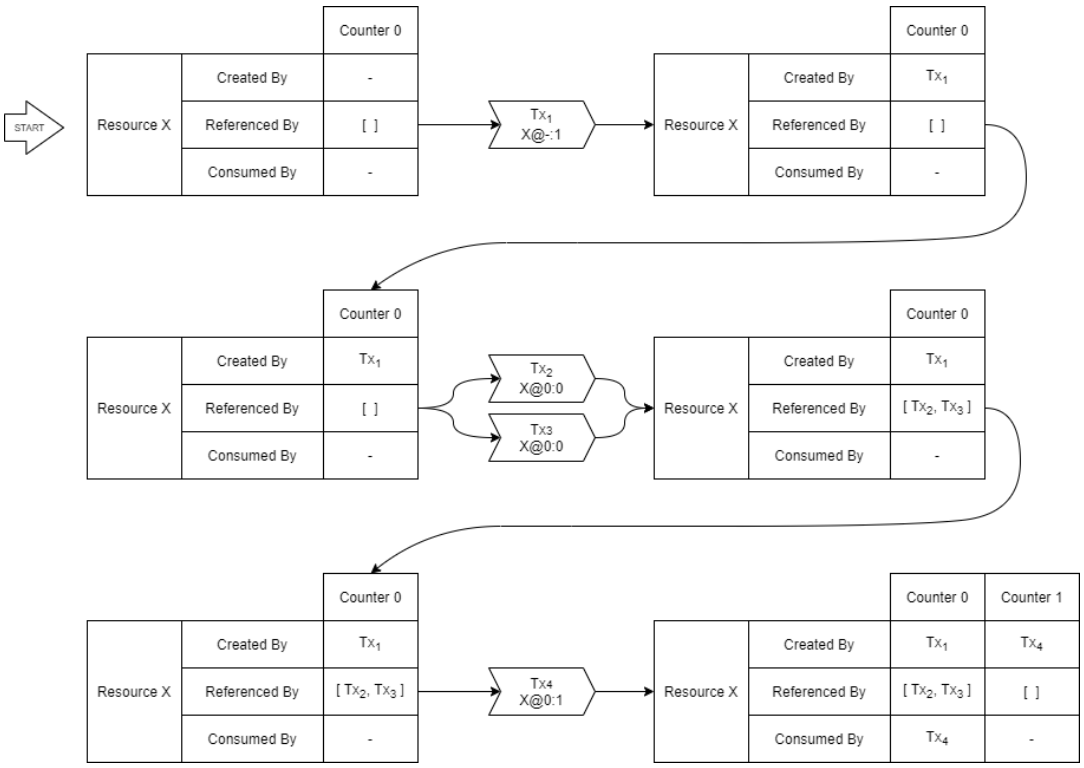

Figure 2: Abstract Ledger Example

3.2 Network Scaling

With Notoros, distributed ledger networks can achieve a larger scale than ever before. The design of the system allows it be highly parallelized and natively sharded.

3.2.1 Parallelization

Transactions can be trivially parallelized if they do not have any overlapping resources required in their policies. If they do have overlaps, they can still be trivially parallelized if those overlaps are `READ` operations at the same clock counter.

Policies with overlapping resources at the same counter are executed by the following rules:

1. `CREATE` policies are applied first. Then,
2. Any `READ` policies are applied.
3. `WRITE` policies are applied last.

Figure 3 shows how a group of transactions can be ordered for parallelization based on their policies. Each transaction is linked to a parent logical clock update. Mapping dependencies reveals the ordering relationship. Transactions within a execution group can be safely processed in parallel without conflicts.

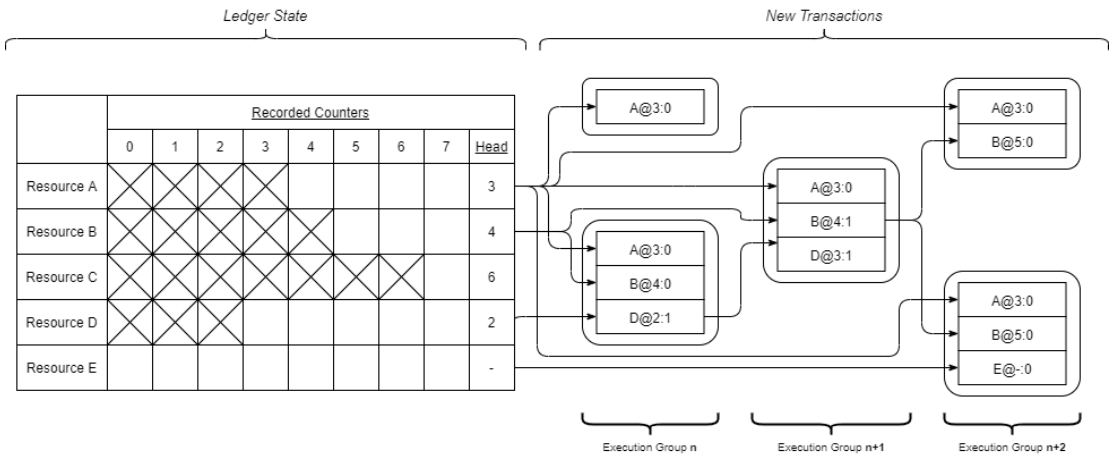

Figure 3: Example: Determining Parallelization

3.2.2 Sharding

Each resource in Notoros can be logically mapped to a shard in a decentralized network, either in a one-to-one or many-to-one fashion. Within the nodes of a distributed ledger network operating with Notoros, there is little difference between shards apart from their historic usage rates and the amount of data they store. When a transaction has policies that cross shards, all impacted shards receive and validate the transaction [4].

In the example shown in Figure 4, seven transactions are recorded against two shards. Each shard has an initial transction followed by a write each for the first four transactions. The fifth transaction refrences resources in both shards, so each one must validate the transaction before it is accepted.

[4]Depending on the consensus model, cross-shard validation may require shards to validate the other's history. In this case, additional considerations should be made to account for this extra load, such as increased transaction fees or rate limits

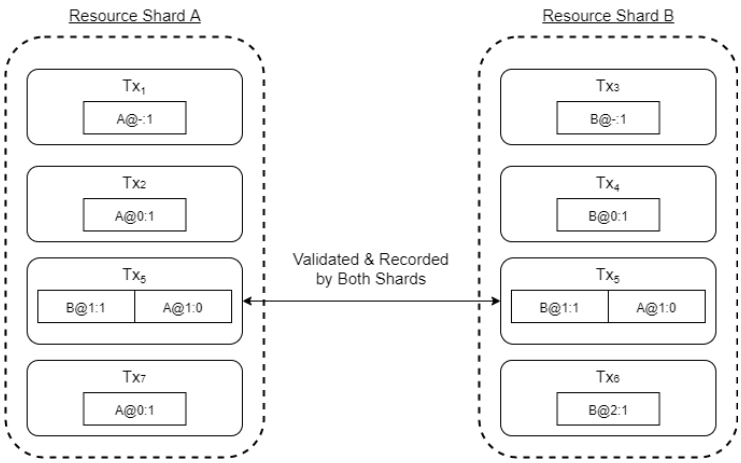

Figure 4: Example: Abstract Shard Reconciliation

3.3 Privacy Mechanisms

A significant portion of the distributed ledger industry is highly concerned with maintaining privacy while ensuring trustlessness. Notoros has many mechanisms to help achieve this goal.

3.3.1 Pseudoanonymity

Since Notoros is intended for use in a decentralized system, it uses pseudo-anonymity provided by public key cryptography to provide a layer of privacy for users. This is the same level of privacy provided by Bitcoin.

3.3.2 Encrypted Messages

Notoros does not care about the content of messages, so they can be encrypted or otherwise obfuscated to prevent public view. Those messages can then reference ephemeral data (such as values in a local database) for inputs, ensuring that only the portion of nodes that can decode the messages and retrieve the information in time are able to determine the state.

3.3.3 Chaotic Usage

With enough usage, patterns in resource activity will emerge that expose information about the applications using those resources. This prevents a problem for architectes desiring complete privacy. To compensate for this, applications can implement deterministic but seemingly chaotic algorithms for distributing their activity across pseudorandom resources. For example, a directed acyclic graph could be used to define the location resources in the application.

Another option is to pseudorandomly inject cryptographic salt, making the data much harder to read. This could be combined with red herring ”no-op” transactions that appear to be valid to outside users but do not represent real activity within the application.

3.4 Application Design Considerations

3.4.1 Virtual Machine Applications

The execution boundaries of a transaction's message within an application's virtual machine are defined by the create/read/write policies in the transaction. As a requirement, the virtual machine must be configured such that if the execution of the instruction set exceeds those boundaries, the entire transaction's virtual machine state changes are reverted and the next transaction is processed using the unmodified state.

3.4.2 Variable Resource Definitions

The granularity of an application's chosen definition of resources can greatly influence its architecture and design. While increased granularity provides more throughput through parallelization and greater security through decentralization, it comes with a literal cost in the form of increased transaction fees to include the additional policies in the transaction.

Intelligent mappings of application resources to a Notoros resource can provide many different dynamics for application usage. For example, an architect may want only a specific portion of the network to be used by the application, and thus limit the scope of their application's resource definitions to resources maintained in that section. This may also be used to partition multiple instances of the same application across the network.

3.4.3 Shard Allocations

Applications can choose resources definitions that distribute those definitions across various portions of the shard space. Figure 5 shows how various definitions can impact network usage. For instance, at either extreme are the options for using the full shard space or only a single shard. There are effectively infinite ways to configure an application for shardability with Notoros.

Additionally, there are opportunities for overlapping virtual machines that use the same resource but in separated contexts. A "Meta VM" can then be created that creates a combined state from the results of transactions from both applications, presenting opportunities for inter-VM communication strategies.

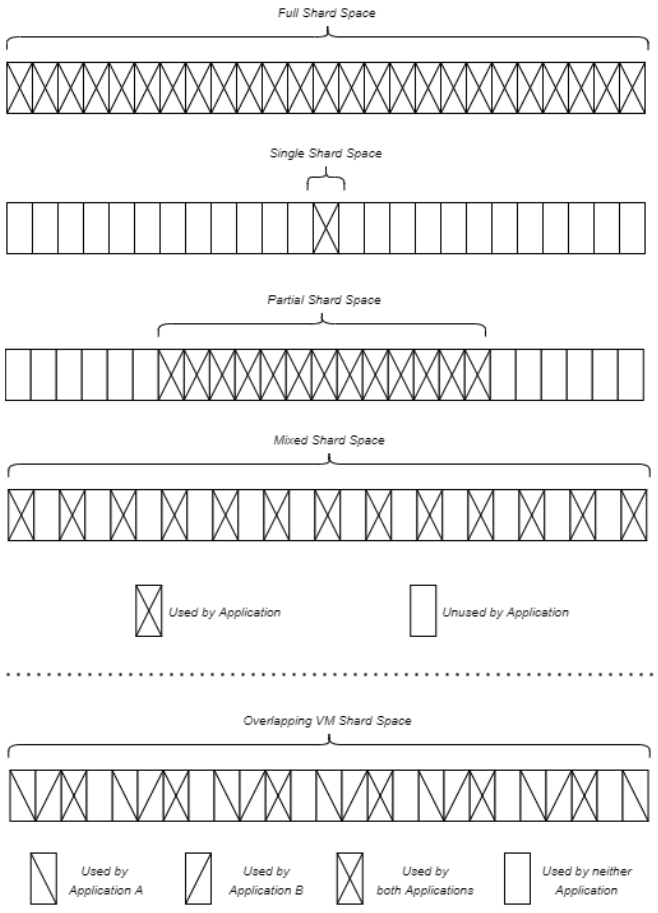

Figure 5: Application Shard Allocation Strategies

4 Implementation on Radix

Notoros is designed to be flexible and adaptable to many networks. The first public network it will be implemented on is Radix's mainnet. Radix uses the Atom Model to shard and process UTXO-style transactions via the Cerberus[11] consensus algorithm.

Each Radix Particle executes a transition function within the Radix Finite State Machine. The validity of the transition and its effects are determined by the particle's most derived class definition.

Particle Groups in Radix are intended to represent a single action by an application. With this in mind, each Notoros Transaction in Radix is defined by the contents of a given Particle Group: Particle transitions specify **policies**, and Radix Messages specify **messages**. Finally, the signatures on the Radix transaction can be directly mapped to the **signatures** array in each Notoros Transaction.

Since each Radix transaction may contain multiple Particle Groups, they are capable of specifying multiple Notoros Transactions. This is convenient for application designers that want to call separate VM implementations at the same time or execute a series of transactions.

4.1 Notoros Messages and Signatures

Radix allows the use of multiple Radix Message Particles in a Particle Group. These messages may contain arbitrary byte data and therefore meet the requirements for passing Notoros Messages.

Radix also allows multiple signatures to be applied to a transaction, directly meeting the requirements for Notoros again.

4.2 Particle Spin

In Radix's Atom Model, UTXO functionality is provided via a particle's **spin** property. A Radix Particle's **spin** can be one of the following:

- `null`: The particle does not exist
- `UP`: The particle has been created but not consumed (as UTXO-style output)
- `DOWN`: The particle has been consumed (as UTXO-style input)

4.3 Resource Identifiers

To fully maximize the capabilities of Radix and allow for future extensibility, resources are identified within the ledger using a combination of properties:

- Address (256 bits): Identifies the specific shard that tracks the resource
- Slot (256 bits): Identifies a subsection of the shard that stores the resource history and state.

- Type (8 bits): Identifies a variation on the resource, with odd values being owned by the private key corresponding to the resource's address.

4.4 Notoros Particles

Radix has the capability to use user-defined Particle definitions in its network. User-defined Radix Particles inherit the **spin** property.

Notoros Particles are extensions of Radix Particles that are used to enforce Notoros transition policies on the Radix Ledger. They are uniquely identified in Radix by the hash of their resource identifier and counter (among other properties as revised): $id = H(address : slot : type : counter)$. This makes finding conflicts and collisions trivial, since it is simply a matter of checking the **spin** state of the particle's **id** in the Radix Ledger.

Table 4.4 shows how the Notoros components are mapped to Radix Particles.

- Create Particles
  - Initializes a resource's **counter** from *null* to 0
  - Reference particle must be a Radix Void Particle
  - Output particle is a Notoros Particle with `spin=UP`
- Read Particles
  - Enforce that the resource's logical clock is at the specified **counter**
  - Reference particle must be a Notoros Create Particle or Notoros Write Particle with the same resource ID and **counter** and `spin=UP`
  - Output particle is the originally referenced particle (indicating no changes to resource) with `spin=UP`
- Write Particles
  - Enforce that the resource's logical clock is at the specified **counter** and then increase the resource's clock by 1
  - Reference particle must be a Notoros Create Particle or Notoros Write Particle with the same resource ID and ascodecounter and `spin=UP`
  - The reference particle is consumed (`spin=DOWN` , indicating the resource has been potentially modified)
  - Output particle is a new Notoros Write Particle at the specified resources with the updated **counter** value and `spin=UP` (providing a new reference for future operations).

4.5 Notoros Policies

A single Notoros policy is the combination of an (potentially *null*) input Particle and an output Particle. That means each policy also contains a unique **id** from the output Particle and a reference (**ref**) to the policy that was previously applied to the resource via the input Particle. Table 2 shows the full data available from a policy.

Notoros Policy		
Property	Type	Description
address	uint256	Location of resource being operated on (i.e. shard)
slot	uint256	ID of resource being operated on (vm-dependent)
type	uint8	Variation of resource
counter	uint64	Current logical clock value for resource variation
action	enum	Ledger operation to perform: ”create”, ”read”, ”write”
id	uint256	Policy ID (i.e. output hash)
ref	uint256	Parent Policy ID (input hash or nulled if action=”create”)

Table 2: Notoros Policy Definitions in Radix

4.5.1 Address

The address is the primary shard that contains that record. The shard size of a transaction is determined exclusively by the width of the set of addresses that are being accessed.

4.5.2 Slot

The slot of a policy is the sector of state within a shard/address that is being operated on. Slots can represent message channels, memory registers, and other shared resource types.

4.5.3 Access Type

The access type allows transactions to specify up to 256 (2^8, or one byte's worth) different variations of resource accesses. This can be used to stack applications within the same shard space or specify different types of storage mechanisms, depending on application requirements.

Access types are split into two categories: public and witnessed. Access types that are even numbers (including 0) are public: they may be used by any account in a transaction. Access types that are odd are witnessed: they can only be used when the transaction is signed by the private key corresponding to that address. This system allows developers to create anti-DoS mechanisms in their apps as well as multi-signature flows.

4.6 Messages

Messages within a transaction designate a virtual machine specification, an instruction set to execute, and the original sender of the message. Messages are given a unique identifier based on the hash of their contents.

4.6.1 Virtual Machine Specification

The **application** field within a message specifies which virtual machine should be used to execute the instruction set. It is recommended that this field be constructed using the hash

Notoros Radix Message		
Property	Type	Description
sender	uint256	Indicates origin of tx. Sender must sign tx.
application	bytes	The VM application targeted: 8-1024 bytes
body	bytes	Command to be executed by the VM application
nonce	bytes	Random bytes to generate unique ID: 0-32 bytes
id	uint256	Message ID (i.e. hash)

Table 3: Notoros Message Definition in Radix

of any genesis and/or network parameters to provide a traceable identifier for all users and exclude non-conforming machines.

4.6.2 Instruction Set

The **body** field within the message contains the instruction set to be executed in the VM (e.g. EVM bytecode). This field can also be used to transmit simple messages via the network if the message must be continuously available or guaranteed delivered.

4.6.3 Sender

The **sender** field is the official origin of the message. The senders of every message in a transaction must sign the transaction with their key and add it to the 'signatures' array for the transaction to be valid.

One naive way of creating multi-signature transactions is to have the same message body in messages from multiple parties contained in a single transaction to enforce that all parties must sign the transaction.

5 EVM Implementations with Notoros

Notoros was designed from the start to enable any form of application to be operated using inputs ordered via a distributed ledger. There are several virtual machines that use distributed ledgers for their inputs, with Ethereum being the most widely known.

Notoros is implementing a highly sharded Ethereum Virtual Machine[5] (EVM) on the Radix public network. Ethereum has the largest number of developers and widest array of tools available of any ecosystem in the blockchain industry. There are numerous implementations of Ethereum, each with their own unique aspects designed for the distributed applications running on them. The specification outlined in this paper is intended to provide a guideline rather than dictate a singular implementation: many of the parameters provided to the Ethereum VM can be modified to create innovative operating models and provide a level of obfuscation for privacy purposes.

Due to the different nature of the Ethereum environment compared to the Notoros environment in relation to consensus and sharding, some properties are not directly translatable. However, variations of the EVM implementation can specify whatever definitions are best suited for their users' needs.

5.1 Address Translations for Radix

One application-specific definition is the translations between Ethereum Addresses and resources. In the Radix network implementation, resources are mapped to shards by Radix Addresses. Since Ethereum Addresses are based on the hashes of public keys and do not take a full 256 bits like Radix Addresses, each Notoros Policy's **address** field must be translated as allowable. The following solution is proposed for simplicity:

- Translation from Ethereum address to Notoros Policy **address**
    1. Use the Ethereum address as a seed for a new private/public key pair. The public key's address is the corresponding **address**
        - *Note: This makes the private/public key pair for a given Ethereum address effectively impossible to find, barring quantum solutions*
    2. The same as the previous option, but where the seed is modified through a function. This modification function can be known only to the application developers and users in order to provide a way to obfuscate patterns of access within an application.
- Translation from Notoros Policy **address** to Ethereum address
    1. Generate an Ethereum address directly from the policy's **address** (which represents a valid public key)
        - *Note: This conversion uses a one-way hash, making it impossible to reverse lookup a policy* ***address*** *from an Ethereum address (barring quantum solutions).*

5.2 Setting Execution Boundaries

Applying the definition, we can define the following boundaries in an Ethereum VM:

Variation	Ethereum Scope	Address Translation	Description
0	Account Nonce	Policy address directly converted to account address	Used to read or modify an external account's transaction nonce. Enforces that the transaction is signed by the private key holder.
1	Contract Nonce	Policy generated from contract address	Used to read or modify a contract's transaction nonce.
2	Account Balance	Policy address directly converted to account address	Used to read or modify an external account's balance. Enforces that the transaction is signed by the private key holder.
3	Contract Balance	Policy address generated from contract address	Used to read or modify a contract's balance. Account ownership enforced within VM
5	Contract Code	Policy address generated from contract address	Used to read or modify a contract's bytecode.
7	Contract Storage	Policy address generated from contract address	Used to read or modify a contract's storage. Can be further granularized by specifying the **slot**.

Table 4: Radix/Notoros Policy Type Application in Ethereum

For Account balances, there are multiple valid approaches, including the above and:

- Use a policy **type** to indicate balance storage (as in Table 4), but generate the policy address from the account address. Has the secondary effect of spreading account balances across the entire network and separating from the shard of the private key holder. This increases transaction complexity while providing a validation gateway.

- Using a specific address (e.g. the $0x0$ address), assign all balance records to "contract storage". This consolidates all balance changes (inherently one per transaction) to a single shard, which eases management but centralizes validation.

5.2.1 EVM Policy Granularity

The most granular implementation of the EVM specifies Notoros Policies down to the individual storage slot in the VM. This is the level of granularity that will be used by the primary Notoros EVM implementation on Radix. It allows transactions impacting the same contract to be parallelized if they don't modify the same storage, providing the highest level of per-smart contract throughput and spreading activty across the entire network.

5.3 Blocks

Conceptually, blocks represent ordered groupings of transactions that have been executed on the chain. Due to Notoros's layered architecture, this concept is not directly translatable to mainline Ethereum definitions. As such, it is up to the implementation to decide how to define blocks. In a simple implementation, blocks can be analogous the Notoros Policy that included the executed transaction. There may be implementations where blocks are created secondary to the recording of the original transaction on the ledger and represent execution proofs or other consensus-related data structure. The guides below serve as a starting point for designers.

A transaction's Block Context is determined in large part by the network architecture:

- Where possible, consistently incrementing state version numbers should be used to identify the **blockNumber**. In Notoros's EVM implementation on Radix, the network's accumulated state version is used.

- Unique identifiers on the transaction's meta context should be used for the **blockHash**. In Notoros's EVM on Radix, the Radix Atom's hash is used.

- The **coinbase** (or miner) of the block should be determined based on an algorithm specific to the EVM's implementation context. For some implementations, the **coinbase** can be the primary validator on the transaction. For others, it can be an address based on the contents of the transaction, perhaps referencing a smart contract that enforces consensus rules. The latter is the case for Notoros's EVM on Radix.

- The **timestamp** of the block should be the closest analog to a timestamp available to a transaction. For Notoros's EVM on Radix, the timestamp of the transaction is included in the Radix Atom.

Due to the conceptually different nature of Notoros Policies and blocks, the full definition of a block may have many modifications:

- **stateRoot**, **transactionsRoot**, and **receiptsRoot** are currently undefined due to the sharded nature of the VM. Because some shards may not have access to the full state of the VM, these roots should be calculable from only the states required for

each transaction. Using existing Merkle mechanisms to generate these values is likely to be computationally intensive across many sharded transactions, and thus additional research is being done to determine the most appropriate data structure for these fields. Implementations are free to use the definition that best fits their needs.

- **uncles** is currently undefined due to the consensus mechanisms around sharding. This may be a set of intermediate validation step hashes or other application-specific set.
- **sha3Uncles** is currently undefined due to the consensus mechanisms around sharding. This may be a set of validators, or simply remain empty based on the implementation's requirements.

5.4 Transaction Processing

Raw Ethereum transactions in Notoros are represented in a Notoros Message. Message processing follows these general guidelines:

- Each message in a Notoros transaction should be processed sequentially. Some applications may change the ordering process to obfuscate their operation; as long as the order is deterministicly generated, alternative ordering methods are acceptable.
- The **application** field in the message is the hash of the `genesis.json` (or set of genesis files) used for instantiating the Ethereum VM.
- The **data** field in the message is the raw Ethereum transaction in bytecode (or other acceptable format)
- If the raw transaction's signer is not the **sender** of the message, the transaction is invalid and not executed. *Note: Alternative implementations may be satisfied if the raw transaction's signer also signed the Notoros transaction*

Transaction receipts have the following modifications:

- **cumulativeGasUsed** under most implementations will be the same as **gasUsed** unless multiple **messages** are contained in the Notoros transaction.
- **transactionHash** may not be the hash of the raw Ethereum transaction bytecode, but rather a more suitable ID for the ledger (e.g. Radix Atom hash).

5.5 VM Execution

Execution within the EVM is modified with the following rules:

- When reading values from persistent state (e.g. the `SLOAD` and `EXTCODECOPY` opcodes), the transaction must include a Notoros Policy for the appropriate scope. If it does not, the transaction is reverted. *Note: Some implementations may require that the policy is only "read" (not "write" or "create") if the transaction never modifies state in order to incentivize transaction formatting best practices*

- When recording values to persistent state (e.g. the `SSTORE` and `CREATE` opcodes), the transaction must include a Notoros Policy for the appropriate scope that has either a ”write” or ”create” policy. If it does not, the transaction is reverted.

5.6 Example EVM Transaction Execution

An example ERC20 token transfer will be used to demonstrate the execution of an Ethereum transaction using Notoros.

5.6.1 Assumptions

Assume that Alice and Bob operate compatible EVM instances and agree on the current Ethereum state. They could be using a secondary consensus layer to determine the state, or they could have figured out the current state themselves by executing the respective transaction histories. The “how” doesn't matter for this example, the important thing is that they agree on this starting place.

Furthermore, assume:

- Alice owns Account A, Bob owns Account B, and an ERC20 contract has been deployed to Account C.
- Within the ERC20 contract, Alice has 200 tokens and Bob has 1000 tokens.
- Bob will transfer 500 tokens to Alice.
- Gas Fees will be transfered to contract balance storage at Account R.

5.6.2 State Definitions

First, the chain state and application definitions are determined. Basic resource identifier patterns are used for simplicity. Aribitrary logical clock values and secondary EVM states are chosen. These are shown in Table 5.

Ledger State		Application Definition	
Resource	Clock	EVM Resource	EVM State
B0	5	Account B, External Nonce	3
B2	9	Account B, Account Balance	658,245,332
C5	0	Account C, Contract Code	0x60604...
C7(A)	23	Account C, Contract Storage, Position A	200
C7(B)	17	Account C, Contract Storage, Position B	1000
R3	42	Account R, Contract Balance	22,357,837

Table 5: Example Starting State

5.6.3 Transaction Execution

Next, we can generate a transaction that appropriately updates and reads the resources needed to fully execute the Ethereum message. In addition to Bob's **signature** and **message** call, multiple policies are dictated by the transaction. The transaction's policies and their effects are shown in Table 6.

Looking only at the resources impacted by the transaction, it can be seen that Bob's account nonce is updated, his balance is updated, the token contract's code is read, two slots in the contract's memory are updated (one each for Alice's and Bob's balances), and the destination balance for Bob's gas fees is updated.

Ledger State		Application Definition	
Resource	Policy	EVM Resource	Updated EVM State
B0	B0@5:1	Account B, External Nonce	4
B2	B2@9:1	Account B, Account Balance	654,722,851
C5	C5@0:0	Account C, Contract Code	0x60604...
C7(A)	C7(A)@23:1	Account C, Contract Storage, Position A	700
C7(B)	C7(B)@17:1	Account C, Contract Storage, Position B	500
R3	R3@42:1	Account R, Contract Balance	25,880,318

Table 6: Example Post-Execution State

After executing the transaction within the virtual machine, the updated application state is recorded. Note that the contract code cannot change change due to the READ policy. This allows multiple users to transact in parallel on the same smart contract, providing much greater network throughput than one-contract-one-shard solutions.

5.7 Benefits of EVM Execution with Notoros

Notoros provides a host of scalability and pluggability features to baseline Ethereum with the sharded, multi-layered transaction ordering process. Beyond that, it improves on basic security and usability features such as:

- **Deterministic Execution**: Notoros allows the transaction sender to exactly specify the version of state that the transaction should operate on. If another transaction changes these values first, the Notoros transaction will fail rather than executing on unknown state. This is in contrast to mainnet Ethereum, where malicious actors rely on its lack of specificity to preempt legitimate transactions with their own and change the conditions of the legitimate transaction in frontrunning or double spend attacks.
- **Overlapping VMs**: With multiple VMs capable of operating on the same Notoros network comes the ability to overlap the virtual machine states into a new, "meta" VM state. For example, suppose Alice's and Bob's VMs contain smart contracts that track their respective store's inventory, and each has a unique application ID that indicates which messages to be applied to their VMs. Charlie can then compose a modified

VM that accepts transactions with either of the application IDs to monitor Alice and Bob's inventories. Charlie's VM can also accept transactions with a third application ID that read the state of Alice and Bob's smart contracts and performs actions within Charlie's smart contracts. Extensions of this idea with zero-knowledge-proofs can be constructed to allow transaction and state privacy between VMs. This overlapping use can be considered analagous to colored coins allowing multiple cryptocurrencies on the same ledger.

- **Private Transactions**: Notoros Messages can specify arbitrary bytes, meaning it is possible to encrypt the content of the transaction such that only a consortium of network users can decrypt and apply the transaction, allowing private applications to exist alongside public applications. Combining the overlapping VM concept and zero knowledge proofs, it becomes possible to construct private VMs that are interoperable with public VMs. Additional obfuscation of the transaction can be done by dynamically changing the required scope of any policies in the VM so state accesses appear random to outside viewers.

6 Future Work

6.1 Application State Proof Networks

First, application state proof networks are needed for light client validation. While application state can always be computed directly from the chain state, users need a way of succinctly verifying state through an API. This opens the door for traditional mining mechanisms as a Layer 2 solution.

6.2 Zero Knowledge Proofs

The chained and counter-based nature of the Notoros Protocol makes it attractive for creating zero knowledge systems. Zero Knowledge Virtual Machine interactions would allow private or encrypted virtual machines to interact without sharing state.

6.3 Quantum Resistant Cryptography

Vectorizing resource states and transitions could allow efficient validation using matrices and lattice-based cryptography. This would also resist disruption from advancements in quantum computing.

6.4 Ethereum EIP

An extensive EIP should be created to allow the Ethereum mainnet to adopt Notoros Protocol for their own scaling needs. This would involve updates to the communications protocol, block definitions, and transaction entries.

6.5 Expanded Network Implementations

The Notoros protocol can be implemented on many contemporary networks. Of particular interest are Algorand[9] and Cosmos[7], which are well suited to implementation of the protocol for scaling applications.

References

[1] Leslie Lamport. "A New Solution of Dijkstra's Concurrent Programming Problem". In: (1974). URL: https://lamport.azurewebsites.net/pubs/bakery.pdf.

[2] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: (1978). URL: https://lamport.azurewebsites.net/pubs/bakery.pdf.

[3] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2008). URL: https://bitcoin.org/bitcoin.pdf.

[4] Vitalik Buterin. "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform". In: (2013). URL: https://ethereum.org/en/whitepaper/.

[5] Gavin Wood. "Ethereum: A Secure Decentralized Generalized Transaction Ledger". In: (2013). URL: https://ethereum.org/en/whitepaper/.

[6] David Mazieres. "The Stellar consensus protocol: A federated model for internet-level consensus". In: (2015). URL: https://www.stellar.org/papers/stellar-consensus-protocol.

[7] Ethan Buchman Jae Kwon. "Cosmos Whitepaper: A Network of Distributed Ledgers". In: (2016). URL: https://cosmos.network/resources/whitepaper.

[8] Gavin Wood. "Polkadot: Vision For a Heterogeneous Multi-Chain Framework". In: (2016). URL: https://polkadot.network/PolkaDotPaper.pdf.

[9] Yossi Gilad et al. "Algorand: Scaling Byzantine Agreements for Cryptocurrencies". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 51–68. ISBN: 9781450350853. DOI: 10.1145/3132747.3132757. URL: https://doi.org/10.1145/3132747.3132757.

[10] Mustafa Al-Bassam. "LazyLedger: A Distributed Data Availability Ledger With Client-Side Smart Contracts". In: (2019). arXiv: 1905.09274 [cs.CR].

[11] Daniel P. Hughes. "Cerberus: A Parallelized BFT Consensus Protocol for Radix". In: (2020). URL: https://www.radixdlt.com/wp-content/uploads/2020/03/Cerberus-Whitepaper-v1.0.pdf.

[12] Daniel P. Hughes. "Tempo - Consensus Lessons Learned". In: (2020). URL: https://www.radixdlt.com/post/tempo-consensus-lessons-learned/.

[13] SKALE Labs. "The Skale Network: An Ethereum Interoperable Elastic Blockchain Network". In: (2020). URL: https://skale.network/whitepaper.

[14] Leslie Lamport. "The Writings of Leslie Lamport". In: (2021). URL: http://lamport.azurewebsites.net/pubs/pubs.html.

[15] Inc. MongoDB. *Home*. 2021. URL: https://www.mongodb.com/.

[16] Optimism PBC. *Understanding the Optimistic Ethereum Protocol*. 2021. URL: https://community.optimism.io/docs/protocol/protocol.html.

[17] Redis. *Home*. 2021. URL: https://redis.io/.