Enterprise Ethereum Alliance Permissioned Blockchains Specification

Version 1, 29 May 2020

Latest editor's draft:

https://entethalliance.github.io/client-spec/chainspec.html

Editors:

Robert Coote (PegaSys)

Chaals Nevile (Enterprise Ethereum Alliance)

Grant Noble (PegaSys)

George Polzer (Everymans.ai)

Contributors to this version:

Imran Bashir (Quorum), Juan Blanco (Nethereum), Dan Burnett (PegaSys), Cody Burns (Accenture), Jean-Charles Cabalguen (iExec), Rob Dawson (PegaSys), Antony Denyer (Web3Labs), Sara Feenan (Clearmatics), Andreas Freund (Consensys), David Izzo (DTCC), Arash Mahboubi (PegaSys), George Ornbo (Clearmatics), Fernando Paris (ioBuilders), Yaz Khoury (ETC Cooperative), Julien Marchand (PegaSys), Niraj Pore (Fnality), Yorke Rhodes III (Microsoft), Lucas Saldanha (PegaSys), Roberto Saltini (PegaSys), Satpal Sandhu (Quorum), Przemek Siemion (Santander), Bob Summerwill (ETC Cooperative), Conor Svensson (Web3Labs), Sai Murali Krishna V. (Quorum), John Whelan (Santander), Jim Zhang (Kaleido), Lei Zhang (iExec), Weijia Zhang (Wanchain)

Abstract

This document, the Enterprise Ethereum Alliance Permissioned Blockchains specification, defines requirements for Enterprise Ethereum blockchains to ensure they can be processed interoperably by Enterprise Ethereum clients that conform to the Enterprise Ethereum Client specification [EEA-client]. Its primary intended audience is operators of Enterprise Ethereum blockchains.

Status of This Document

This section describes the status of this document at the time of its publication. Newer documents may supersede this document.

This is version 1 of the Enterprise Ethereum Alliance Permissioned blockchains specification, approved by the EEA Board as a formal publication of the EEA.

Although predicting the future is known to be difficult, as well as ongoing quality enhancement, future work on this Specification is expected to include the following aspects:

- Typical enterprise features for permissioning contracts.
- Adopting an agreed [Byzantine-Fault-Tolerant] consensus algorithm
- Cross-chain interoperability
- Tracking developments for Ethereum 1.x and Ethereum 2.0

The group is also expecting to hear about further implementation experience, that could potentially lead to proposed modifications. This particularly applies to experimental sections of the specification:

- Organization Registry contracts
- The object syntax for `maxCodeSize`

Please send any comments to the EEA Technical Steering Committee through https://entethalliance.org/contact/.

Table of Contents

1. Introduction

This section is non-normative.

This document, Enterprise Ethereum Alliance Permissioned Blockchains specification, defines requirements for Enterprise Ethereum blockchains. Operators of Enterprise Ethereum blockchains who want to be sure that they can use different conformant Enterprise Ethereum clients on their blockchain interoperably can do so by meeting the requirements described in this specification.

This is a companion document to the Enterprise Ethereum Alliance Client Specification [EEA-client], which defines requirements for Enterprise Ethereum clients to ensure interoperability of clients on an Enterprise Ethereum blockchain.

For the purpose of this Specification:

- ***Public Ethereum*** (Ethereum) is the public blockchain-based distributed computing platform featuring smart contract (programming) functionality defined by the [Ethereum-Yellow-Paper], [EIPs], and associated specifications.
- ***Ethereum MainNet*** (MainNet) is the public Ethereum blockchain whose `chainid` and `network ID` are both `1`.
- ***Enterprise Ethereum*** is a standards-based ecosystem of software that extends Ethereum to provide functionality important to solve different use cases for Ethereum blockchains that have requirements not met by Public Ethereum. These extensions provide the ability to perform private transactions, and enforce permissioning, for Ethereum blockchains that use them.

- An ***Enterprise Ethereum blockchain*** is an Ethereum-based blockchain, that meets the requiremments described in this specification, in order to enable Enterprise Ethereum clients to operate it.
- An ***Enterprise Ethereum client*** (a client) is the software that implements Enterprise Ethereum, and is used to run nodes on an Enterprise Ethereum blockchain. Clients need to meet the requirements defined in the Enterprise Ethereum Alliance Client Specification.
- A ***node*** is an instance of an Enterprise Ethereum client running on an Enterprise Ethereum blockchain.

1.1 Why Produce a Blockchain Specification?

A number of vendors are developing Enterprise Ethereum clients, that can communicate with each other and ***interoperate*** reliably on a given Enterprise Ethereum blockchain.

It is therefore important to define an Enterprise Ethereum blockchain more formally than just *the obvious implications from reading the Client Specification*.

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *SHALL*, and *SHOULD* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.1 Experimental Requirements

This Specification includes requirements and Application Programming Interfaces (APIs) that are described as *experimental*. ***Experimental*** means that a requirement or API is in early stages of development and might change as feedback is incorporated. Implementors are encouraged to implement these experimental requirements, with the knowledge that requirements in future versions of the Specification are not guaranteed to be compatible with the current version. Please send comments and feedback on experimental portions of this Specification to the EEA Technical Steering Committee at https://entethalliance.org/contact/.

3. Security Considerations

This section is non-normative.

Security of information systems is a major field of work. Enterprise Ethereum software development shares with all software development the need to consider security issues and the obligation to update implementations in line with new information and techniques to protect its users and the ecosystem in which it operates.

However, some aspects of Ethereum in general, and Enterprise Ethereum specifically, are especially important in an enterprise environment.

3.1 Positive Security Design Patterns

Complex interfaces increase security risk by making user error more likely. For example, entering Ethereum addresses by hand is prone to errors. Therefore, implementations can reduce the risk by providing user-friendly interfaces, ensuring users correctly select an opaque identifier using tools like a contact manager.

Gas (defined in the [Ethereum-Yellow-Paper]) is a virtual pricing mechanism for transactions and smart contracts that is implemented by Ethereum to protect against Denial of Service attacks and resource-consumption attacks by compromised, malfunctioning or malicious nodes. Enterprise Ethereum provides additional tools to reduce security risks, such as more granular permissions for actions in a network.

Permissioning can play an important role in mitigating network-level attacks, like the 51% attack. However, it is important to ensure permissioning administration does not compromize security.

3.2 Handling of Sensitive Data

The implications of private data storage are also important to consider, and motivate several requirements within this Specification.

The long-term persistence of encrypted data exposes it to eventual decryption by brute-force attack. Advances in cryptanalysis as well as computing power increase the likelihood of this decryption, by decreasing the cost. A future shift to post-quantum cryptography is a current concern, but it is unlikely to be the last advance in the field. Assuming no encryption scheme endures for eternity, a degree of protection is required to reasonably exceed the lifetime of the data's sensitivity.

3.3 Upgradeable and Proxy contracts

Proxy contracts to enable upgrades for core contracts such as permissioning need to be designed carefully to ensure that upgrades can be made by the parties intended, and only by them, through the lifetime of the blockchain. In particular, storage collisions and function signature collisions [Function-collision] can arise due to the way the EVM processes smart contracts. These issues, and important precautions, caveats, and mitigations are described in varoious articles, such as "Building Upgradeable Smart Contracts" [Upgrade-contracts].

4. Enterprise Ethereum Architecture

This section is non-normative.

The following diagram shows the relationship between Enterprise Ethereum components.

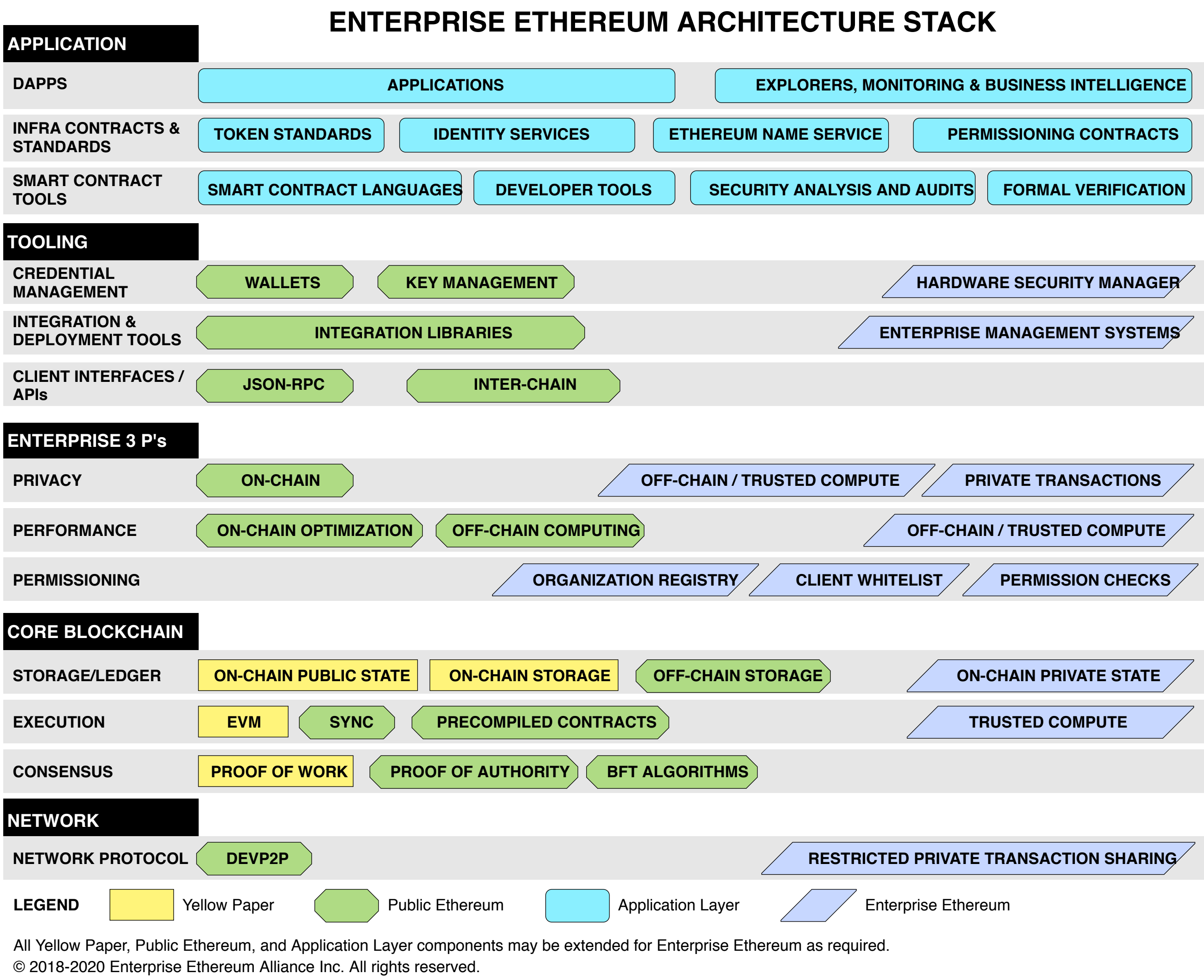

Figure 1 Enterprise Ethereum Architecture Stack

The architecture stack for Enterprise Ethereum consists of five layers:

- Application
- Tooling
- Enterprise 3 P's
- Core Blockchain
- Network.

These layers are described in the following sections.

5. Application Layer

The Application layer is where higher-level services are provided. For example, Ethereum Name Service (ENS), node monitors, blockchain state visualizations and explorers, and any other applications of the ecosystem envisaged.

5.1 ÐApps Sublayer

Decentralized Applications, or ***ÐApps***, are software applications running on a decentralized peer-to-peer network, often a blockchain. A ÐApp might include a user interface running on another (centralized or decentralized) system. ÐApps run on top of Ethereum.

Also at the ÐApps sublayer are blockchain explorers, tools to monitor the blockchain, and business intelligence tools.

5.2 Infrastructure Contracts and Standards Sublayer

This section is non-normative.

Some important tools for managing a blockchain are built at the Application layer. These components together make up the Infrastructure Contracts and Standards sublayer.

A ***Permissioning contract*** determines whether nodes and accounts can access, or perform specific actions on, an Enterprise Ethereum blockchain, according to the needs of the blockchain. Permissioning contracts can implement Role-based access control (RBAC) [WP-RBAC] or Attribute-based access control (ABAC) [WP-ABAC], as well as simpler permissioning models, as described in the Permissioning Management Examples section of the Implementation Guide [EEA-implementation-guide].

Token standards provide common interfaces and methods along with best practices. These include [ERC-20], [ERC-223], [ERC-621], [ERC-721], and [ERC-827].

The ***Ethereum Name Service*** (ENS) provides a secure and decentralized mapping from simple, human-readable names to Ethereum addresses for resources both on and off the blockchain.

5.3 Smart Contract Tools Sublayer

Enterprise Ethereum inherits the smart contract tools used by public Ethereum such as smart contract languages and associated parsers, compilers, and debuggers.

6. Tooling Layer

The Tooling layer contains the APIs used to communicate with clients. The ***Ethereum JSON-RPC API***, implemented by public Ethereum is the primary API to submit transactions for execution and to deploy smart contracts. The [JSON-RPC] remote procedure call protocol and format is used for the JSON-RPC API implementation. Other APIs are allowed, including those intended for inter-blockchain operations and to call external services, such as trusted *oracles*.

Enterprise Ethereum implementations can restrict operations based on permissioning and authentication schemes.

6.1 Credential Management Sublayer

Credentials, in the context of Enterprise Ethereum blockchains, refer to an individual's cryptographic private keys, which are associated with that user's Ethereum account.

6.1.1 Registry for Organizational Accounts

This section is experimental. The EEA is looking for feedback on

- how the Organization Registry is used;
- whether the design is clearly explained
- whether the particular design can be improved for better usability

This section presents a smart contract based registry, to provide on-chain validation that a particular Ethereum accounts or nodes is owned by a participating organization in an Enterprise Ethereum blockchain.

Ethereum accounts are used in both system level functionalities and application level operations:

- consensus block proposers to sign the proposed block
- consensus block validators to sign the vote on the proposed block
- p2p subsystem to sign p2p messages
- applications to sign submitted transactions

In enterprise settings, identifying organizational ownership of signing accounts is critical in many use cases. In the off-chain world, organizations, private businesses, governments and academic institutions all have defined identities. It is critical to have a robust binding between the organization's off-chain identity and their on-chain signing accounts.

One example of where this binding can be useful is permissioning. A permissioning smart contract can use this registry to look up the organization that owns the subject account, validate the ownership by verifying the attached cryptographic proof, and make permissioning decisions.

The binding is established with identity proofs. An ***identity proof*** is a cryptographical data structure that can be independently verified, either on-chain in the smart contract, or off-chain by client applications, describing the relationship between defined entities such as an account, node, or another participating organization (by defining the root signing account as a member of a participating organization).

The registry does not act as the source of truth for network membership. The membership of the blockchain network is maintained by the permissioning contracts.

The registry relies on client certificates or equivalent technologies. It is important to consider the mechanisms to set and revoke expiration, to allow for use cases with different freshness requirements.

ORGIDS-300: Enterprise Ethereum Blockchains *MAY* implement a smart contract based Organization Registry that provides cryptographic bindings between Ethereum accounts and their owning organization with identity proofs.

An ***Organization Registry*** follows the design outlined below.

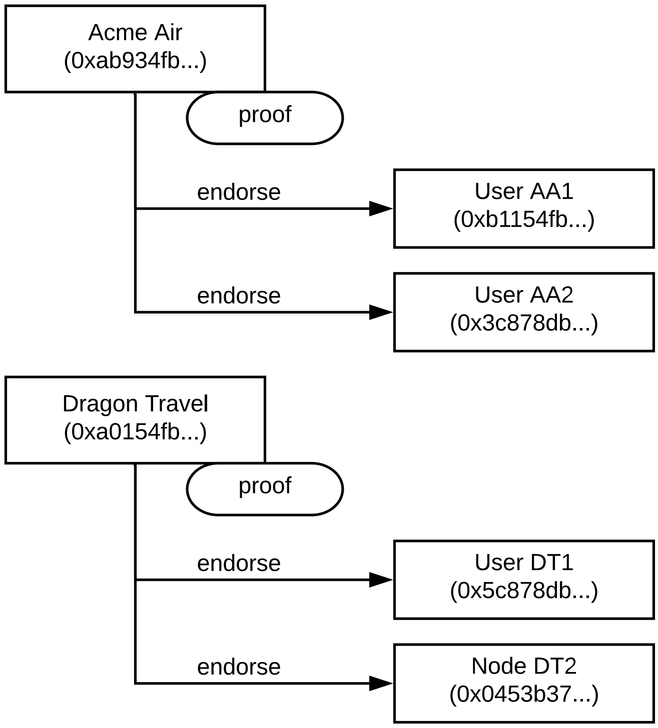

Figure 2 Organizational Ownership of Accounts

A ***participating organization*** represents a collection of accounts and nodes that share a collective identity, for example they are owned by the same company, or they are held by officers of a particular organization. The participating organization is identified by an account called the ***root signing account***.

ORGIDS-310: An Organization Registry *MUST* require all root entries to present an identity proof with:

- a signing authority attesting that the proof has been uniquely issued for the organization identified by the subject
- a digital signature generated by the private key for the root signing account

With the above properties, the proof not only demonstrates that the submitter of the registration is associated with the subject organization, because it has access to the organization's signing authority, but also demonstrates possession of the root signing account's private key.

Once the organization's root registry is established, the organization can add more entries for Ethereum accounts or nodes the participating organization uses on the Enterprise Ethereum blockchain. As illustrated above, other accounts or nodes are "attached" under the root account. The smart contract requires the organization's root signing account to be used to add children, thus ensuring the organizational ownership of these "child" accounts are clearly demonstrated.

ORGIDS-320: An Organization Registry *MUST* require the root signing account for an organization to insert child entries under that organization.

The accounts inserted under the root account can either be Ethereum addresses or enode IDs.

6.1.1.1 Pluggability To Support Different Types of Proofs

ORGIDS-330: An Organization Registry *MUST* support an extensibility mechanism to allow different types of proofs to be submitted and verified.

ORGIDS-340: An Organization Registry *MUST* support at least one of the following proof types:

- X.509 Certificates [rfc5280] generated by a trusted Certificate Authority (CA), attached to a chain of intermediate CAs leading up to a globally recognized root CA.
- A Verifiable Presentations data structure [VC-presentations] as defined within the W3C Verifiable Credentials Data Model [vc-data-model].

Verifiable Credentials is a new W3C standard in the Decentralized Identifier (DID) ecosystem. The Verifiable Credentials data model is not itelf suitable as a proof type because it does not support chain-unique challenges for replay attack protections.

ORGIDS-350: An Organization Registry *MAY* verify the proofs in the smart contract and immediately reject a registration that did not present a valid proof, or allow a registration to be validated or invalidated by an off-chain agent.

ORGIDS-360: If an Organization Registry performs proof validation in the smart contract, it *MUST* offer at least the following options to support different "freshness" requirements:

- validate once during registration, rely on administration operations to update expired or revoked proofs
- validate during registration, replicate expiration date in the contract for faster checking subsequently
- validate every time the account is used

ORGIDS-370: Identity proofs *MUST* protect against re-use by a malicious party, by embedding a chain-unique challenge segment, such as the chain ID, in the signed claims inside the proof.

Since the proofs are available to all network participants, protection against taking a proof from one network and using it in a different network is essential.

An issuer of an Identity proof signs an identity claim that includes a unique identifier for the network where the proof is issued.

ORGIDS-380: A Registry for Organizational Accounts *MUST* not allow a registered proof to be used to register a new root entry.

Using an X.509 certificate as an illustration:

```
Subject: CN=Acme Air-290528951
Issuer: CN=Acme Air
   |
   | (signed by)
   |
Intermediate CA:
  Subject: CN=Acme Air
  Issuer: CN=Symantec
```

If the `CN` value of the *subject* property contains the chain ID 290528951, then a malicious party will not be able to steal this certificate and re-use it in a different blockchain network because the chain ID will not match. It is imnpossible to modify the chain ID without the private key of the intermediate CA.

6.1.1.2 Smart Contract Based Registry for Organizational Accounts

The following interface is the minimal functionality set for the smart contract based registry to work according to the proposed design. Functions such as getters and queries might be helpful as optional enhancements.

```
Interface OrganizationalIDRegistry {
// establish the root account for the organization, the address of the t
ransaction
// sender will be recorded by the contract as the root account for the o
rganization.
// Implementations may choose to validate the proof inside the smart con
tract, and
// cache certain aspects of the proof to the state that helps with faste
r checking for
// administrative operations, such as expiration date.
function registerOrganization(bytes32 orgID, bytes32 orgName, string pro
of) external;
// endorse and register user account within the organization
// the user account will be inserted under the root account in the ident
ity tree
function registerUser(bytes32 metadata, address userAccount) external;
// endorse and register a node within the organization
// the enode ID will be inserted under the root account in the identity
```

```
tree
function registerNode(bytes32 metadata, bytes32 enodeIDHigh, bytes32 eno
deIDLow) external;
// marks the user account within the organization as deleted/inactive
// the operation is only allowed with the root account
function removeUser(address userAccount) external;
// marks the node within the organization as deleted/inactive
// the operation is only allowed with the root account
function removeNode(bytes32 enodeIDHigh, bytes32 enodeIDLow) external;
// returns the root organization account that owns the user account
function getOwningOrganization(address userAccount) external view return
s (bytes32 orgID, bytes32 orgName);
// returns the root organization account that owns the node
function getOwningOrganization(bytes32 enodeIDHigh, bytes32 enodeIDLow)
external view returns (bytes32 orgID, bytes32 orgName);
// updates the proof for the organization's root account
function updateProof(address rootAccount) external view returns (string
proof);
// returns the proof for the organization's root account for verificatio
n
function getProof(address rootAccount) external view returns (string pro
of);
// broadcast registered organizations for participants to download and i
nspect the proof
event OrganizationRegistered(bytes32 orgID, bytes32 orgName, address roo
tAccount, string proof);
// broadcast registered users
event UserRegistered(bytes32 orgID, address userAccount);
// broadcast removed users
event UserRemoved(bytes32 orgID, address userAccount);
// broadcast registered nodes
event NodeRegistered(bytes32 orgID, ytes32 enodeIDHigh, bytes32 enodeIDL
ow);
// broadcast removed nodes
event NodeRemoved(bytes32 orgID, ytes32 enodeIDHigh, bytes32 enodeIDLow)
;
}
```

6.2 Integration and Deployment Tools Sublayer

This section is non-normative.

This sublayer provides integration with enterprise management systems using common APIs, libraries, and techniques.

6.3 Client Interfaces and APIs Sublayer

An Ethereum JSON-RPC API is used to communicate between ÐApps and nodes.

6.3.1 Permissioning Smart Contract

This section presents interfaces for the permissioning contracts. These are the smart contracts needed on the blockchain to provide necessary information for Enterprise Ethereum clients to enforce permissioning models in an interoperable manner. There are permissioning interfaces for both both nodes and accounts.

It is based on a chain deployment architecture where permissioning is split into two parts:

- Permissioning enforcement functions.

  Clients call permission-allowed functions within the permissioning contracts. These are common functions for all clients on the Enterprise Ethereum blockchain to use. These functions include:

  - `connectionAllowed`, in the node permissioning contract, to determine whether to permit a connection with another node.
  - `transactionAllowed`, in the account permissioning contract, to determine whether to accept a transaction received from a given Ethereum account.

  A client is not required to be able to update the permissioning scheme nor have knowledge of its implementation.

  The node and account permissioning contracts emit `NodePermissionsUpdated` and `AccountPermissionsUpdated` events respectively, when the underlying rules are changed. Clients register for these events, that signal when to re-assess any permissions that were granted, and when to re-assess any permission check results that were cached.

  The events contain the `addsRestrictions` and `addsPermissions` Boolean flags. If either flag is set to `true`, any previous `connectionAllowed` or `transactionAllowed` call could now result in a different outcome, because the previously checked permissions have changed. If `addsRestrictions` is `true`, then one or more `connectionAllowed` or `transactionAllowed` calls that previously returned `true` will now return `false`. Similarly, if `addsPermissions` is `true`, at least one `connectionAllowed` or `transactionAllowed` call that previously returned `false` will now return `true`.

- Permissioning management functions.

> These functions provide the ability to configure and manage the permissioning model in use. These include the bulk of the constructs used to organize permissions, processes to adjust permissions, administration of the permissioning mechanism, and enforcing any regulatory requirements.
>
> The definition of these management functions depends on the permissioning model in use for the specific Enterprise Ethereum blockchain. It is outside the scope of this Specification, but crucial to the operation of the system. Enterprise Ethereum blockchain operators can choose any permissioning model that suits their needs.

Implementations of the permissioning contracts (both enforcement and management functions) are provided on the Enterprise Ethereum blockchain by the blockchain operator. The implementation of permissioning enforcement functions, such as `connectionAllowed`, is part of the permissioning management smart contract.

When a management function is called that updates the permissioning model, the node or account smart contract interfaces emit `NodePermissionsUpdated` or `AccountPermissionsUpdated` events respectively, based on the permissions change.

6.3.1.1 Node Permissioning

The ***Node permissioning contract*** restricts the peer connections that can be established with other nodes in the Enterprise Ethereum blockchain. This helps to prevent interference and abuse by external parties and can establish a trusted whitelist of nodes.

The `connectionAllowed` function returns a `bytes32` type, which is interpreted as a bitmask with each bit representing a specific permission for the connection.

PERMIT-020 If the permissions for a blockchain are updated to revoke any permission previously granted to nodes, the node permissioning contract *MUST* emit a `NodePermissionsUpdated` event containing an `addsRestrictions` property with the value `true`. See also **PERM-220:**.

PERMIT-030 If the permissions for a blockchain are updated to grant any new permissions for nodes the node permissioning contract *MUST* emit a `NodePermissionsUpdated` event containing an `addsRestrictions` property with the value `false`. See also **PERM-230:**.

6.3.1.1.1 Node Permissioning Functions

PERMIT-070 The node connection rules *MUST* support both the IPv4 and IPv6 protocol versions.

IPv6 addresses are represented using their logical byte value with big endian byte ordering. IPv4 addresses are specified in the IPv4 reserved space within the IPv6 address space, which is found at `0000:0000:0000:0000:0000:ffff:`, and can be be assembled by taking the logical byte value of the IPv4 address with big endian byte ordering, and prefixing it with 80 bits of zeros followed by 16 bits of ones.

The ***`connectionAllowed`*** function implements the following interface, including the `NodePermissionsUpdated` event:

```
Interface
[
  {
    "name": "connectionAllowed",
    "stateMutability": "view",
    "type": "function",
    "inputs": [
      {
        "name": "sourceEnodeHigh",
        "type": "bytes32"
      },
      {
        "name": "sourceEnodeLow",
        "type": "bytes32"
      },
      {
        "name": "sourceIp",
        "type": "bytes16"
      },
      {
        "name": "sourcePort",
        "type": "uint16"
      },
      {
        "name": "destinationEnodeHigh",
        "type": "bytes32"
      },
      {
        "name": "destinationEnodeLow",
        "type": "bytes32"
      },
      {
        "name": "destinationIp",
```

```
        "type": "bytes16"
      },
      {
        "name": "destinationPort",
        "type": "uint16"
      },
    ],
    "outputs": [
      {
        "name": "result",
        "type": "bytes32"
      }
    ]
  },
  {
    "type": "event",
    "name": "NodePermissionsUpdated",
    "inputs": [
    {
      "name": "addsRestrictions",
      "type": "bool",
      "indexed": false
    },
    {
      "name": "addsPermissions",
      "type": "bool",
      "indexed": false
    }
    ]
  }
]
```

Arguments

- `sourceEnodeHigh`: The high (first) 32 bytes of the enode address of the node initiating the connection.
- `sourceEnodeLow`: The low (last) 32 bytes of the enode address of the node initiating the connection.
- `sourceIp`: The IP address of the node initiating the connection. If the address is IPv4, it should be prefixed by 80 bits of zeros and 16 bits of ones, bitmasking it such that it fits the IPv4 reserved space in IPv6. For example, `::ffff:127.0.0.1`.
- `sourceEnodePort`: The peer-to-peer listening port of the node initiating the connection.

- `destinationEnodeHigh`: The high (first) 32 bytes of the enode address of the node receiving the connection.
- `destinationEnodeLow`: The low (last) 32 bytes of the enode address of the node receiving the connection.
- `destinationIp`: The IP address of the node receiving the connection. If the address is IPv4, it should be prefixed by 80 bits of zeros and 16 bits of ones, bitmasking it such that it fits the IPv4 reserved space in IPv6. For example, `::ffff:127.0.0.1`.
- `destinationEnodePort`: The peer-to-peer listening port of the node receiving the connection.
- `result`: A bitmask of the permissions granted for this connection.
- `addsRestrictions`: If the rules change that caused the `NodePermissionsUpdated` event to be emitted involves further restricting existing permissions, this will be `true`, otherwise `false`.
- `addsPermissions`: If the rules change that caused the `NodePermissionsUpdated` event to be emitted involves granting new permissions, this will be `true`, otherwise `false`.

6.3.1.1.2 Node Permissions

While the core premise of node permissioning is whether a connection is allowed to occur or not, there are additional restrictions that can be imposed on a connection between two nodes based on the permitted behavior of the nodes.

The various permissions that can be granted to a connection are represented by bits being set in the bitmask response from `connectionAllowed`. Where bits are unset, the client restricts the behavior of the remote node according to the unset bits.

The remaining bits in the response are normally set to one. If any of the remaining bits are zero, an unknown permission restriction was placed on the connection and the connection will be denied. These unknown zeros are likely to represent permissions defined in future versions of this specification. Where they cannot be interpreted by a client, the connection is rejected.

Connection Permitted

Permission Bit Index: 0

The connection is allowed to be established.

6.3.1.1.3 Client Implementation

A client connecting to a chain that maintains a permissioning contract finds the address of the contract in the network configuration. When a peer connection request is received, or a new connection request initiated, the permissioning contract is queried to assess whether the connection is permitted. If permitted, the connection is established and when the node is queried for peer discovery, this connection can be advertised as an available peer. If not permitted, the connection is either refused or not attempted, and the peer excluded from any responses to peer discovery requests.

During client startup and initialization the client begins at a bootnode and initially has a global state that is out of sync. Until the client reaches a trustworthy head it is unable to reach a current version of the node permissioning that correctly represents the current blockchain's state.

6.3.1.1.4 Chain Initialization

CONFIG-040: A node permissioning contract with the `connectionAllowed` function as defined in section 6.3.1.1.1 Node Permissioning Functions, *MUST* be included in the genesis block (block 0), available at the address specified in the network configuration parameter ***`nodePermissionContract`***.

The configuration of the node permissioning contract allows initial nodes to establish connections to each other.

6.3.1.2 Account Permissioning

The ***account permissioning contract*** controls which accounts are allowed to send transactions, and the type of transactions permitted.

6.3.1.2.1 Account Permissioning Smart Contract Interface Function

The ***`transactionAllowed`*** function implements the following interface, including the ***`AccountPermissionsUpdated`*** event:

```
Interface
[
  {
    "name": "transactionAllowed",
    "stateMutability": "view",
    "type": "function",
```

```
      "inputs": [
        {
          "name": "sender",
          "type": "address"
        },
        {
          "name": "target",
          "type": "address"
        },
        {
          "name": "value",
          "type": "uint256"
        },
        {
          "name": "gasPrice",
          "type": "uint256"
        },
        {
          "name": "gasLimit",
          "type": "uint256"
        },
        {
          "name": "payload",
          "type": "bytes"
        }
      ],
      "outputs": [
        {
          "name": "result",
          "type": "bool"
        }
      ]
    },
    {
      "type": "event",
      "name": "AccountPermissionsUpdated",
      "inputs": [
      {
        "name": "addsRestrictions",
        "type": "bool",
        "indexed": false
      },
      {
        "name": "addsPermissions",
        "type": "bool",
        "indexed": false
```

```
      }
      ]
    }
  ]
```

Arguments

- `sender`: The address of the account that created this transaction.
- `target`: The address of the account or contract that this transaction is directed at. For a creation contract where there is no target, this should be zero filled to represent the `null` address.
- `value`: The eth value being transferred in this transaction.
- `gasPrice`: The gas price included in this transaction
- `gasLimit`: The gas limit in this transaction.
- `payload`: The payload in this transaction. Either empty if a simple value transaction, the calling payload if executing a contract, or the EVM code to be deployed for a contract creation.
- `addsRestrictions`: If the rules change that caused the `AccountPermissionsUpdated` event to be emitted involves further restricting existing permissions, this will be `true`.
- `addsPermissions`: If the rules change that caused the `AccountPermissionsUpdated` event to be emitted grants new permissions, this will be `true`.

Return value

- boolean `result`: A value of `true` means the account submitting the transaction has permission to submit it.

PERMIT-090 Account permissioning contracts *MUST* respond with a `bool` value of `true` for the case where the transaction is allowed, or `false` for the case where the transaction is not allowed.

6.3.1.2.2 Client Implementation

A client connecting to a chain that maintains a smart contract exposing the account permissioning interface can expect to be supplied the address of the contract.

Reading of a contract is an unrestricted operation.

6.3.1.2.3 Contract Implementation

When a transaction is checked by the contract it can be assessed by any of the fields provided to restrict operations, such as transferring value between accounts, rate limiting spend or receipt of value, restricting the ability to execute code at an address, limiting gas expenditure or enforcing a minimum expenditure, or restricting the scope of a created contract.

When checking the execution of code at an address, it can be useful to be aware of the `EXTCODEHASH` EVM operation, which allows for checking whether there is code present to be executed at the address that received the request.

For restricting the scope of created contracts it might be necessary to do static code analysis of the EVM bytecode payload for properties that are not allowed. For example, restricting creation of contracts that employ the create contract opcode.

6.3.1.2.4 Chain Initialization

CONFIG-050 A permissioning contract with the `transactionAllowed` function as defined in section 6.3.1.2.1 Account Permissioning Smart Contract Interface Function, *MUST* be included in the genesis block (block 0), available at the address specified in the network configuration parameter ***`transactionPermissionContract`***.

The permissioning contract function is configured so initial accounts can perform required value transactions, a predetermined set of accounts can invoke the contracts defined in the genesis file, and if desired, a predetermined set of accounts can create new contracts.

7. Enterprise 3 P's Layer

Privacy, *performance*, and *permissioning* are the "3 P's" of Enterprise Ethereum. This section describes the extensions in Enterprise Ethereum that support these requirements.

Privacy and performance solutions are broadly categorized into:

- ***Layer 1*** solutions, which are implemented at the base level protocol layer using techniques such as [sharding] and easy parallelizability [EIP-648].
- ***Layer 2*** solutions, which do not require changes to the base level protocol layer. They are implemented at the application protocol layer, for example using [Plasma], [state-channels], and Off-Chain Trusted Computing mechanisms.

7.1 Privacy Sublayer

Many use cases for Enterprise Ethereum blockchains have to comply with regulations related to privacy. For example, banks in the European Union are required to comply with the European Union revised Payment Services Directive [PSD2] when providing payment services, and the General Data Protection Regulation [GDPR] when storing personal data regarding individuals.

Enterprise Ethereum clients support privacy with techniques such as private transactions and enabling an Enterprise Ethereum blockchain to permit anonymous participants. Clients can also support privacy-enhanced Off-Chain Trusted Computing.

New privacy mechanisms are are also being explored as extensions to public Ethereum, including ***zero-knowledge proofs*** [ZKP], which is a cryptographic technique where one party (the prover) can prove to another party (the verifier) that the prover knows a value x, without conveying any information apart from the fact that the prover knows the value. [ZK-STARKS] is an example of a zero-knowledge proof method.

A ***transaction*** is a request to execute operations on a blockchain that change the state of one or more accounts. Transactions are a core component of most blockchains, including Public Ethereum and Enterprise Ethereum. Nodes processing transactions is the fundamental basis of adding blocks to the chain.

A ***private transaction*** is a transaction where some information about the transaction, such as the payload data, or the sender or the recipient, is only available to the subset of parties privy to that transaction.

Enterprise Ethereum implementations can also support off-chain trusted computing, enabling privacy during code execution.

7.1.1 On-chain Privacy

This section is non-normative.

Various on-chain techniques can improve the security and privacy capabilities of Enterprise Ethereum blockchains.

> NOTE: On-chain Security Techniques
>
> Future on-chain security techniques could include techniques such as [ZK-STARKS], range proofs, or ring signatures.

7.1.2 Off-chain Privacy (Trusted Computing)

This section is non-normative.

Off-chain trusted computing uses a privacy-enhanced system to handle some of the computation requested by a transactions. Such systems can be hardware-based, software-based, or a hybrid, depending on the use case.

The EEA has developed Trusted Compute APIs for Ethereum-compatible trusted computing [EEA-OC].

7.2 Performance Sublayer

This section is non-normative.

Performance is an important requirement because many use cases for Enterprise Ethereum blockchains imply a high volume of transactions, or computationally heavy tasks. The overall performance of a blockchain is constrained by the slowest node.

There are many different aspects of performance, and instead of mandating specific requirements, this Specification notes the importance of performance, leaving Enterprise Ethereum blockchain implementers free to implement whatever strategies are appropriate.

This Specification does not constrain experimentation to improve performance. This is an active area of research, and it is likely various techniques to improve performance will be developed over time, which cannot be exactly predicted.

This Specification does mandate or allow for several optimizations to improve performance. The most important techniques maximize the throughput of transactions.

7.2.1 On-chain (Layer 1 and Layer 2) Scaling

Techniques to improve performance through scaling are valuable for blockchains where processing is kept on the blockchain and have high transaction throughput requirements.

On-chain (layer 1) scaling techniques, like [sharding], are changes or extensions to the public Ethereum protocol to facilitate increased transaction speeds.

On-chain (layer 2) scaling techniques use smart contracts, and approaches like [Plasma], or [state-channels], to increase transaction speed without changing the underlying Ethereum protocol. For more information, see [Layer2-Scaling-Solutions].

7.2.2 Off-chain (Layer 2 Compute)

Off-chain computing can be used to increase transaction speeds, by moving the processing of computationally intensive tasks from nodes processing transactions to one or more trusted computing services. This reduces the resources needed by nodes allowing them to produce blocks faster.

7.3 Permissioning Sublayer

This section is non-normative.

Permissioning is the property of a system that ensures operations are executed by and accessible to designated parties. For Enterprise Ethereum, permissioning refers to the ability of a node to join an Enterprise Ethereum blockchain, and the ability of individual accounts or nodes to perform specific functions. For example, an Enterprise Ethereum blockchain might allow only certain nodes to act as validators, and only certain accounts to instantiate smart contracts.

Enterprise Ethereum provides a permissioned implementation of Ethereum supporting peer node connectivity permissioning, account permissioning, and transaction type permissioning.

7.3.1 Nodes

PERMIT-030: The node permissioning contract *SHOULD* specify a list of static peer nodes to establish peer-to-peer connections with. See also **NODE-010:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

PERMIT-040: The node permissioning contracts *MUST* manage a whitelist required by **NODE-030** through a transaction into a smart contract. See also **NODE-030:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

7.3.2 Ethereum Accounts

For the purpose of this Specification:

- An ***organization*** is a logical group composed of Ethereum accounts, nodes, and other organizations or suborganizations. A ***suborganization*** is an organization controlled by and subordinate to another organization. An organization typically represents an enterprise, or some identifiable part of an enterprise. For the purpose of permissioning, organizations roughly correspond to the UNIX concept of groups.

- A ***user*** is a human or an automated process interacting with an Enterprise Ethereum blockchain using the Ethereum JSON-RPC API. The identity of a user is represented by an Ethereum account. Public key cryptography is used to sign transactions made by the user so the EVM can authenticate the identity of a user sending a transaction.
- An ***Ethereum account*** is an established relationship between a user and an Ethereum blockchain. Having an Ethereum account allows users to interact with a blockchain, for example to submit transactions or deploy smart contracts.
- ***Groups*** are collections of users that have or are allocated one or more common attributes. For example, common privileges allowing users to access a specific set of services or functionality.
- ***Roles*** are sets of administrative tasks, each with associated permissions that apply to users or administrators of a system, used for example in RBAC permissioning contracts.

PERMIT-010: An Enterprise Ethereum blockchain account permissioning contract *MUST* enable whitelisting accounts that are permitted to interact with the blockchain. See also **PART-010:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

PERMIT-060: The account permissioning contract *MUST* manage separate permissioning for an account to:

- Deploy smart contracts.
- Call functions that change the state of specified smart contracts.
- Perform a value transfer to a specified account.

7.3.3 Additional Permissioning Requirements

Permissioning contracts can use the Proxy / Updateable contract patttern, for example to ensure that it is possible to change the management functions if an Enterprise Ethereum Blockchain needs a system with more features. If a new node is trying to synchronise the entire chain, it is important that it can "replay" each transaction, including those that make changes to the management of permissioning.

PERMIT-080 Permissioning contracts that are updateable *MUST NOT* allow changes through a private transaction.

Network configuration refers to the collection of settings defined for a blockchain, such as which consensus algorithm to use, addresses of permissioning smart contracts, and so on.

8. Core Blockchain Layer

The Core Blockchain layer consists of the Storage and Ledger, Execution, and Consensus sublayers.

The Storage and Ledger sublayer is provided to store the blockchain state, such as smart contracts for later execution.

The Execution sublayer implements the ***Ethereum Virtual Machine*** (EVM), which is a runtime computing environment for the execution of smart contracts. Each node operates an EVM.

Smart contracts are computer programs that the EVM executes. A ***precompiled contract*** is a smart contract compiled in EVM bytecode and stored by a node.

Finally, the Consensus sublayer provides a mechanism to establish consensus between nodes. ***Consensus*** is the process of nodes on a blockchain reaching agreement about the current state of the blockchain.

A ***consensus algorithm*** is the mechanism by which a blockchain achieves consensus. Different blockchains can use different consensus algorithms, but all nodes of a given blockchain need to use the same consensus algorithm.

8.1 Execution Sublayer

DOCUMT-010: Enterprise Ethereum blockchains *MUST* document any extension to the public Ethereum EVM op-code set [EVM-Opcodes] that can be used in smart contracts in the EEA Opcode Registry. See also **EXEC-020:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

8.1.1 Finality

Finality occurs when a transaction is definitively part of the blockchain and cannot be removed. A transaction reaches finality after some event defined for the relevant blockchain occurs. For example, an elapsed amount of time or a specific number of blocks added.

8.2 Consensus Sublayer

A common consensus algorithm implemented by all clients is required to ensure interoperability between clients.

9. Blockchain Configuration

CONFIG-010: Any limit on the size of smart contracts that can be deployed on an Enterprise Ethereum Blockchain *MUST* be specified by the `maxCodeSize` network configuration parameter, as defined in the section 9.1 The `maxCodeSize` network configuration parameter below. See also **SMRT-040:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

9.1 The `maxCodeSize` network configuration parameter

This section is experimental

The purpose of the ***`maxCodeSize`*** network configuration parameter is to specify a *limit* in kilobytes for the size of a smart contract that can be deployed by a transaction. A transaction to deploy a smart contract larger than the current *limit* is invalid.

The default value of the *limit* is implementation-dependent and determined by individual Enterprise Ethereum clients. It is **at least** 24 kilobytes.

Smart contracts that have already been deployed to the chain can be executed regardless of the current value of the *limit*. Deployed smart contracts can be stopped from operating through the permissioning contract.

The value of the `maxCodeSize` parameter is either an integer, specifying the *limit* directly, or a Javascript object, consisting of pairs of integers.

If the value is an object, for each pair of integers:

- the first number in the pair specifies the *limit*,
- the second number specifies the first block at which the associated *limit* applies.

A missing or non-integer value for the *limit* means the blockchain imposes the default value.

A negative value for the *limit* means the blockchain imposes no limit.

> NOTE
>
> Enterprise Ethereum clients can have an implementation-dependent limit, that is guaranteed to be **at least** 24 kilobytes.

A value of 0 for the *limit* means that any transaction to deploy a smart contract is invalid: no new smart contract can be added to the blockchain.

A missing, negative or non-integer value for the *block height* is an error, and clients will ignore any associated *limit*.

A value for the *block height* that is lower than a previous value is an error, and clients will ignore any associated *limit*.

EXAMPLE 1

Given the following value of maxCodeSize:

```
"maxCodeSize" : {
    48 : 5000,
    92 : 12750,
    -1 : 15000,
    256: 14000,
    256: -1000,
    256: 20000,
    0: 25000,
    "default": 30000
  }
```

- Since there is no value specified for the first 4999 blocks, the default limit is applied. This means that Transactions can deploy smart contracts of
- *at least** 24 kilobytes, with an unknown implementation-dependent limit imposed by nodes.
- Transactions to deploy smart contracts from blocks 5000 to 12749 are only valid if the smart contract they are deploying is not larger than 48 kilobytes.
- From blocks 12750 to 14999 there is a limit of 92 kilobytes applied.
- From block 15000 to block 19999 there is no specified limit. Enterprise Ethereum clients might be unable to process smart contracts because they are too large for the software, but are required to process smart contracts of **at least** 23576 bytes and can generally process much larger ones.
- The fourth and fifth lines are treated as errors, and have no effect.
- From block 20000 to 24999, smart contracts larger than 256kb bytes cannot be deployed.
- From block 25000 to 29999 no transaction to deploy a smart contract is valid.
- From block 30000, the implementation-dependent default limit (at least 24kb) will be applied again.

Note that the changes to the maxCodeSize *limit* only affect the size of smart contracts that can be **deployed**. Smart contracts already on the blockchain can still be executed, whatever their size.

INTROP-010: Enterprise Ethereum blockchains *MUST* use the Clique Proof of Authority consensus algorithm [EIP-225]. See also **CONS-093:** in the Enterprise Ethereum Alliance Client

Specification [EEA-client].

The Technical Specification Working Group expects to develop or identify at least one Byzantine Fault Tolerant Consensus algorithm, which could be used instead of Clique.

The ***genesis block*** is the first block of a blockchain.

A ***hard fork*** is a permanent divergence from the previous version of a blockchain. nodes running previous versions are no longer accepted by the newest version.

A ***hard fork block*** is the block that marks the start of a hard fork.

A. Additional Information

A.1 Summary of Requirements

This section provides a summary of all requirements in this Specification.

ORGIDS-300: Enterprise Ethereum Blockchains *MAY* implement a smart contract based Organization Registry that provides cryptographic bindings between Ethereum accounts and their owning organization with identity proofs.

ORGIDS-310: An Organization Registry *MUST* require all root entries to present an identity proof with:

- a signing authority attesting that the proof has been uniquely issued for the organization identified by the subject
- a digital signature generated by the private key for the root signing account

ORGIDS-320: An Organization Registry *MUST* require the root signing account for an organization to insert child entries under that organization.

ORGIDS-330: An Organization Registry *MUST* support an extensibility mechanism to allow different types of proofs to be submitted and verified.

ORGIDS-340: An Organization Registry *MUST* support at least one of the following proof types:

- X.509 Certificates [rfc5280] generated by a trusted Certificate Authority (CA), attached to a chain of intermediate CAs leading up to a globally recognized root CA.

- A Verifiable Presentations data structure [VC-presentations] as defined within the W3C Verifiable Credentials Data Model [vc-data-model].

Verifiable Credentials is a new W3C standard in the Decentralized Identifier (DID) ecosystem. The Verifiable Credentials data model is not itelf suitable as a proof type because it does not support chain-unique challenges for replay attack protections.

ORGIDS-350: An Organization Registry *MAY* verify the proofs in the smart contract and immediately reject a registration that did not present a valid proof, or allow a registration to be validated or invalidated by an off-chain agent.

ORGIDS-360: If an Organization Registry performs proof validation in the smart contract, it *MUST* offer at least the following options to support different "freshness" requirements:

- validate once during registration, rely on administration operations to update expired or revoked proofs
- validate during registration, replicate expiration date in the contract for faster checking subsequently
- validate every time the account is used

ORGIDS-370: Identity proofs *MUST* protect against re-use by a malicious party, by embedding a chain-unique challenge segment, such as the chain ID, in the signed claims inside the proof.

ORGIDS-380: A Registry for Organizational Accounts *MUST* not allow a registered proof to be used to register a new root entry.

PERMIT-020 If the permissions for a blockchain are updated to revoke any permission previously granted to nodes, the node permissioning contract *MUST* emit a `NodePermissionsUpdated` event containing an `addsRestrictions` property with the value `true`. See also **PERM-220:**.

PERMIT-030 If the permissions for a blockchain are updated to grant any new permissions for nodes the node permissioning contract *MUST* emit a `NodePermissionsUpdated` event containing an `addsRestrictions` property with the value `false`. See also **PERM-230:**.

PERMIT-070 The node connection rules *MUST* support both the IPv4 and IPv6 protocol versions.

CONFIG-040: A node permissioning contract with the `connectionAllowed` function as defined in section 6.3.1.1.1 Node Permissioning Functions, *MUST* be included in the genesis block (block 0), available at the address specified in the network configuration parameter `nodePermissionContract`.

PERMIT-090 Account permissioning contracts *MUST* respond with a `bool` value of `true` for the case where the transaction is allowed, or `false` for the case where the transaction is not allowed.

CONFIG-050 A permissioning contract with the `transactionAllowed` function as defined in section 6.3.1.2.1 Account Permissioning Smart Contract Interface Function, *MUST* be included in the genesis block (block 0), available at the address specified in the network configuration parameter `transactionPermissionContract`.

PERMIT-010: An Enterprise Ethereum blockchain account permissioning contract *MUST* enable whitelisting accounts that are permitted to interact with the blockchain. See also **PART-010:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

PERMIT-060: The account permissioning contract *MUST* manage separate permissioning for an account to:

- Deploy smart contracts.
- Call functions that change the state of specified smart contracts.
- Perform a value transfer to a specified account.

PERMIT-080 Permissioning contracts that are updateable *MUST NOT* allow changes through a private transaction.

DOCUMT-010: Enterprise Ethereum blockchains *MUST* document any extension to the public Ethereum EVM op-code set [EVM-Opcodes] that can be used in smart contracts in the EEA Opcode Registry. See also **EXEC-020:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

CONFIG-010: Any limit on the size of smart contracts that can be deployed on an Enterprise Ethereum Blockchain *MUST* be specified by the `maxCodeSize` network configuration parameter, as defined in the section 9.1 The `maxCodeSize` network configuration parameter below. See also **SMRT-040:** in the Enterprise Ethereum Alliance Client Specification [EEA-client].

INTROP-010: Enterprise Ethereum blockchains *MUST* use the Clique Proof of Authority consensus algorithm [EIP-225]. See also **CONS-093:** in the Client Specification.

A.2 Defined Terms

The following is a list of terms defined in this Specification.

account permissioning contract

consensus

consensus algorithm

ĐApps

Enterprise Ethereum

Enterprise Ethereum blockchain

Enterprise Ethereum client

Ethereum account

Ethereum JSON-RPC API

Ethereum MainNet

Ethereum Name Service

Ethereum Virtual Machine

experimental

finality

gas

genesis block

groups

hard fork

hard fork block

identity proof

interoperate

layer 1

layer 2

network configuration

node

node permissioning contract

off-chain trusted computing

organization

organization registry

participating organization

permissioning

permissioning contract

A.3 Acknowledgments

The EEA acknowledges and thanks the many people who contributed to the development of this version of the specification. Please advise us of any errors or omissions.

This version builds on the work of all who contributed to previous versions of the Enterprise Ethereum Client Specification, whom we hope are all acknowledged in those documents. We apologize to anyone whose name was left off the list. Please advise us at https://entethalliance.org/contact/ of any errors or omissions.

Enterprise Ethereum is built on top of Ethereum, and we are grateful to the entire community who develops Ethereum, for their work and their ongoing collaboration to helps us maintain as much compatibility as possible with the Ethereum ecosystem.

A.4 Legal Notice

B. References

B.1 Normative references

[EIP-225]
Clique proof-of-authority consensus protocol. Ethereum Foundation. URL: https://eips.ethereum.org/EIPS/eip-225

[EIP-648]
Easy Parallelizability. Ethereum Foundation. URL: https://github.com/ethereum/EIPs/issues/648

[EVM-Opcodes]
Ethereum Virtual Machine (EVM) Opcodes and Instruction Reference. URL: https://github.com/trailofbits/evm-opcodes

[GDPR]
European Union General Data Protection Regulation. European Union. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1528874672298&uri=CELEX%3A32016R0679

[JSON-RPC]

JavaScript Object Notation - Remote Procedure Call. JSON-RPC Working Group. URL: http://www.jsonrpc.org/specification

[Plasma]
Plasma: Scalable Autonomous Smart Contracts. Joseph Poon and Vitalik Buterin. August 2017. URL: https://plasma.io/plasma.pdf

[PSD2]
European Union Personal Service Directive. European Union. URL: https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[rfc5280]
Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. D. Cooper; S. Santesson; S. Farrell; S. Boeyen; R. Housley; W. Polk. IETF. May 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5280

[RFC8174]
Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8174

[sharding]
Sharding FAQs. Ethereum Foundation. URL: https://github.com/ethereum/wiki/wiki/Sharding-FAQs

[state-channels]
Counterfactual: Generalized State Channels. URL: https://counterfactual.com/statechannels

[vc-data-model]
Verifiable Credentials Data Model 1.0. Manu Sporny; Grant Noble; Dave Longley; Daniel Burnett; Brent Zundel. W3C. 19 November 2019. W3C Recommendation. URL: https://www.w3.org/TR/vc-data-model/

[VC-presentations]
Verifiable presentations', section in 'Verifiable Credentials Data Model'. W3C. URL: https://www.w3.org/TR/2019/REC-vc-data-model-20191119/#dfn-verifiable-presentations

[ZK-STARKS]
Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive. 2018-03-16. URL: https://eprint.iacr.org/2018/046.pdf

[ZKP]
Zero Knowledge Proof. Wikipedia. URL: https://en.wikipedia.org/wiki/Zero-knowledge_proof

B.2 Informative references

[Byzantine-Fault-Tolerant]
Byzantine Fault Tolerant. URL: https://en.wikipedia.org/wiki/Byzantine_fault_tolerance

[EEA-client]
Enterprise Ethereum Alliance Client Specification v5. URL: https://entethalliance.org/wp-content/uploads/2020/05/EEA_Enterprise_Ethereum_Client_Specification_V5.pdf

[EEA-implementation-guide]
Enterprise Ethereum Alliance Implementation Guide (Work in Progress). Enterprise Ethereum Alliance, Inc. URL: https://entethalliance.github.io/client-spec/implementing.html

[EEA-OC]
Enterprise Ethereum Alliance Off-Chain Trusted Compute Specification v1.1. Enterprise Ethereum Alliance, Inc. URL: https://entethalliance.org/wp-content/uploads/2019/11/EEA_Off-Chain_Trusted_Compute_Specification_v1.1.pdf

[EIPs]
Ethereum Improvement Proposals. Ethereum Foundation. URL: https://eips.ethereum.org/

[ERC-20]
Ethereum Improvement Proposal 20 - Standard Interface for Tokens. Ethereum Foundation. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[ERC-223]
Ethereum Improvement Proposal 223 - Token Standard. Ethereum Foundation. URL: https://github.com/ethereum/EIPs/issues/223

[ERC-621]
Ethereum Improvement Proposal 621 - Token Standard Extension for Increasing & Decreasing Supply. Ethereum Foundation. URL: https://github.com/ethereum/EIPs/pull/621

[ERC-721]
Ethereum Improvement Proposal 721 - Non-fungible Token Standard. Ethereum Foundation. URL: https://github.com/ethereum/eips/issues/721

[ERC-827]
Ethereum Improvement Proposal 827 - Extension to ERC-20. Ethereum Foundation. URL: https://github.com/ethereum/EIPs/issues/827

[Ethereum-Yellow-Paper]
Ethereum: A Secure Decentralized Generalized Transaction Ledger. Dr. Gavin Wood. URL: https://ethereum.github.io/yellowpaper/paper.pdf

[Function-collision]
Malicious backdoors in Ethereum Proxies. Patricio Palladino. URL: https://medium.com/nomic-labs-blog/malicious-backdoors-in-ethereum-proxies-62629adf3357

[Layer2-Scaling-Solutions]
Making Sense of Ethereum's Layer 2 Scaling Solutions: State Channels, Plasma, and Truebit. Josh Stark. February 2018. URL: https://medium.com/l4-media/making-sense-of-

ethereums-layer-2-scaling-solutions-state-channels-plasma-and-truebit-22cb40dcc2f4

[Upgrade-contracts]
Building Upgradeable Smart Contracts. Andrea Di Nenno, Clearmatics. URL: https://medium.com/clearmatics/upgrading-smart-contracts-c9fb144eceb7

[WP-ABAC]
Attribute-based access control. Wikipedia. URL: https://en.wikipedia.org/wiki/Attribute-based_access_control

[WP-RBAC]
Role-based access control. URL: https://en.wikipedia.org/wiki/Role-based_access_control

↑