Enterprise Ethereum Alliance Client Specification V3

13 May 2019

Latest editor's draft:

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

Editors:

Robert Coote (PegaSys)

Chaals Nevile (Enterprise Ethereum Alliance)

Grant Noble (PegaSys)

George Polzer (Everymans.ai)

Former editors:

Daniel Burnett (PegaSys)

David Hyland-Wood (PegaSys)

Contributors to this version:

Duarte Aragao (Clearmatics), Sanjay Bakshi (Intel), Imran Bashir (JP Morgan), Thomas Bertani (Provable Things), Jean-Charles Cabalguen (iExec), Zak Cole (Whiteblock), Rob Dawson (PegaSys), Samer Falah (JP Morgan), Julio Fauro (Adhara), Sara Feenan (Clearmatics), Bill Gleim (ConsenSys), Mark Grand (HLC), Marley Gray (Microsoft), Daniel Heyman (PegaSys), Kieren James-Lubin (BlockApps), Faisal Khan (PegaSys), Ivaylo Kirilov (Web3 Labs), Sally MacFarlane (PegaSys), Chris McKay (PegaSys), Boris Mann (SPADE), Denis Milecevic (Provable Things), Horacio Mijail (PegaSys), Mike Myers (Trail of Bits), Immad Naseer (Microsoft), Alex Oberhauser (Cambridge Blockchain), George Ornbo (Clearmatics), Fernando Paris (IOBuilders), Dhyan Raj (Synechron), Peter Robinson (PegaSys), Hector Rodes Lopez (Adhara), José Aurelio Rodrigo (DEKRA), Peter de Rooij (Accenture), Lior Saar (BlockApps), Lucas Saldanha (PegaSys), Roberto Saltini (PegaSys), Joseph Schweitzer (Ethereum Foundation), Felix Shnir (JP Morgan), Przemek Siemion (Santander), Adrian Sutton (PegaSys), Conor Svensson (Web3 Labs), Krishnakumar Swaminathan (JP Morgan), Antoine Toulme (ConsenSys), John Whelan (Santander), Victor Wong (BlockApps), Tom Willis (Intel), Yevgeniy 'Eugene' Yarmosh (Intel), Jim Zhang (ConsenSys)

Abstract

This document, the Enterprise Ethereum Alliance Client Specification, defines the implementation requirements for Enterprise Ethereum clients, including the interfaces to external-facing components of Enterprise Ethereum and how they are intended to be used.

1. Legal Notice

The copyright in this document is owned by Enterprise Ethereum Alliance. Inc. (“EEA” or “Enterprise Ethereum Alliance”).

No modifications, edits or changes to the information in this document are permitted. Subject to the terms and conditions described herein, this document may be duplicated for internal use, provided that all copies contain all proprietary notices and disclaimers included herein. Except as otherwise provided herein, no license, express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.

Use of this document and any related intellectual property incorporated herein, is also governed by the Bylaws, Intellectual Property Rights Policy and other governing documents and policies of EEA and is subject to the disclaimers and limitations described below.

No use or display of any of the following names or marks "Enterprise Ethereum Alliance", the acronym "EEA", the EEA logo, or any combination thereof, to claim compliance with or conformance to this document (or similar statements) is permitted absent EEA membership and express written permission from the EEA. The EEA is in process of developing a compliance testing and certification program only for the EEA members in good standing, which it expects to launch in second half of 2020.

THE CONTENTS OF THIS DOCUMENT ARE PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, SATISFACTORY QUALITY, OR REASONABLE SKILL OR CARE, OR ANY WARRANTY ARISING OUT OF ANY COURSE OF DEALING, USAGE, TRADE PRACTICE, PROPOSAL, SPECIFICATION OR SAMPLE. EEA DOES NOT WARRANT THAT THIS DOCUMENT IS COMPLETE OR WITHOUT ERROR AND DISCLAIMS ANY WARRANTIES TO THE CONTRARY.

Each user of this document hereby acknowledges that sofftware or products implementing the technology specified in this document ("EEA-Compliant Products") may be subject to various regulatory controls under the laws and regulations of various governments worldwide. Such laws and regulatory controls may govern, among other things, the combination, operation, use, implementation and distribution of EEA-Compliant Products. Examples of such laws and regulatory controls include, but are not limited to, airline regulatory controls, telecommunications regulations, finance industry and security regulations, technology transfer controls, health and safety and other types of regulations. Each user of this document is solely responsible for the compliance by their EEA-Compliant Products with any such laws and regulations and for obtaining any and all required authorizations, permits, or licenses for their EEA-Compliant Products related to such regulations within the applicable jurisdictions. Each user of this document acknowledges that nothing in this document or the relevant specification provides any information or assistance in connection with securing such compliance, authorizations or licenses. NOTHING IN THIS

DOCUMENT CREATES ANY WARRANTIES WHATSOEVER REGARDING THE APPLICABILITY OR NON-APPLICABILITY OF ANY SUCH LAWS OR REGULATIONS OR THE SUITABILITY OR NON-SUITABILITY OF ANY SUCH PRODUCT OR SERVICE FOR USE IN ANY JURISDICTION.

EEA has not investigated or made an independent determination regarding title or non-infringement of any technologies that may be incorporated, described or referenced in this document. Use of this document or implementation of any technologies described or referenced herein may therefore infringe undisclosed third-party patent rights or other intellectual property rights. The user is solely responsible for making all assessments relating to title and non-infringement of any technology, standard, or specification referenced in this document and for obtaining appropriate authorization to use such technologies, standards, and specifications, including through the payment of any required license fees.

NOTHING IN THIS DOCUMENT CREATES ANY WARRANTIES OF TITLE OR NONINFRINGEMENT WITH RESPECT TO ANY TECHNOLOGIES, STANDARDS OR SPECIFICATIONS REFERENCED OR INCORPORATED INTO THIS DOCUMENT.

IN NO EVENT SHALL EEA OR ANY OF ITS MEMBERS BE LIABLE TO THE USER OR TO A THIRD PARTY FOR ANY CLAIM ARISING FROM OR RELATING TO THE USE OF THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, A CLAIM THAT SUCH USE INFRINGES A THIRD PARTY'S INTELLECTUAL PROPERTY RIGHTS OR THAT IT FAILS TO COMPLY WITH APPLICABLE LAWS OR REGULATIONS. BY USE OF THIS DOCUMENT, THE USER WAIVES ANY SUCH CLAIM AGAINST EEA AND ITS MEMBERS RELATING TO THE USE OF THIS DOCUMENT.

EEA reserves the right to adopt any changes or alterations to this document as it deems necessary or appropriate without any notice. User is solely responsible for determining whether this document has been superseded by a later version or a different document.

Status of This Document

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

This is the Enterprise Ethereum Alliance Client Specification, version 3. Changes made since version 2 of the Specification, released on 15 October 2018, have been reviewed by the Enterprise Ethereum Alliance (EEA) Technical Specification Working Group (TSWG), and the EEA Board.

The TSWG *expects* at time of writing that the next revision of this Specification will be released in the final quarter of 2019, obsoleting this version.

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:

- Improved privacy management.
- Stronger requirements for interoperability as important components of the ecosystem become more generally interoperable.
- Adoption of improvements to the Ethereum ecosystem, such as new technologies or techniques.
- Continued assessment of the needs of different industries to ensure their requirements for Enterprise Ethereum are taken into account.

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

Table of Contents

2. Introduction

This section is non-normative.

This document, the Enterprise Ethereum Alliance Client Specification, defines the implementation requirements for Enterprise Ethereum clients, including the interfaces to external-facing components of Enterprise Ethereum and how they are intended to be used.

For the purpose of this Specification:

- An ***organization*** is a logical group composed of Ethereum accounts, nodes, and organizations. It typically represents an enterprise, or some identifiable part of an enterprise. For the purpose of permissioning, an organization roughly corresponds to the Unix concept of a group.
- ***Public Ethereum*** (or "Ethereum") is the public blockchain-based distributed computing platform featuring smart contract (programming) functionality defined by the Ethereum Yellow Paper [Ethereum-Yellow-Paper], EIPs [EIPs], and associated specifications.

- ***Enterprise Ethereum*** is the set of enterprise-focused extensions to public Ethereum defined in this Specification. These extensions provide the ability to perform private transactions, and enforce permissioning, for Ethereum blockchains that use them. Such blockchains are known as ***Enterprise Ethereum blockchains***.
- An ***Enterprise Ethereum client*** (a client) is the software that implements Enterprise Ethereum, and is used to run nodes on an Enterprise Ethereum blockchain.
- A ***node*** is an instance of an Enterprise Ethereum client running on an Enterprise Ethereum blockchain.

> NOTE
>
> Multiple clients might run on an individual device, or a client might run on a cloud service.

2.1 Why Produce a Client Specification?

With a growing number of vendors developing Enterprise Ethereum clients, meeting the requirements outlined in this Client Specification ensures different clients can communicate with each other and interoperate reliably on a given Enterprise Ethereum blockchain.

For ĐApp developers, for example, a Client Specification ensures clients provide a set of identical interfaces so that ĐApps will work on all conforming clients. This enables an ecosystem where users can change the software they use to interact with a running blockchain, instead of being forced to rely on a single vendor to provide support.

From the beginning, this approach has underpinned the development of Ethereum, and it meets a key need for blockchain use in many enterprise settings.

Client diversity also provides a natural mechanism to help verify that the protocol specification is unambiguous because interoperability errors revealed in development highlight parts of the protocol that different engineering teams interpret in different ways.

Finally, standards-based interoperability allows users to leverage the widespread knowledge of Ethereum in the blockchain development community to minimize the learning curve for working with Enterprise Ethereum, and thus reduces risk when deploying an Enterprise Ethereum blockchain.

3. 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*, *SHOULD*, and *SHOULD NOT* are to be interpreted as described in [RFC2119].

3.1 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 your comments and feedback on the experimental portions of this Specification to the EEA Technical Steering Committee at https://entethalliance.org/contact/.

3.2 Requirement Categorization

All requirements in this Specification are categorized as either:

- ***Protocol requirements***, which are denoted by **[P]** prefixed to the requirement ID.

  Protocol requirements are requirements where the desired properties and correctness of the system can be jeopardized unless all clients implement the requirement correctly.

- ***Client requirements***, which are denoted by **[C]** prefixed to the requirement ID.

  Client requirements do not impact global system behavior, but if not implemented correctly in a client, that client might not function correctly, or to a desirable level, in an Enterprise Ethereum blockchain.

> Example: Requirement Categorization
>
> **[C] JRPC-050:** Enterprise Ethereum clients *MUST* implement the [JSON-RPC-PUB-SUB] API.
>
> Requirement **JRPC-050** is a client requirement, which if not implemented correctly, does not disrupt the correct functioning of an Enterprise Ethereum blockchain.
>
> **[P] SMRT-030:** Enterprise Ethereum clients *MUST* support smart contracts of at least 24,576 bytes in size.
>
> Requirement **SMRT-030** is a protocol requirement. Running a client that does not implement this requirement on an Enterprise Ethereum blockchain risks causing an error in the functioning of the blockchain.

4. 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, in particular, are especially important in an organizational environment.

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.

4.1 Attacks on Enterprise Ethereum

Modeling attacks against a node helps identify and prioritize the necessary security countermeasures to implement. Some attack categories to consider include:

- Attacks on unauthenticated [JSON-RPC] interfaces through malicious JavaScript in the browser using DNS rebinding.
- Eclipse attacks (attacks targeting specific nodes in a decentralized network) that attempt to exhaust client network resources or fool its node-discovery protocol.
- Targeted exploitation of consensus bugs in EVM implementations.
- Malicious code contributions to open-source repositories.
- All varieties of social engineering attacks.

4.2 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 plays some role in mitigating network-level attacks (like the 51% attack), but it is important to carefully consider which risks are of most concern to a client implementation versus those that are better mitigated by updates to the Ethereum consensus protocol design.

4.3 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 on any public platform (such as Ethereum) exposes it to eventual decryption by brute-force attack, accelerated by the inevitable periodic advances in cryptanalysis. A future shift to post-quantum cryptography is a current concern, but it will likely not be the last advancement 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.

Besides user-generated data, a client is also responsible for managing and protecting private keys. Encrypting private keys with a passphrase or other authentication credential before storage helps protect them from disclosure. It is also important not to disclose sensitive data when recording events to a log file.

4.4 Security of Client Implementations

There are several specific functionality areas that are more prone to security issues arising from implementation bugs. The following areas deserve a greater focus during the design and the security assessment of an Enterprise Ethereum client:

- Peer-to-peer protocol implementation
- Object deserialization routines
- Ethereum Virtual Machine (EVM) implementation
- Key pair generation.

The peer-to-peer protocol used for communication among nodes in Ethereum is a client's primary vector for exposure to untrusted input. In any software, the program logic that handles untrusted inputs is the primary focus area for implementing secure data handling.

Object serialization and deserialization is commonly part of the underlying implementation of the P2P protocol, but also a source of complexity that, historically, is prone to security vulnerabilities across many implementations and many programming languages. Selecting a deserializer that offers strict control of data typing can help mitigate the risk.

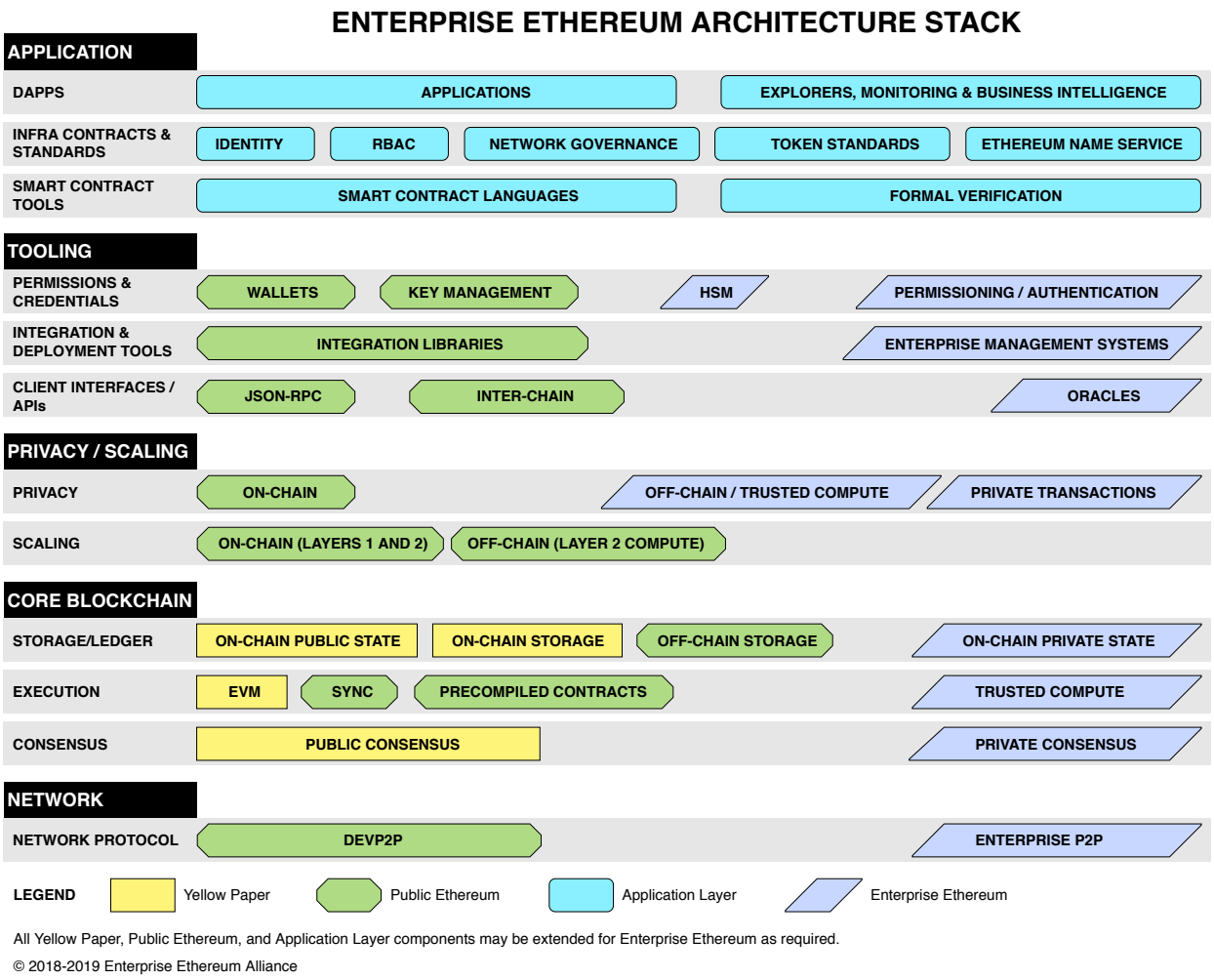
EVM implementation correctness is an especially important security consideration for clients. Unless EVMs behave identically for all possibilities of input, there is a serious risk of a hard fork caused by an input that elicits the differences in behavior across clients. EVM implementations are also exposed to denial-of-service attempts by maliciously constructed smart contracts, and the even more serious risk of an exploitable remote-code-execution vulnerability.

The Ethereum specification defines many of the technical aspects of public/private key pair format and cryptographic algorithm choice, but a client implementation is still responsible for properly generating these keys using a well-reviewed cryptographic library. Specifically, a client implementation needs a properly seeded, cryptographically secure, pseudo-random number generator (PRNG) during the keypair generation step. An insecure PRNG is not generally apparent by merely observing its outputs, but enables attackers to break the encryption and reveal users' sensitive data.

5. Enterprise Ethereum Architecture

This section is non-normative.

The following two diagrams show the relationship between Enterprise Ethereum components that can be part of any Enterprise Ethereum client implementation. The first is a stack representation of the architecture showing a library of interfaces, while the second is a more traditional style architecture diagram showing a representative architecture.



All Yellow Paper, Public Ethereum, and Application Layer components may be extended for Enterprise Ethereum as required.

Figure 1 Enterprise Ethereum Architecture Stack

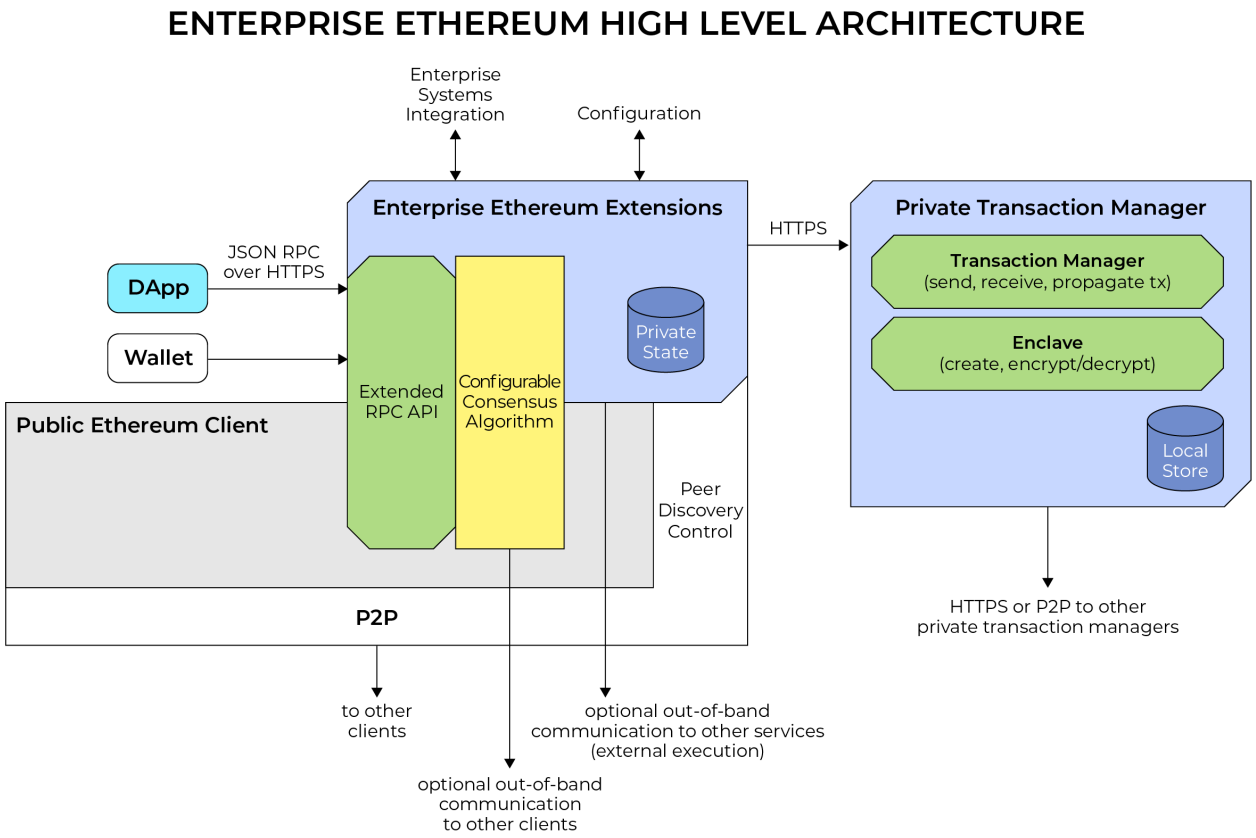


Figure 2 Representative Enterprise Ethereum High-level Architecture

The architecture stack for Enterprise Ethereum consists of five layers:

- Application
- Tooling
- Privacy and Scaling
- Core Blockchain
- Network.

These layers are described in the following sections.

6. Application Layer

The Application layer exists, often fully or partially outside of a client, where higher-level services are provided. For example, Ethereum Name Service (ENS), node monitors, blockchain state

visualizations and explorers, self-sovereign and other identity schemes, wallets, and any other applications of the ecosystem envisaged.

Wallets are software applications used to store an individual's credentials (cryptographic private keys), which are associated with the state of that user's Ethereum account.

Wallets can interface with Enterprise Ethereum using the Extended RPC API, as shown in Figure 2. A wallet can also interface directly with the enclave of a private transaction manager, or interface with public Ethereum.

A ***private transaction manager*** is a subsystem of an Enterprise Ethereum system for implementing privacy and permissioning.

6.1 ĐApps Sublayer

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

[C] DAPP-010: ĐApps *MAY* use the extensions to the Ethereum JSON-RPC API defined in this Specification.

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

6.2 Infrastructure Contracts and Standards Sublayer

This section is non-normative.

The Infrastructure Contracts and Standards sublayer shows emerging standards outside this Specification. The components in this sublayer provide enablers for the applications built on top of them.

Role Based Access Control (RBAC) defines methods for authentication and restricting system access to authorized nodes and accounts. This can be realized through smart contracts such as the permissioning Contracts in Section § 7.3.3.3 Permissioning management: examples.

Network governance methods control which entities can join the network and hence assist with safeguarding exchanges.

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.

6.3 Smart Contract Tools Sublayer

Enterprise Ethereum inherits the smart contract tools used by public Ethereum. These tools include smart contract languages and associated parsers, compilers, and debuggers, as well as methods used for formal verification of smart contracts.

Enterprise Ethereum implementations enable use of these tools and methods through implementation of the Execution sublayer, as described in Section § 9.2 Execution Sublayer.

[P] SMRT-030: Enterprise Ethereum clients *MUST* support smart contracts of at least 24,576 bytes in size.

[P] SMRT-040: Enterprise Ethereum clients *MUST* read and enforce a size limit for smart contracts from the current network configuration (e.g. the genesis block).

[P] SMRT-050: If no contract size limit is specified in a genesis block, subsequent hard fork block or network configuration, Enterprise Ethereum clients *MUST* enforce a size limit on smart contracts of 24,576 bytes.

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

A ***hard fork*** is a permanent divergence from the previous version of a blockchain, and nodes running previous versions are no longer accepted by the newest version. All nodes that are meant to work in accordance with the new version of the blockchain must upgrade their software.

A ***hard fork block*** is the block from which a hard fork occured.

7. 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, deploy smart contracts, and to allow ĐApps and wallets to interact with the platform. 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.

Integration libraries, such as [web3j], [web3.js], and [Nethereum], are software libraries used to implement APIs with different language bindings (like the Ethereum JSON-RPC API) for interacting with Ethereum nodes.

Public Ethereum nodes can choose to offer local handling of user credentials, such as key management systems and wallets. Such facilities might also be implemented outside the scope of a client.

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

The Tooling layer also provides support for the compilation, and possibly formal verification of, smart contracts through the use of parsers and compilers for one or more smart contract languages.

Smart contract languages are the programming languages and associated tooling used to create smart contracts.

Formal verification is the mathematical verification of the logical correctness of a smart contract in the context of the EVM.

Smart contract languages such as [Solidity] and [LLL] are commonly implemented, but support for other languages might be provided without restriction.

7.1 Permissions and Credentials Sublayer

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 only allow 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.

7.1.1 Nodes

[C] NODE-010: Enterprise Ethereum implementations *MUST* provide the ability to specify at startup a list of static peer nodes to establish peer-to-peer connections with.

[C] NODE-020: Enterprise Ethereum clients *MUST* provide the ability to enable or disable peer-to-peer node discovery.

[P] NODE-030: Enterprise Ethereum clients *MUST* provide the ability to specify a whitelist of the nodes permitted to connect to a node.

[P] NODE-040: Enterprise Ethereum clients *MAY* provide the ability to specify a blacklist of the nodes not permitted to connect to a node.

[P] NODE-050: It *MUST* be possible to specify the node whitelist required by **NODE-030** through a transaction into a smart contract.

[P] NODE-060: It *MUST* be possible to specify the node blacklist allowed by **NODE-040** (if implemented) through a transaction into a smart contract.

[P] NODE-080: Enterprise Ethereum clients *MUST* provide the ability to specify node identities in a way aligned with the concept of groups.

[P] NODE-090: Enterprise Ethereum clients *MUST* document which metadata parameters (if any) can affect transaction ordering, and what the effects are.

7.1.2 Ethereum Accounts

For the purpose of this Specification:

- A ***User*** is a human or an automated process interacting with Enterprise Ethereum through the Ethereum JSON-RPC API. The identity of a user is represented by an Ethereum account. Public key cryptography is used to sign transactions so the EVM can authenticate the identity of the 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 Ethereum, for example to submit transactions or deploy smart contracts. See also Wallet.
- ***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.
- ***Ethereum MainNet*** is the public Ethereum blockchain whose `chainid` and `network ID` are both `1`.

[P] PART-010: Enterprise Ethereum clients *MUST* provide the ability to specify a whitelist of accounts that are permitted to transact with the blockchain.

[P] PART-015: Enterprise Ethereum clients *MUST* be able to verify that accounts are present on the whitelist required by **PART-010:** when adding transactions from the account to a block, and when verifying a received block containing transactions created by that account.

[P] PART-020: Enterprise Ethereum clients *MAY* provide the ability to specify a blacklist of accounts that are not permitted to transact with the blockchain.

[P] PART-025: Enterprise Ethereum clients *MUST* be able to verify that accounts are not present on the blacklist allowed by **PART-020:** (if implemented) when adding transactions from the account to a block, and when verifying a received block containing transactions created by that account.

[P] PART-030: It *MUST* be possible to specify the account whitelist required by **PART-010:** through a transaction into a smart contract.

[P] PART-040: It *MUST* be possible to specify the account blacklist allowed by **PART-020:** (if implemented) through a transaction into a smart contract.

[P] PART-050: Enterprise Ethereum clients *MUST* provide a mechanism to identify organizations that participate in the Enterprise Ethereum blockchain.

> NOTE
>
> A specific mechanism could be identified in a future version of this specification.

[P] PART-055 Enterprise Ethereum clients *MUST* support anonymous accounts.

[P] PART-060: Enterprise Ethereum clients *MUST* provide the ability to specify accounts in a way aligned with the concepts of groups and roles.

[P] PART-070: Enterprise Ethereum clients *MUST* be able to authorize the types of transactions an account can submit, providing separate permissioning for the ability to:

- Deploy smart contracts.
- Call functions that change the state of specified smart contracts.
- Perform a simple value transfer between specified accounts.

> NOTE
>
> Since deep nesting of structures can introduce unacceptable performance issues, implementations can limit how many levels of nesting they actually enable. This specification defines a bare minimum requirement, although in practice the number of levels implementations support is not constrained to any paticular level, and depends entirely on implementation choices.

[C] PERM-075: Enterprise Ethereum clients *MUST* allow organizations to be nested to a minimum of at least 3 levels (i.e. an organization containing an organization that contains another organization).

7.1.3 Additional Permissioning Requirements

[C] PERM-020: Enterprise Ethereum clients *SHOULD* provide the ability for network configuration to be updated at run time without the need to restart.

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.

[C] PERM-040: Enterprise Ethereum clients *MAY* support local key management allowing users to secure their private keys.

[C] PERM-050: Enterprise Ethereum clients *MAY* support secure interaction with an external key management system for key generation and secure key storage.

Implementations could securely interact with a ***Hardware Security Module*** (HSM), which is a physical device to provide strong and secure key generation, key storage, and cryptographic processing for deployments where higher security levels are needed.

7.2 Integration and Deployment Tools Sublayer

Enterprise Management Systems

Many software systems implemented in organizations today provide the ability to integrate with enterprise management systems using common APIs, libraries, and techniques, as shown in Figure 3.

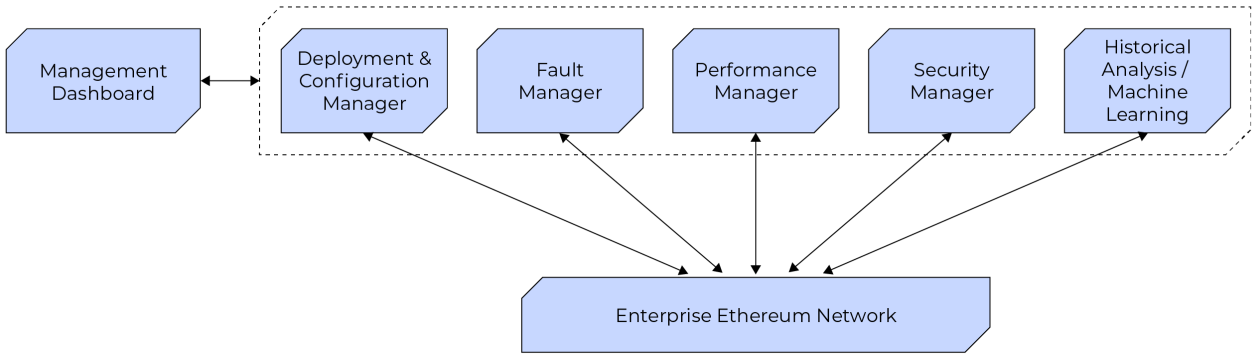


Figure 3 Management Interfaces

In addition to deployment and configuration capabilities, Enterprise Ethereum clients can offer possibilities such as software fault reporting, performance management, security management, integration with other enterprise software, and historical analysis tools.

These are not requirements of this specification, but features that different software can offer as part of what distinguishes it from other Enterprise Ethereum clients.

7.3 Client Interfaces and APIs Sublayer

As part of the Client Interfaces and APIs sublayer, [JSON-RPC-API] is a stateless, light-weight remote procedure call (RPC) protocol using [JSON] as its data format. The [JSON-RPC] specification defines several data structures and the rules around their processing.

A Ethereum JSON-RPC API is used to communicate between ĐApps and nodes.

7.3.1 Compatibility with the Core Ethereum JSON-RPC API

[P] JRPC-010: Enterprise Ethereum clients *MUST* provide support for the following methods of the Ethereum JSON-RPC API:

- `net_version`
- `net_peerCount`
- `net_listening`
- `eth_protocolVersion`
- `eth_syncing`
- `eth_coinbase`
- `eth_hashrate`
- `eth_gasPrice`
- `eth_accounts`
- `eth_blockNumber`
- `eth_getBalance`
- `eth_getStorageAt`
- `eth_getTransactionCount`
- `eth_getBlockTransactionCountByHash`
- `eth_getBlockTransactionCountByNumber`
- `eth_getCode`

- `eth_sendRawTransaction`
- `eth_call`
- `eth_estimateGas`
- `eth_getBlockByHash`
- `eth_getBlockByNumber`
- `eth_getTransactionByHash`
- `eth_getTransactionByBlockHashAndIndex`
- `eth_getTransactionByBlockNumberAndIndex`
- `eth_getTransactionReceipt`
- `eth_getUncleByBlockHashAndIndex`
- `eth_getUncleByBlockNumberAndIndex`
- `eth_getLogs`.

[P] JRPC-007: Enterprise Ethereum clients *SHOULD* implement [JSON-RPC-API] methods to be backward compatible with the definitions given in version 27e37ee of the Ethereum JSON-RPC API reference [JSON-RPC-API-v27e37ee], unless breaking changes have been made and widely implemented for the health of the ecosystem. For example, to fix a major security or privacy problem.

[C] JRPC-015: Enterprise Ethereum clients *MUST* provide the capability to accept and respond to JSON-RPC method calls over a websocket interface.

[C] JRPC-040: Enterprise Ethereum clients *MUST* provide an implementation of the `debug_traceTransaction` method [debug-traceTransaction] from the Go Ethereum Management API.

[C] JRPC-050: Enterprise Ethereum clients *MUST* implement the [JSON-RPC-PUB-SUB] API.

[P] JRPC-070: Enterprise Ethereum clients implementing additional nonstandard subscription types for the [JSON-RPC-PUB-SUB] API *MUST* prefix their subscription type names with a namespace prefix other than `eea_`.

7.3.2 Extensions to the JSON-RPC API

This section is experimental.

[P] JRPC-080: The [JSON-RPC] method name prefix `eea_` *MUST* be reserved for future use for RPC methods specific to the EEA.

[P] JRPC-020: Enterprise Ethereum clients *MUST* provide one of the following sets of extensions to create private transaction types defined in Section § 8.1.3 Private Transactions:

- `eea_sendTransactionAsync` and `eea_sendTransaction`, or
- `eea_sendRawTransactionAsync` and `eea_sendRawTransaction`.

[P] JRPC-030: The `eea_sendTransactionAsync`, `eea_sendTransaction`, `eea_sendRawTransactionAsync`, and `eea_sendRawTransaction` methods *MUST* respond with a [JSON-RPC] error response when an unimplemented private transaction type is requested. The error response *MUST* have the *code* `-50100` and the *message* `Unimplemented private transaction type`.

Example response

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -50100,
    "message": "Unimplemented private transaction type"
  }
}
```

7.3.2.1 eea_sendTransactionAsync

A call to `eea_sendTransactionAsync` creates a private transaction, signs it, submits it to the transaction pool, and returns immediately.

Using this method allows sending many transactions without waiting for recipient confirmation.

> NOTE
>
> As in the public Ethereum [JSON-RPC-API], the two key datatypes for this call, which are passed hex encoded, are unformatted data byte arrays (DATA) and quantities (QUANTITY). When encoding unformatted data, encode as hex, prefix with "0x", and use two hex digits per byte. When encoding quantities (integers and numbers), encode as hex and prefix with "0x".

Parameters

The transaction object for this call contains:

- `from` DATA, 20 bytes – The address of the account sending the transaction.

- `to` DATA, 20 bytes – The address of the account receiving the transaction.
- `gas` QUANTITY – Optional. The gas, as an integer, provided for the transaction.
- `gasPrice` QUANTITY – Optional. The gas price, as an integer.
- `value` QUANTITY – Optional. The value, as an integer, sent with this transaction.
- `data` DATA, 20 bytes – Transaction data (compiled smart contract code or encoded method data).
- `nonce` QUANTITY – Optional. A nonce value, as an integer. This allows you to overwrite your own pending transactions that use the same nonce.
- `privateFrom` DATA, 20 bytes – The public key of the sender of this private transaction.
- `privateFor` DATA – An array of the public keys of the intended recipients of this private transaction.
- `restriction` STRING – If `restricted`, the transaction is a restricted private transaction. If `unrestricted`, the transaction is an unrestricted private transaction. For more information, see Section § 8.1.3 Private Transactions.
- `callbackUrl` STRING – The URL to post the results of the transaction to.

Callback Body

The callback object for this call contains:

- `txHash` DATA, 32 bytes – The transaction hash (if successful).
- `txIndex` QUANTITY – The index position, as an integer, of the transaction in the block.
- `blockHash` DATA, 32 Bytes – The hash of the block this transaction was in.
- `blockNumber` QUANTITY – The number of the block, as an integer, this transaction was in.
- `from` DATA, 20 Bytes – The public key of the sender of this private transaction.
- `to` DATA, 20 Bytes – The account address of the receiver. `null` if a contract creation transaction.
- `cumulativeGasUsed` QUANTITY – The total amount of gas used when this transaction was executed in the block.
- `gasUsed` QUANTITY – The amount of gas used by this specific transaction.
- `contractAddress` DATA, 20 Bytes – The contract address created, if a contract creation transaction, otherwise `null`.
- `logs` Array – An array of log objects generated by this transaction.
- `logsBloom` DATA, 256 Bytes – A bloom filter for light clients to quickly retrieve related logs.
- `error` STRING – Optional. Includes an error message describing what went wrong.

- `id` DATA – Optional. The ID of the request corresponding to this transaction, as provided in the initial [JSON-RPC] call.

Also returned is either:

- `root` DATA, 32 bytes – The post-transaction stateroot (pre-Byzantium).
- `status` QUANTITY – The return status, either 1 (success) or 0 (failure).

Request Format

```
curl -X POST --data
'{"jsonrpc":"2.0","method":"eea_sendTransactionAsync","params":[{
"from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
"to": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
"gas": "0x76c0",
"gasPrice": "0x9184e72a000",
"value": "0x9184e72a",
"data":"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb
"privateFrom": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
"privateFor": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
"callbackUrl": "http://myserver/id=1",
"restriction": "restricted"}],
"id":1}'
```

Response Format

```
{
"id":1,
"jsonrpc": "2.0"
}
```

Callback Format

```
{
"txHash":
"0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
"txIndex":  "0x1", // 1
"blockNumber": "0xb", // 11
"blockHash": "0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed
"cumulativeGasUsed": "0x33bc", // 13244
"gasUsed": "0x4dc", // 1244
"contractAddress": "0xb60e8dd61c5d32be8058bb8eb970870f07233155", // or nu
"logs": "[{
  // logs as returned by getFilterLogs, etc.
}, ...]",
"logsBloom": "0x00...0", // 256 byte bloom filter
"status": "0x1"
}
```

7.3.2.2 eea_sendTransaction

A call to `eea_sendTransaction` creates a private transaction, signs it, generates the transaction hash and submits it to the transaction pool, and returns the transaction hash.

> NOTE
>
> As in the public Ethereum [JSON-RPC-API], the two key datatypes for this call, which are passed hex encoded, are unformatted data byte arrays (DATA) and quantities (QUANTITY). When encoding unformatted data, encode as hex, prefix with "0x", and use two hex digits per byte. When encoding quantities (integers and numbers), encode as hex and prefix with "0x".

Parameters

The transaction object containing:

- `from` DATA, 20 bytes – The address of the account sending the transaction.
- `to` DATA, 20 bytes – Optional when creating a new contract. The address of the account receiving the transaction.
- `gas` QUANTITY – Optional. The gas, as an integer, provided for the transaction.
- `gasPrice` QUANTITY – Optional. The gas price, as an integer.
- `value` QUANTITY – Optional. The value, as an integer, sent with this transaction.
- `data` DATA, 20 bytes – Transaction data (compiled smart contract code or encoded method data).

- `nonce` QUANTITY – Optional. A nonce value, as an integer. This allows you to overwrite your own pending transactions that use the same nonce.
- `privateFrom` DATA, 20 bytes – The public key of the sender of this private transaction.
- `privateFor` DATA – An array of the public keys of the intended recipients of this private transaction.
- `restriction` STRING – If `restricted`, the transaction is a restricted private transaction. If `unrestricted`, the transaction is an unrestricted private transaction. For more information, see Section § 8.1.3 Private Transactions.

Returns

DATA, 32 Bytes – The transaction hash, or the zero hash if the transaction is not yet available.

If creating a contract, use `eth_getTransactionReceipt` to retrieve the contract address after the transaction is finalized.

Request Format

```
curl -X POST --data
'{"jsonrpc":"2.0","method":"eea_sendTransaction","params": [{
"from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
"to": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
"gas": "0x76c0",
"gasPrice": "0x9184e72a000",
"value": "0x9184e72a",
"data":
"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb9708
"privateFrom": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
"privateFor": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
"restriction": "restricted"}],
"id":1}'
```

Response Format

```
{
"id":1,
"jsonrpc": "2.0",
"result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527
}
```

7.3.2.3 eea_sendRawTransaction

Creates a private transaction, which has already been signed, generates the transaction hash and submits it to the transaction pool, and returns the transaction hash.

The signed transaction passed as an input parameter is expected to include the `privateFrom`, `privateFor`, and `restriction` fields.

> NOTE
>
> As in the public Ethereum [JSON-RPC-API], the two key datatypes for this call, which are passed hex encoded, are unformatted data byte arrays (DATA) and quantities (QUANTITY). When encoding unformatted data, encode as hex, prefix with "0x", and use two hex digits per byte. When encoding quantities (integers and numbers), encode as hex and prefix with "0x".

Parameters

The transaction object containing:

- `data` DATA – The signed transaction data.

```
params: ["0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058
```

Returns

DATA, 32 Bytes – The transaction hash, or the zero hash if the transaction is not yet available.

If creating a contract, use `eth_getTransactionReceipt` to retrieve the contract address after the transaction is finalized.

Request Format

```
curl -X POST --data
'{"jsonrpc":"2.0","method":"eea_sendRawTransaction","params": [{see above
"id":1}'
```

Response Format

```
{
"id":1,
"jsonrpc": "2.0",
"result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527
}
```

7.3.2.4 eea_sendRawTransactionAsync

Creates a private transaction, which has already been signed, submits it to the transaction pool, and returns immediately.

Using this method allows sending many transactions without waiting for recipient confirmation.

The signed transaction passed as an input parameter is expected to include the `privateFrom`, `privateFor`, `restriction`, and `callbackUrl` fields.

> NOTE
>
> As in the public Ethereum [JSON-RPC-API], the two key datatypes for this call, which are passed hex encoded, are unformatted data byte arrays (DATA) and quantities (QUANTITY). When encoding unformatted data, encode as hex, prefix with "0x", and use two hex digits per byte. When encoding quantities (integers and numbers), encode as hex and prefix with "0x".

Parameters

The transaction object containing:

- `data` DATA – The signed transaction data.

```
params: ["0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058
```

Callback Body

The callback object for this call contains:

- `txHash` DATA, 32 bytes – The transaction hash (if successful).
- `txIndex` QUANTITY – The index position, as an integer, of the transaction in the block.
- `blockHash` DATA, 32 Bytes – The hash of the block this transaction was in.
- `blockNumber` QUANTITY – The number of the block, as an integer, this transaction was in.
- `from` DATA, 20 Bytes – The public key of the sender of this private transaction.
- `to` DATA, 20 Bytes – The address of the account receiving this transaction. `null` if a contract creation transaction.
- `cumulativeGasUsed` QUANTITY – The total amount of gas used when this transaction was executed in the block.
- `gasUsed` QUANTITY – The amount of gas used by this specific transaction.
- `contractAddress` DATA, 20 Bytes – The contract address created, if a contract creation transaction, otherwise `null`.
- `logs` Array – An array of log objects generated by this transaction.

- `logsBloom` DATA, 256 Bytes – A bloom filter for light clients to quickly retrieve related logs.
- `error` STRING – Optional. Includes an error message describing what went wrong.
- `id` DATA – Optional. The ID of the request corresponding to this transaction, as provided in the initial [JSON-RPC] call.

Also returned is either:

- `root` DATA, 32 bytes – The post-transaction stateroot (pre-Byzantium).
- `status` QUANTITY – The return status, either 1 (success) or 0 (failure).

Request Format

```
curl –X POST ––data
'{"jsonrpc":"2.0","method":"eea_sendRawTransactionAsync","params": [{see
"id":1}'
```

Response Format

```
{
"id":1,
"jsonrpc": "2.0"
}
```

Callback Format

```
{
"txHash":
"0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
"txIndex":  "0x1", // 1
"blockNumber": "0xb", // 11
"blockHash": "0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed
"cumulativeGasUsed": "0x33bc", // 13244
"gasUsed": "0x4dc", // 1244
"contractAddress": "0xb60e8dd61c5d32be8058bb8eb970870f07233155", // or nu
"logs": "[{
  // logs as returned by getFilterLogs, etc.
}, ...]",
"logsBloom": "0x00...0", // 256 byte bloom filter
"status": "0x1"
}
```

7.3.2.5 eea_clientCapabilities

A call to `eea_clientCapabilities` provides more information about the capabilities supported by the client. This call returns the private transaction restriction levels and the kinds of consensus mechanisms supported.

Parameters

None.

Returns

Client capability information fields in the format of [JSON] name values pairs:

- `consensus : ["PoW", "IBFT" , "Raft"]`
- `restriction: ["restricted", "unrestricted"]`

Request Format

```
curl –X POST ––data
'{"jsonrpc":"2.0","method":"eea_clientCapabilities","params":[],"id":1}'
```

Response Format

```
{
"id":1,
"jsonrpc": "2.0",
"result": [{"consensus": ["PoW", "IBFT" , "Raft"]},
{"restriction": ["restricted", "unrestricted"]} }
```

7.3.3 Permissioning Smart Contract interfaces

This section presents smart contract providing the necessary information for Enterprise Ethereum clients to enforce permissioning models in an interoperable manner. This includes both node and account permissioning interfaces.

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

- Permissioning enforcement smart contract functions.

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

  - `connectionAllowed`, to determine whether to permit a connection with another node.

    - `transactionAllowed`, 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.

  Both node and account smart contract interfaces emit a `permissionsUpdated` event 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 event contains an `addsRestrictions` Boolean flag. If the flag is `true`, any previous `connectionAllowed` or `transactionAllowed` call could now result in a different outcome, potentially making the previously checked permissions more restrictive. If `addsRestrictions` is `false`, this indicates that previous `connectionAllowed` or `transactionAllowed` calls that returned `false` could now return `true` because the permissions are now less restrictive than when previously checked.

- Permissioning management smart contract functions.

  These smart contract 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 of 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 smart contract (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 a `permissionsUpdated` event based on the permissions change.

7.3.3.1 Node Permissioning

Node permissioning 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.

[P] PERM-200: Enterprise Ethereum clients *MUST* call the `connectionAllowed` function, as specified in Section § 7.3.3.1.1 Node Permissioning Functions, to determine whether a connection with another node is permitted, and any restrictions to be placed on that connection.

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

[P] PERM-210: When checking the response to `connectionAllowed`, if any unknown permissioning bits are found to be zero, Enterprise Ethereum clients *MUST* reject the connection.

[P] PERM-220: On receipt of a `NodePermissionsUpdated` event containing an 'addsRestrictions' property with value 'true', Enterprise Ethereum clients *MUST* close any network connections that are no longer permitted, and impose newly added restrictions on any network connections that have had restrictions added.

[P] PERM-230: On receipt of a `NodePermissionsUpdated` event containing an 'addsRestrictions' property with value 'false', Enterprise Ethereum clients *MUST* conduct a check on whether existing network connections have had their restrictions lifted and allow future actions that are now permitted.

7.3.3.1.1 Node Permissioning Functions

Support must be provided for both IPv4 and IPv6 protocol versions, so the node connection rules must account for this. 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 endain byte ordering and prefixing it with 80 bits of 0's followed by 16 bits of 1's.

```
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": "res",
        "type": "bytes32"
      }
    ]
  },
```

```
  {
    "type": "event",
    "name": "NodePermissionsUpdated",
    "inputs": [
      {
        "name": "addsRestrictions",
        "type": "bool",
        "indexed": false
      }
    ]
  }
]
```

Arguments

- `sourceEnodeHigh`: The high (first) 32 bytes of the enode address of the node initializing 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.
- `res`: 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 if `false`, the change event represents the permission set becoming more permissive.

7.3.3.1.2 Node Permissions

While the core premise of node permissioning is whether a connection is allowed to occur, there are additional restrictions that can be imposed on a connection between two nodes based on the permitted behaviour 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 should restrict the behaviour of the remote node according to the unset bits.

The remaining bits in the response are to be set to one. If any of the remaining bits are found to be zero, then it should be assumed that an unknown permission restriction was placed on the connection and the connection should be denied. These unknown zeros are likely to be representing 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.

7.3.3.1.3 Client Implementation

A client connecting to a chain that maintains a smart contract exposing the node permissioning interface can expect to be supplied the address of the contract. When a peer connection request is received, or a new connection request initiated, the smart 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 it is expected that the client will be provided a bootnode and initially have 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.

7.3.3.1.4 Chain Initialization

At the genesis block the node permissioning contract should be included in block 0 and configured such that the initial nodes are able to establish connections to each other.

7.3.3.2 Account Permissioning

Account permissioning controls which accounts are able to send transactions and the type of transactions permitted.

[P] PERM-240: When validating or mining a block, Enterprise Ethereum clients *MUST* call the `transactionAllowed` function, as specified in Section § 7.3.3.2.1 Account Permissioning Smart Contract Interface Function, with worldstate as at the block's parent, to determine if a transaction is permitted in a block.

7.3.3.2.1 Account Permissioning Smart Contract Interface Function

```
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": "res",
        "type": "bool"
      }
    ]
  },
  {
    "type": "event",
    "name": "AccountPermissionsUpdated",
    "inputs": [
      {
        "name": "addsRestrictions",
        "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. In case of 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.
- `res`: A Boolean representing whether the transaction should be allowed and considered valid.
- `addsRestrictions`: If the rules change that caused the `AccountPermissionsUpdated` event to be emitted involves further restricting existing permissions this will be `true`. Otherwise if `false` the change event represents the permission set becoming more permissive.

7.3.3.2.2 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.

When mining new blocks the node checks the validity of transactions using the appropriate smart contract with the state at the block's parent. If not permitted, the transaction is discarded. If permitted, the transaction is included in the new block and the block dispatched to other nodes.

When receiving a block the node checks each included transaction using the smart contract with the state at the block's parent. If any transactions in the new block are not permitted, the block is considered invalid and discarded. If all transactions are permitted, the block passes the permissioning validation check and is then subject to any other validity assessments the client might usually perform.

Depending on the use case of a client, the implementation might also check validity of transactions upon being lodged through RPC methods or received through peer-to-peer communication. For

such validation, it is expected that the contracts are used with the state represented at the current head.

Reading of a contract is an unrestricted operation.

7.3.3.2.3 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.

7.3.3.2.4 Chain Initialization

At the genesis block the account permissioning smart contract function should be included in block 0 and configured such that 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.

7.3.3.3 Permissioning management: examples

The smart contracts that allow Enterprise Ethereum clients to apply permissioning require a management system behind them. This section provides some examples of how such a system might work, but each Enterprise Ethereum blockchain is free to implement whatever system is suitable, as long as it includes the required functions for clients to query.

Example: Permissioning contracts: simple authorization whitelist

This section presents a basic permissioning smart contract that maintains a whitelist of enodes allowed to participate in the Enterprise Ethereum blockchain, and a list of administrative accounts that can alter that whitelist.

In this model administrators are allowed to add and remove administrators, and add and remove nodes. Clients can check whether a given connection is permitted via the `connectionAllowed` function which checks that the enode addresses of both nodes involved in the connection are in the whitelist. When a permission rule update occurs a `NodePermissionsUpdated` event is emitted indicating that a rules change has occurred, and whether the rule change makes the Enterprise Ethereum blockchain more permissive or adds new restrictions.

When deploying the contract it is initialised with the account deploying it as the first administrator. This account can then add additional administrators and add nodes to the whitelist.

This contract serves as an example only. As written it has shortcomings that should be considered in any production environment. Some potentially desirable features which are not included in this example are:

- Protection against all administors being removed
- A list of current administrators
- The current whitelisted enodes
- Protection against a single administrator taking over the permissioning system
- A way to upgrade the contract logic
- Grouping or organization of whitelist members

```
pragma solidity >=0.4.0 <0.6.0;
contract SimplePermissioning {
    // Struct representing an enode
    struct Enode {
        bytes32 enodeHigh;
        bytes32 enodeLow;
        bytes16 enodeHost;
        uint16 enodePort;
    }
```

```
// Event emitted when a rules change occurs
event NodePermissionsUpdated(
    bool addsRestrictions
);

// List of nodes permitted to participate in the network
mapping(bytes => Enode) private whitelist;

// List of accounts allowed to modify the network
mapping(address => bool) private adminList;

constructor() public {
    // set the contract creator as the first admin
    adminList[msg.sender] = true;
}

// Guard modifier for functions that should only be invokable by admins
modifier onlyAdmin() {
    require(adminList[msg.sender] == true, "Caller must be in the admin list")
    _;
}

// Add an admin to the contract
function addAdmin(address newAdmin) public onlyAdmin {
    adminList[newAdmin] = true;
}

// Remove an admin from the contract
function removeAdmin(address oldAdmin) public onlyAdmin {
    adminList[oldAdmin] = false;
}

// Check if a connection between two nodes should be permitted
function connectionAllowed(
    bytes32 sourceEnodeHigh,
    bytes32 sourceEnodeLow,
    bytes16 sourceEnodeIp,
    uint16 sourceEnodePort,
    bytes32 destinationEnodeHigh,
    bytes32 destinationEnodeLow,
    bytes16 destinationEnodeIp,
    uint16 destinationEnodePort
) public view returns (bytes32) {
    // Check that both are in the whitelist
    if (
        enodeAllowed(sourceEnodeHigh, sourceEnodeLow, sourceEnodeIp, sourceEno
        && enodeAllowed(destinationEnodeHigh, destinationEnodeLow, destination
    ) {
        // If both are then indicate permitted by returning the first bit as s
        return 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    } else {
        // If one or neither are permitted then indicate not permitted by unse
```

```
        return 0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    }
}

// Check if a specified enode is in the whitelist
function enodeAllowed(
    bytes32 sourceEnodeHigh,
    bytes32 sourceEnodeLow,
    bytes16 sourceEnodeIp,
    uint16 sourceEnodePort
) public view returns (bool){
    bytes memory key = computeKey(sourceEnodeHigh, sourceEnodeLow, sourceEnode
    Enode storage whitelistSource = whitelist[key];
    if (whitelistSource.enodeHost > 0) {
        return true;
    }
}

// Add a specified enode to the whitelist
function addEnode(
    bytes32 enodeHigh,
    bytes32 enodeLow,
    bytes16 enodeIp,
    uint16 enodePort
) public onlyAdmin {
    Enode memory newEnode = Enode(enodeHigh, enodeLow, enodeIp, enodePort);
    bytes memory key = computeKey(enodeHigh, enodeLow, enodeIp, enodePort);
    whitelist[key] = newEnode;
    // Emit an event indicating a permissioning update occurred that only
    // allows new nodes into the network
    emit NodePermissionsUpdated(false);
}

// Remove a specified enode from the whitelist
function removeEnode(
    bytes32 enodeHigh,
    bytes32 enodeLow,
    bytes16 enodeIp,
    uint16 enodePort
) public onlyAdmin {
    bytes memory key = computeKey(enodeHigh, enodeLow, enodeIp, enodePort);
    Enode memory zeros = Enode(bytes32(0), bytes32(0), bytes16(0), 0);
    whitelist[key] = zeros;
    // Emit an event indicating a permissioning update occurred that can
    // cause existing connections to now be disallowed
    emit NodePermissionsUpdated(true);
}

// Compute a consistent hashkey for a given enode
function computeKey(
    bytes32 enodeHigh,
    bytes32 enodeLow,
    bytes16 enodeIp,
```

```
        uint16 enodePort
    ) public pure returns (bytes memory) {
        return abi.encode(enodeHigh, enodeLow, enodeIp, enodePort);
    }

}
```

Example: Permissioning contracts: memberGroups and authorizedUsers

This section presents an example permissioning model with four smart contract interfaces for management, and methods for enforcement. These example interfaces are for node permissioning.

A permissioned Enterprise Ethereum blockchain includes smart contracts that implement the permissioning enforcement interfaces specified in Section § 7.3.3 Permissioning Smart Contract, along with whatever management functions are appropriate for the chosen permissioning model.

The model in this example is based on the concept of a ***network***, that consists of a set of ***memberGroups***, each representing an enterprise or organization, and each made up of authorizedUsers.

In this model nodes are added to or removed from the node list of a memberGroup. If a memberGroup joins the network, the nodes in that memberGroup's node list are permitted to join. Conversely, if the memberGroup is removed from the network, the nodes associated with that memberGroup are disconnected.

As noted, memberGroups are a collection of authorizedUsers. An ***authorizedUser*** is represented by one or more Ethereum accounts. This provides robustness in case a given user loses the keys to one of their accounts, as well as reflecting the reality that many users operate more than a single account.

Each authorizedUser has individually specified permissions, to act on behalf of the memberGroup in administering the Enterprise Ethereum blockchain. Depending on the specific permissions they have, an authorizedUser can make changes such as adding a new authorizedUser to a memberGroup, adding a new node to a memberGroup's node list, or inviting another memberGroup to join the Enterprise Ethereum blockchain.

A ***permissioning decider*** function is used to decide whether or not a memberGroup should be permitted to join the network, when to evict a memberGroup, and especially whether to change the permissioning decider itself.

This model uses four fundamental smart contract interfaces:

- `AuthorizedUser`
- `MemberGroup`
- `Network`
- `PermissioningDecider`.

AuthorizedUser

The `authorizedUser` smart contract contains authorizedUsers, initialized with a name and an identifier, like an email address. Additional information such as alternative contact information or PGP public keys, could also be included by implementations of `authorizedUser`.

```
pragma solidity ^0.5.1;
interface AuthorizedUser {


 // Metadata

 // Retrieve the user's name.
 function getName() external view returns (string memory) ;

 // Retrieve the user's identifier.
 function getId() external view returns (string memory) ;

 // Authorization mutation.

 // Add an Ethereum account address for an authorizedUser. Multiple addresses
 // can be added.
 function addAddress(address _owner) external ;

 // Remove an Ethereum account address from an authorizedUser.
 function removeAddress(address _owner) external ;

 // Authorization queries.

 // Check if the authorizedUser owns a specific Ethereum account address.
 function owns(address _owner) external view returns (bool) ;

 // Network of trust (reputation) mutators.

 // Check if the authorizedUser vouches for another (child) authorizedUser.
 function hasEndorsed(AuthorizedUser _child) external view returns (bool) ;

 // Vouch for another authorizedUser.
 function endorse(AuthorizedUser _child) external ;

 // Stop vouching for an authorizedUser
 function unendorse(AuthorizedUser _child) external ;

 // Network of trust backlinks. These should be called by `endorse` and
 // `unendorse` implementations respectively, to provide pointers about where
 // to look for endorsements.

 // Set the (parent) authorizedUser as vouching for the authorizedUser.
 function recordEndorsement(AuthorizedUser _parent) external ;

 // Set the (parent) authorizedUser to no longer vouch for the authorizedUser.
 function eraseEndorsement(AuthorizedUser _parent) external ;

}
```

A user is responsible for deciding to endorse someone as an authorizedUser. Some examples of how Alice might decide to endorse Bob

- alice@a.net sends an email to bob@a.net, asking to confirm Bob's authorizedUser address.
- alice@a.net sends an email to bob@b.com, asking Bob to join a video call to assert his ownership of a given Ethereum account.
- Alice walks over to Bob's desk and asks for his Ethereum account address.

> NOTE
>
> Any caller can add any address as a parent of a authorizedUser. To authenticate an authorizedUser, the contract follows the parent links and checks that the corresponding child link is present.

MemberGroup

The `MemberGroup` represents a group of authorizedUsers and their permissions.

```
pragma solidity ^0.5.1;
import "./AuthorizedUser.sol";
interface MemberGroup {


 // Metadata

 // Retrieve the memberGroup name.
 function getName() external view returns (string memory) ;

 // Retrieve the permissions for an authorizedUser.
 function permission(AuthorizedUser) external view returns (uint) ;

 // Retrieve the number of authorizedUsers in the memberGroup.
 function memberCount() external view returns (uint) ;

 // Retrieve an authorizedUser, specified by index idx, from the memberGroup.
 function getMember(uint idx) external view returns (AuthorizedUser) ;

 // Managing membership of the group

 // Add an authorizedUser to the memberGroup. Requester must have
 // `CAN_ADD_USER` permission.
 function addUser(AuthorizedUser requester, AuthorizedUser object,
                              uint _permission) external ;

 // Remove a authorizedUsers from the memberGroup. Requester must have
 // `CAN_REMOVE_USER` permission.
 function removeUser(AuthorizedUser requester,
                              AuthorizedUser object) external ;

 // Events

 // Emitted when an authorizedUser is added to a memberGroup.
 event MemberAdded(AuthorizedUser _user, uint _permission) ;

 // Emitted when an authorizedUser is removed from a memberGroup.
 event MemberRemoved(AuthorizedUser _user, uint _permission) ;

}
```

An authorizedUser is given permissions as follows:

- The authorizedUser who calls the `addUser` funtion - the "requestor" - proposes a set of permissions as the `_permission` parameter.
- The contract does a bitwise AND of the requester's own permissions with the request.
- The result is recorded as the permission for the newly added authorizedUser.

A partial definition of permissions could be as follows:

```
pragma solidity ^0.5.1;
contract Permissions {


 // Permission to add or remove an authorizedUser.
 uint constant public CAN_ADD_USER = 0x1;
 uint constant public CAN_REMOVE_USER = 0x2;

 // Permission to change the node list of a memberGroup.
 uint constant public CAN_ADD_NODE = 0x4;
 uint constant public CAN_REMOVE_NODE = 0x8;

 // Permission to vote for other memberGroups to join or leave the network.
 uint constant public CAN_INVITE_MEMBERGROUP = 0x10;
 uint constant public CAN_UNINVITE_MEMBERGROUP = 0x20;

 // Permission to vote for a new <a>permission decider</a>
 uint constant public CAN_PROPOSE_DECIDER = 0x100;

 uint constant public ADMIN = 0x1ff;

 function meets(uint have, uint needed) public pure returns (bool) {
     return have & needed == needed;
 }

}
```

Network

As described above, networks are a collection of memberGroups. Each memberGroup manages a ***node list***: a list of [enode] URLs, corresponding to the nodes allowed to connect to the Enterprise Ethereum blockchain that are controlled by the memberGroup. Any authorizedUser that is part of the memberGroup and has the `CAN_ADD_NODE` permission can add to the node list for that memberGroup.

To add a memberGroup to or remove them from the Enterprise Ethereum blockchain, every memberGroup that is already a member can vote to `invite` or `uninvite` them. The permissioning decider for the network determines whether to update the network members, given the current set of requests.

A memberGroup can have `WRITE` or `READ` permissions, set as part of adding it to the network. A node belonging to a memberGroup that has `WRITE` permission can submit a transaction, but transactions from a node in a memberGroup that only has `READ` permission result in the `transactionAllowed` method returning `res: false` for that transaction.

```
pragma solidity ^0.5.1;
import "./MemberGroup.sol";
import "./AuthorizedUser.sol";
import "./PermissioningDecider.sol";
interface Network {
```

```
// Node queries.

// Retrieve the memberGroup that a node is part of.
function memberGroupOf(string calldata _node) external view
    returns (MemberGroup) ;

// Retrieve the number of nodes in the memberGroup.
function memberGroupsNodeCount(MemberGroup) external view
    returns (uint) ;

// Retrieve a node, specified by index idx, from the memberGroup.
function memberGroupsNode(MemberGroup, uint idx) external view
    returns (string memory) ;

// Authorization queries.

// Retrieve the permissions for the memberGroup.
function permission(MemberGroup) external view returns (uint) ;

// Check if 2 nodes are allowed to connect to one another
function connectionAllowed(
    bytes32 sourceEnodeHigh,
    bytes32 sourceEnodeLow,
    bytes32 sourceIp,
    bytes32 sourcePort,
    bytes32 destinationEnodeHigh,
    bytes32 destinationEnodeLow,
    bytes32 destinationIp,
    bytes32 destinationPort
) external view returns (bytes32) ;

// Group administration.

// Add a node to a memberGroup. This will fail unless the authorizedUser has
// `CAN_ADD_NODE` permission.
function addNode(MemberGroup, AuthorizedUser, string calldata _node) external

// Remove a node from a memberGroup. This will fail unless the authorizedUser
// `CAN_REMOVE_NODE` permission
function removeNode(MemberGroup, AuthorizedUser, string calldata _node) extern

// Group membership queries.

// Retrieve the number of memberGroups.
function memberGroupCount() external view returns (uint) ;

// Retrieve a memberGroup, specified by index idx
function getmemberGroup(uint idx) external view
    returns (MemberGroup) ;

// Network vote counts.
```

```
// Retrieve the number of invites for the memberGroup to have `READ`
// permissions in the context of the network.
function readInvitesReceived(MemberGroup) external view
    returns (uint) ;

// Retrieve the number of invites for the memberGroup to have `WRITE`
// permissions in the context of the network.
function writeInvitesReceived(MemberGroup) external view
    returns (uint) ;

// Retrieve the number of uninvites for the memberGroup to leave the
// network.
function uninvitesReceived(MemberGroup) external view returns (uint) ;

// Group membership mutators.

// Invite a memberGroup to join the network. This will fail unless the authori
// `CAN_INVITE_MEMBERGROUP` permission.
function invite(MemberGroup _invitee, MemberGroup _ginviter,
                    AuthorizedUser _uinviter, string calldata _node, uint _p

// Uninvite a memberGroup from the network. This will fail unless the authoriz
// `CAN_UNINVITE_MEMBERGROUP` permission.
function uninvite(MemberGroup _invitee, MemberGroup _ginviter,
                    AuthorizedUser _uinviter) external ;

// Rule inspection.

// Retrieve the permission decider function currently in use.
function decider() external view returns (PermissioningDecider) ;

// Rule vote counts.

// Retrieve the number of votes received for the permissioning decider.
function deciderVotesReceived(PermissioningDecider) external view
    returns (uint) ;

// Retrieve the permissioning decider nominated by the memberGroup.
// Useful for admin weighting.
function nominatedDecider(MemberGroup) external view
    returns (PermissioningDecider) ;

// Change the rule engine.

// Propose a new permissioning decider.
function proposeDecider(PermissioningDecider _next,
                            MemberGroup _gproposer, AuthorizedUser _uproposer)
                            external ;
// Emitted events.

// The permissions set was updated
event NodePermissionsUpdated(bool addsRestrictions);
```

```
    // A node was added to a memberGroup.
    event NodeAdded(MemberGroup _changed_group, string _node);

    // A node was removed from a memberGroup.
    event NodeRemoved(MemberGroup _changed_group, string _node);

    // An authorizedUser has invited a memberGroup to join the network.
    event memberGroupInvited(MemberGroup _invited_group,
                                uint _permission);

    // An authorizedUser has uninvited a memberGroup from the network.
    event memberGroupUnInvited(MemberGroup _uninvited_group,
                                uint _permission);

    // A memberGroup was added to the network.
    event memberGroupAdded(MemberGroup _added_group,
                                uint _permission);

    // A memberGroup was removed from the network.
    event memberGroupRemoved(MemberGroup _removed_group,
                                uint _permission);

    // A permission decider function was swapped to a new one.
    event DeciderSwapped(PermissioningDecider _old, PermissioningDecider _new);

}
```

The `Network` contract checks whether the caller has permission to call the `invite`, `uninvite`, `proposeDecider`, `addNode`, and `removeNode` functions. The granularity of permissions is implementation-dependent.

PermissioningDecider

The `PermissioningDeciders` smart contract customizes the bylaws of a `Network` smart contract.

```
pragma solidity ^0.5.1;
import "./MemberGroup.sol";
import "./Network.sol";
interface PermissioningDecider {


 // The permission the memberGroup now has, if approved.
 function inviteApproved(Network, MemberGroup) external view
     returns (uint8) ;

 // Whether the network should remove the memberGroup.
 function inviteRevoked(Network, MemberGroup) external view
     returns (bool) ;

 // Whether the network should change its permissioning decider.
 function swapDecider(Network, PermissioningDecider) external view
     returns (bool) ;

}
```

Some example `PermissioningDeciders` include:

- Static: `memberGroups` are never removed or added from the `Network`. Any attempt to change the `PermissioningDecider` will fail.
- AutoApprove: `memberGroups` are automatically included (or removed) when invited (or uninvited). The decider swaps the first time it is asked.
- AdminRun: The `Network` has an administrator group, which is the only vote counted for approving or revoking approval of a `memberGroup`, or changing the `PermissionDecider`.
- MajorityRules: A prospective `memberGroup` needs more than half of the current `memberGroups` to invite it for membership. A prospective `PermissioningDecider` needs more than half of the current groups to nominate it before this `Decider` relinquishes control.

Node Blacklisting

Blacklisting a node from a memberGroup level can be done by by adding the following functions to the `MemberGroup` smart contract.

```
interface MemberGroup {


 //... To the existing content, add

 // Blacklist an authorizedUser.
 function blacklistNode(AuthorizedUser, string _node) interface ;

 // Remove an authorizedUser from the blacklist.
 function unblacklistNode(AuthorizedUser, string _node) interface ;

}
```

Blacklisting of nodes for the whole Enterprise Ethereum blockchain can be done by adding the following functions to the `Network` and `PermissioningDecider` smart contracts.

```
interface Network {


 // ... to the existing content, add

 // Vote to add an authorizedUser to the blacklist.
 voteToBlacklist(MemberGroup, AuthorizedUser, string _node) external ;

 // Vote to remove an authorizedUser from the blacklist.
 voteToUnblacklist(MemberGroup, AuthorizedUser, string _node) external ;

 // Retrieve the number of votes for the node to be added to the blacklist.
 blacklistVotesReceived(string _node) external view returns (uint);

 // Retrieve the number of votes for the node to be removed from the
 // blacklist.
 unblacklistVotesReceived(string _node) external view returns (uint);

 // Emitted when a node is added to the blacklist.
 event NodeBlacklisted(MemberGroup _blacklisted_group, string _node);

 // Emitted when a node is removed from the blacklist.
 event NodeUnblacklisted(MemberGroup _unblacklisted_group, string _node);
 ...

}
```

```
interface PermissioningDecider {

...
// Whether the node should be added to the blacklist.
function blacklistApproved(Network, string _node) external view
    returns (bool);

// Returns whether the node should be removed from the blacklist.
function unblacklistApproved(Network, string _node) external view
    returns (bool);
...

}
```

7.3.4 Inter-chain

This section is non-normative.

With the rapid expansion in the number of different blockchains and ledgers, ***inter-chain mediators*** allow interaction between these blockchains. Like other organizational solutions that provide privacy and scalability, inter-chain mediators can be Layer 2, such as using public Ethereum to anchor (or peg) state needed to track and checkpoint state.

7.3.5 Oracles

In many situations, smart contracts need to interact with real-world information to operate. An ***oracle*** is a service external to either public Ethereum or an Enterprise Ethereum client that is trusted by the creators of smart contracts and is called to provide information, such as a current exchange rate, or the result of a mathematical calculation. Oracles are a secure bridge between smart contracts and real-world information sources.

[C] ORCL-010: Enterprise Ethereum clients *SHOULD* provide the ability to securely interact with oracles to send and receive external off-chain information.

8. Privacy and Scaling Layer

The Privacy and Scaling layer implements the necessary privacy and scaling extensions needed in Enterprise Ethereum to support enterprise-grade deployments.

For the purpose of this Specification:

- ***Privacy***, as defined in ITU [X.800], is "The right of individuals to control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed." This right of privacy also applies to organizations to the extent permitted by law.
- ***Scaling*** is the act of increasing the capability of systems, networks, or processes to handle more work. In terms of Enterprise Ethereum, this is about increasing transaction speed using on-chain scaling or off-chain scaling mechanisms, or both.
- A ***transaction*** is a request to execute operations that change state in an Enterprise Ethereum blockchain. Transactions can involve one or more accounts.
- 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 are required to support private transactions, as outlined in Section § 8.1.3 Private Transactions.

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

This Specification does not seek to constrain experimentation to improve the scalability of future public Ethereum or Enterprise Ethereum clients. Instead, there is recognition that several forms of scaling improvements will be made over time, the exact form of which cannot be known at this time.

Scaling solutions are broadly categorized into:

- Layer 1 scaling solutions, which are implemented at the base level protocol layer, and might be implemented using techniques such as [sharding] and easy parallelizability [EIP-648].
- Layer 2 scaling solutions, which do not require changes to the base level protocol layer and are implemented at the application protocol layer using smart contracts, [Plasma] or [state-channels] as well as other Off-Chain (Compute) scaling mechanisms.

Similarly, various on-chain privacy mechanisms are being explored. ***On-chain privacy mechanisms*** are extensions to public Ethereum, such as zero-knowledge proofs (like ZK-SNARKS), or privacy-preserving trusted computing, enabling private transactions.

A ***zero-knowledge proof*** is a cryptographic method 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 x.

8.1 Privacy Sublayer

Enterprise Ethereum clients are expected to provide some level of transaction privacy. Privacy can be realized in various ways including controlling which nodes see private transactions, and obfuscating transaction data. Options for implementing compliant privacy levels are detailed in Section § 8.1.4 Privacy Levels

8.1.1 On-chain

This section is non-normative.

Various on-chain techniques are proposed to 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-SNARKS, range proofs, or ring signatures.

8.1.2 Off-chain (Trusted Computing)

This section is non-normative.

Off-chain trusted computing is the offloading of computationally intensive processing to achieve scalability improvements, while still maintaining transaction privacy. Transactions and smart contracts are executed outside the core blockchain for improved privacy, performance, or security. Such systems could be hardware-based, software-based, or both depending on the use case.

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

Enterprise Ethereum clients could provide off-chain trusted compute of transactions by implementing requirement EXEC-050.

8.1.3 Private Transactions

Many users and operators of Enterprise Ethereum clients are required by their legal jurisdictions to comply with laws and 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 users specify their preferred private transaction type at runtime by using the `restriction` parameter on their [JSON-RPC-API] calls. The two available private transaction types are:

- ***Restricted private transactions***, where payload data is transmitted to and readable only by the parties to the transaction.
- ***Unrestricted private transactions***, where payload data is transmitted to all nodes in the Enterprise Ethereum blockchain, but readable only by the parties to the transaction.

[P] PRIV-010: Enterprise Ethereum clients *MUST* support private transactions using either restricted private transactions or unrestricted private transactions.

Transaction information consists of two parts:

- ***Metadata***, which is the set of data that describes and gives information about the payload data in a transaction. Metadata is the *envelope* information necessary to execute a transaction.
- ***Payload data***, which is the content of the data field of a transaction, usually obfuscated in private transactions. Payload data is separate from the metadata in a transaction.

If implementing restricted private transactions:

- **[P] PRIV-020:** Enterprise Ethereum clients *MUST* support encryption of the payload data when stored in restricted private transactions.
- **[P] PRIV-030:** Enterprise Ethereum clients *MUST* support encryption of the payload data when in transit in restricted private transactions.
- **[P] PRIV-040:** Enterprise Ethereum clients *MAY* support encryption of the metadata when stored in restricted private transactions.
- **[P] PRIV-050:** Enterprise Ethereum clients *MAY* support encryption of the metadata when in transit in restricted private transactions.
- **[P] PRIV-060:** Nodes that relay a restricted private transaction, but are not party to that transaction, *MUST NOT* store the payload data.
- **[P] PRIV-070:** Nodes that relay a restricted private transaction, but are not party to that transaction, *SHOULD NOT* store the metadata.
- **[P] PRIV-080:** The implementation of the `eea_sendTransactionAsync`, `eea_sendTransaction`, `eea_sendRawTransactionAsync`, or `eea_sendRawTransaction` methods (see Section § 7.3.2 Extensions to the JSON-RPC API) with the `restriction` parameter set to `restricted`, *MUST* result in a restricted private transaction.

NOTE: Restricted Private Transactions

Private transactions can be implemented by creating private channels, that is, private smart contracts where the payload data is only stored by the clients participating in a transaction, and not by any other client (despite that the payload data might be encrypted and only decodable by authorized parties).

Private transactions are kept private between related parties, so unrelated parties have no access to the content of the transaction, the sending party, or the addresses of accounts party to the transaction. In fact, a private smart contract has a similar relationship to the blockchain that hosts it as a private blockchain that is only replicated and certified by a subset of participating nodes, but is notarized and synchronized on the hosting blockchain. This private blockchain is thus able to refer to data in less restrictive private smart contracts, as well as in public smart contracts.

If implementing unrestricted private transactions:

- **[P] PRIV-090:** Enterprise Ethereum clients *SHOULD* support encryption of the recipient identity when stored in unrestricted private transactions.
- **[P] PRIV-100:** Enterprise Ethereum clients *SHOULD* support encryption of the sender identity when stored in unrestricted private transactions.
- **[P] PRIV-110:** Enterprise Ethereum clients *SHOULD* support encryption of the payload data when stored in unrestricted private transactions.
- **[P] PRIV-120:** Enterprise Ethereum clients *MUST* support encryption of the payload data when in transit in unrestricted private transactions.
- **[P] PRIV-130:** Enterprise Ethereum clients *MAY* support encryption of the metadata when stored in unrestricted private transactions.
- **[P] PRIV-140:** Enterprise Ethereum clients *MAY* support encryption of the metadata when in transit in unrestricted private transactions.
- **[P] PRIV-150:** Nodes that relay an unrestricted private transaction, but are not party to that transaction, *MAY* store the payload data.
- **[P] PRIV-160:** Nodes that relay an unrestricted private transaction, but are not party to that transaction, *MAY* store the metadata.
- **[P] PRIV-170:** The implementation of the `eea_sendTransactionAsync`, `eea_sendTransaction`, `eea_sendRawTransactionAsync`, or `eea_sendRawTransaction` methods (see Section § 7.3.2 Extensions to the JSON-RPC API) with the `restriction` parameter set to `unrestricted` *MUST* result in an unrestricted private transaction.

- **[P] PRIV-210:** Enterprise Ethereum clients' implementation of unrestricted private transactions *MUST* provide the ability for nodes to achieve global consensus.

> NOTE: Unrestricted Private Transactions
>
> Obfuscated data that is replicated across all nodes can be reconstructed by any node, albeit in encrypted form. Mathematical transactions on numerical data are intended to be validated by the underlying Enterprise Ethereum blockchain on a zero-knowledge basis. The plaintext content is only available to participating parties to the transaction. Thus, a node is expected to have the ability to maintain and transact against numerical balances certified by the whole community of validators on a zero-knowledge basis.
>
> An alternative to the zero-knowledge approach could be the combined use of ring signatures, stealth addresses, and mixing, which is demonstrated to provide the necessary level of obfuscation that is mathematically impossible to penetrate and does not rely on the trusted setup required by ZK-SNARKS.

[P] PRIV-180: Enterprise Ethereum clients *SHOULD* be able to extend the set of parties to a private transaction (or forward the private transaction in some way).

[P] PRIV-190: Enterprise Ethereum clients *SHOULD* provide the ability for nodes to achieve consensus on their mutually private transactions.

The differences between restricted private transactions and unrestricted private transactions are summarized in the table below.

Table 2 Restricted and Unrestricted Private Transactions

Restricted Private TXNs (if implemented)		**Unrestricted Private TXNs (if implemented)**	
Metadata	Payload Data	Metadata	Payload Data
MAY encrypt	*MUST* encrypt	*MAY* encrypt *SHOULD* encrypt sender and recipient identity	*MUST* encrypt in transit *SHOULD* encrypt in storage
SHOULD NOT allow storage by non-participating nodes	*MUST NOT* allow storage by non-participating nodes	*MAY* allow storage by non-participating nodes	*MAY* allow storage by non-participating nodes

8.1.4 Privacy Levels

Compliance with this Specification allows for three levels of privacy support. Because permissioning and privacy are interrelated concepts, the three privacy levels specified contain requirements related to both the permissioning and privacy sections of this Specification.

8.1.4.1 Privacy Level C

Privacy Level C is the base privacy level for all compliant implementations. Privacy Level C compliance is achieved by implementing all the *MUST* peer node connectivity and account permissioning requirements in Sections § 7.1.1 Nodes and § 7.1.2 Ethereum Accounts, and either:

- All *MUST* and *MUST NOT* restricted private transaction requirements, as outlined in Section § 8.1.3 Private Transactions, or
- All *MUST* and *MUST NOT* unrestricted private transaction requirements, as outlined in Section § 8.1.3 Private Transactions.

8.1.4.2 Privacy Level B

Supporting additional SHOULD requirements increases theprivacy and permissioning abilities for an Enterprise Ethereum client, adding specific value for users.

Privacy Level B compliance is achieved by providing support for the requirements of Privacy Level C, plus implementing all the *SHOULD* requirements related to peer node connectivity and account permissioning requirements in Sections § 7.1.1 Nodes, § 7.1.2 Ethereum Accounts, and § 7.1.3 Additional Permissioning Requirements.

Enterprise Ethereum client obtaining Privacy Level B demonstrate increased interoperability with the public Ethereum ecosystem and other Enterprise Ethereum client.

8.1.4.3 Privacy Level A

Privacy Level A compliance is achieved by providing support for the requirements of Privacy Level B, plus implementing all the *SHOULD* and *SHOULD NOT* requirements in Section § 8.1.3 Private Transactions.

Enterprise Ethereum clients implementing Privacy Level A demonstrate increased security and privacy protections for their users. Privacy Level A is considered best practice for Enterprise Ethereum clients and is strongly encouraged.

The following table summarizes the three privacy level requirements.

Table 3 Summary of Privacy Levels

Privacy Level	Description	Implementations Provide Support For
C	Baseline privacy and permissioning	All the *MUST* peer node connectivity and permissioning requirements from Sections § 7.1.1 Nodes and § 7.1.2 Ethereum Accounts, and either: • All *MUST* and *MUST NOT* Restricted private transaction requirements in Section § 8.1.3 Private Transactions, or • All *MUST* and *MUST NOT* Unrestricted private transaction requirements in Section § 8.1.3 Private Transactions.
B	Best practice permissioning	Privacy Level C plus all the *SHOULD* peer node connectivity and permissioning requirements from Sections § 7.1.1 Nodes, § 7.1.2 Ethereum Accounts, and § 7.1.3 Additional Permissioning Requirements.
A	Best practice privacy and permissioning	Privacy Level B and all the *SHOULD* and *SHOULD NOT* requirements in Section § 8.1.3 Private Transactions.

8.2 Scaling Sublayer

This section is non-normative.

Enterprise Ethereum blockchains will likely have demands placed on them to handle higher volume transaction rates and potentially computationally heavy tasks. Various scaling methods can be employed to increase transaction processing rates.

8.2.1 On-chain (Layer 1 and Layer 2)

Techniques to improve scalability are valuable for blockchains with high transaction throughput requirements combined with a desire or need to keep the processing on the blockchain.

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, like using smart contracts, [Plasma], or [state-channels], facilitate increased transactions speeds without changing the underlying Ethereum protocol. For more information, see [Layer2-Scaling-Solutions].

8.2.2 Off-chain (Layer 2 Compute)

Off-chain (layer 2 compute) scaling is where processing is executed externally to the Ethereum blockchain to facilitate increased transaction speeds. For example, offloading computationally intensive tasks to one or more Trusted Computing services.

Enterprise Ethereum clients could provide off-chain scaling functionality by implementing requirement EXEC-050.

8.2.3 Performance

Performance is the total effectiveness of a system, including overall throughput, individual transaction time, and system availability. For Enterprise Ethereum, this refers to the overall performance of the Enterprise Ethereum blockchain. Ideally, increased usage of the Enterprise Ethereum blockchain does not degrade its performance.

> Example: EEA Certification
>
> Certificates of Certification could require minimum transaction rates in terms of [ERC-20] smart contract executions per second, or other measures.

9. Core Blockchain Layer

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. Different consensus algorithms are available for both public Ethereum and Enterprise Ethereum.

Enterprise Ethereum clients can provide additional consensus algorithms for operations within their private ***consortium network*** (an Ethereum blockchain, either public Ethereum or Enterprise Ethereum, which is not part of the Ethereum MainNet).

> Example: Consensus Algorithms
>
> An example public consensus algorithm is the Proof of Work (PoW) algorithm, which is described in the [Ethereum-Yellow-Paper]. Over time, PoW is likely to be phased out from use and replaced with new methods of consensus. Other example consensus algorithms include Istanbul [Byzantine-Fault-Tolerant] (IBFT) [EIP-650], [RAFT], and Proof of Elapsed Time [PoET].

The Execution sublayer implements the ***Ethereum Virtual Machine*** (EVM), which is a runtime computing environment for the execution of smart contracts on Ethereum. Each client running on a node operates an EVM. Also implemented at this layer is Ethereum-flavored WebAssembly [eWASM], its instruction set, and other computational capabilities as required.

Smart contracts are computer programs that, in an Ethereum context, are executable on the EVM. Smart contracts can be written in higher-level programming languages that compile to EVM bytecode. Smart contracts are most often used to implement a contract between parties where the execution is guaranteed and auditable to the level of security provided by Ethereum itself.

A ***precompiled contract*** is a smart contract compiled from its source language to EVM bytecode and stored by an Ethereum node for later execution.

Finally, the Storage and Ledger sublayer is provided to store the blockchain state, such as smart contracts for later execution. This sublayer follows blockchain security paradigms such as using cryptographically hashed tries, an UTXO model, or at-rest-encrypted key-value stores.

Unspent Transaction Output (UTXO) is output from a transaction that can be spent as input for a new transaction.

9.1 Storage and Ledger Sublayer

To operate a client on the Ethereum MainNet, and to support optional off-chain operations, local data storage is required. For example, Enterprise Ethereum clients can locally cache the results from a trusted oracle or store information related to systems extensions beyond the scope of this Specification.

[C] STOR-030: Enterprise Ethereum clients providing support for multiple blockchains (for example, more than one Enterprise Ethereum blockchain, or a public network) *MUST* store data related to restricted private transactions for those blockchains in private state dedicated to the relevant blockchain.

Private State is the state data that is not shared in the clear in the globally replicated state tree. This data can represent bilateral or multilateral arrangements between parties, for example in private

transactions.

[P] STOR-040: Enterprise Ethereum clients *SHOULD* permit a smart contract operating on private state to access private state created by other smart contracts involving the same parties to the transaction.

[P] STOR-050: Enterprise Ethereum clients *MUST NOT* permit smart contract operating on private state to access private state created by other smart contracts involving different parties to the transaction.

[P] STOR-070: If an Enterprise Ethereum client stores private state persistently, it *SHOULD* protect the data using an Authenticated Encryption with Additional Data (AEAD) algorithm, such as one described in [RFC5116].

9.2 Execution Sublayer

[P] EXEC-010: Enterprise Ethereum clients *MUST* provide a smart contract execution environment implementing the public Ethereum EVM op-code set [EVM-Opcodes].

[P] EXEC-020: Enterprise Ethereum clients *MAY* provide a smart contract execution environment extending the public Ethereum EVM op-code set [EVM-Opcodes].

[P] EXEC-030: Enterprise Ethereum clients *SHOULD* support the ability to synchronize their public state with the public state held by other public Ethereum nodes.

[P] EXEC-040: Enterprise Ethereum clients *SHOULD* support compilation, storage, and execution of precompiled contracts.

Trusted Computing ensures only authorized parties can execute smart contracts on an execution environment available to a given Enterprise Ethereum blockchain.

[C] EXEC-050: Enterprise Ethereum clients *MAY* support off-chain Trusted Computing

Multiple encryption techniques can be used to secure Trusted Computing and private state.

[C] EXEC-060: Enterprise Ethereum clients *SHOULD* provide configurable encryption options for Enterprise Ethereum blockchains.

9.2.1 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.

[P] FINL-010: When a deterministic consensus algorithm is used, Enterprise Ethereum clients *SHOULD* treat transactions as final after a defined interval or event. For example, after a defined time period has elapsed, or after a defined number of blocks were created since the transaction was included in a block.

9.3 Consensus Sublayer

A common consensus algorithm implemented by all Enterprise Ethereum clients is necessary to ensure interoperability between clients.

[Byzantine-Fault-Tolerant] consensus algorithms ensure a certain proportion of malfunctioning nodes performing voting, block-making, or validation roles do not pose a critical risk to the blockchain. This makes them an excellent choice for many blockchains. The Technical Specification Working Group are considering existing and new Byzantine-Fault-Tolerant consensus algorithms, primarily those related to IBFT [EIP-650], with the goal of adopting the outcomes of that work as a required consensus algorithm as soon as possible.

[P] CONS-030: One or more consensus algorithms *SHOULD* allow operations as part of an Enterprise Ethereum blockchain.

[P] CONS-050: Enterprise Ethereum clients *MAY* implement multiple consensus algorithms and use them on sidechain networks.

A ***sidechain*** is a separate Ethereum blockchain operating on the Enterprise Ethereum blockchain's nodes. A sidechain can be public or private and can also operate on a consortium network.

[P] CONS-093: Enterprise Ethereum clients *MUST* support the "Clique" Proof of Authority consensus algorithm. [EIP-225]

[P] CONS-110: Enterprise Ethereum clients *MUST* provide the ability to specify the consensus algorithms, through network configuration, to be used for each public blockchain, private blockchain, and sidechain in use.

10. Network Layer

The Network layer consists of an implementation of a peer-to-peer networking protocol allowing nodes to communicate with each other. For example, the ***DEVp2p*** protocol, which defines messaging between nodes to establish and maintain a communications channel for use by higher layer protocols.

Additional organizational peer-to-peer protocols will be formalized over time to provide the communications needed to supplement higher levels of the stack.

10.1 Network Protocol Sublayer

Network protocols define how nodes communicate with each other.

[P] PROT-010: Nodes *MUST* be identified and advertised using the Ethereum enode URL format [enode].

[P] PROT-015: Enterprise Ethereum clients *MUST* implement the DEVp2p Node Discovery protocol [DEVp2p-Node-Discovery].

The [Ethereum-Wire-Protocol] defines higher layer protocols, known as ***capability protocols***, for messaging between nodes to exchange status, including block and transaction information. [Ethereum-Wire-Protocol] messages are sent and received over an already established DEVp2p connection between nodes.

[P] PROT-020: Enterprise Ethereum clients *MUST* use the DEVp2p Wire Protocol [DEVp2p-Wire-Protocol] for messaging between nodes to establish and maintain a communications channel for use by capability protocols.

[P] PROT-040: Enterprise Ethereum clients *MAY* add new protocols or extend existing Ethereum protocols.

[P] PROT-050: To minimize the number of point-to-point connections needed between private nodes, some private nodes *SHOULD* be capable of relaying private transaction data to multiple other private nodes.

> Example: Relaying Private Transaction Data
>
> Multi-party private smart contracts and transactions do not require direct connectivity between all parties because this is very impractical in enterprise settings, especially when many parties are allowed to transact. Nodes common to all parties (for example, voters or blockmakers acting as bootnodes to all parties, and as backup or disaster recovery nodes) are intended to function as gateways to synchronize private smart contracts transparently. Transactions on private smart contracts could then be transmitted to all participating parties in the same way.

[P] PROT-060: Enterprise Ethereum clients *SHOULD* implement the [Whisper-protocol].

11. Anti-spam

This section refers to mechanisms to prevent the Enterprise Ethereum blockchain being degraded with a flood of intentional or unintentional transactions. This might be realized through interfacing with an external security manager, as described in Section § 7.2.1 Enterprise Management Systems, or implemented by the client, as described in the following requirement.

[P] SPAM-010: Enterprise Ethereum clients *SHOULD* provide effective anti-spam mechanisms so attacking nodes or accounts (either malicious, buggy, or uncontrolled) can be quickly identified and stopped.

> Example: Anti-spam Mechanisms
>
> Anti-spam mechanisms might include:
>
> - Stopping parties attempting to issue transactions above a threshold volume.
> - Providing a mechanism to enforce a cost for gas, so transacting parties have to acquire and pay for (or destruct) private ether to transact.
> - Having a dynamic cost of gas based on activity intensity.

12. Cross-client Compatibility

Cross-client compatibility refers to the ability of an Enterprise Ethereum blockchain to operate with different clients.

This Specification extends the capabilities and interfaces of public Ethereum. The requirements relating to supporting and extending the public Ethereum opcode set are outlined in Section § 9.2 Execution Sublayer.

[P] XCLI-005: Features of public Ethereum implemented in Enterprise Ethereum clients *MUST* be compatible with the Constantinople hard fork of Ethereum [EIP-1013], which occured on 28 February, 2019.

Future versions of this Specification are expected to align with newer versions of public Ethereum as they are deployed.

[P] XCLI-020: Enterprise Ethereum clients *MAY* extend the public Ethereum APIs. To maintain compatibility, Enterprise Ethereum clients *SHOULD* ensure these new features are a superset of the public Ethereum APIs.

> Example: Extensions to the Public Ethereum API
>
> Extensions to public Ethereum APIs could include peer-to-peer APIs, the [JSON-RPC-API] over IPC, HTTP/HTTPS, or websockets.

[P] XCLI-030: Enterprise Ethereum clients *MUST* implement the gas mechanism specified in the [Ethereum-Yellow-Paper].

[P] XCLI-040: Gas price *MAY* be set to zero.

[P] XCLI-050: Enterprise Ethereum clients *MUST* implement the eight precompiled contracts defined in Appendix E of the [Ethereum-Yellow-Paper]:

- `ecrecover`
- `sha256hash`
- `ripemd160hash`
- `dataCopy`
- `bigModExp`
- `bn256Add`
- `bn256ScalarMul`
- `bn256Pairing`

> NOTE
>
> Sample [implementation-code-in-Golang], as part of the Go-Ethereum client is available from the Go-Ethereum source repository [geth-repo]. Be aware this code uses a combination of GPL3 and LGPL3 licenses

Cross-client compatibility extends to the different message encoding formats used by clients.

[P] XCLI-060: Enterprise Ethereum clients *MUST* support the Contract Application Binary Interface ([ABI]) for interacting with smart contracts.

[P] XCLI-070: Enterprise Ethereum clients *MUST* support Recursive Length Prefix ([RLP]) encoding for binary data.

13. Synchronization and Disaster Recovery

Synchronization and disaster recovery refers to how node in a blockchain behave when connecting for the first time or reconnecting.

Various techniques can help do this efficiently. For an Enterprise Ethereum blockchain with few copies, off-chain backup information can be important to ensure the long-term existence of the information stored. A common backup format helps increase client interoperability.

A. Additional Information

A.1 Terms defined in this specification

authorizedUser

capability protocols

Client requirements

Consensus

consensus algorithm

consortium network

ĐApps

DEVp2p

Enterprise Ethereum

Enterprise Ethereum blockchains

Enterprise Ethereum client

Ethereum account

Ethereum JSON-RPC API

Ethereum Name Service

Ethereum Virtual Machine

Finality

Formal verification

Gas

genesis block

Groups

hard fork block

hard fork

Hardware Security Module

inter-chain mediators

Integration libraries

Ethereum MainNet

memberGroups

Metadata

network

Network Configuration

node list

node

Off-chain (layer 2 compute) scaling

On-chain (layer 1) scaling

On-chain (layer 2) scaling

On-chain privacy mechanisms

oracle

organization

Payload data

Performance

permissioning decider

Permissioning

precompiled contract

Privacy

Private State

private transaction manager

Protocol requirements

Public Ethereum

Restricted private transactions

Roles

Scaling

sidechain

Smart contract languages

Smart contracts

private transaction

transaction

Off-chain trusted computing

Unrestricted private transactions

User

Unspent Transaction Output

Wallets zero-knowledge proof

A.2 Summary of Requirements

This section summarizes all of the requirements in this Specification into one section.

[C] DAPP-010: ĐApps *MAY* use the extensions to the Ethereum JSON-RPC API defined in this Specification.

[P] SMRT-030: Enterprise Ethereum clients *MUST* support smart contracts of at least 24,576 bytes in size.

[P] SMRT-040: Enterprise Ethereum clients *MUST* read and enforce a size limit for smart contracts from the current network configuration (e.g. the genesis block).

[P] SMRT-050: If no contract size limit is specified in a genesis block or subsequent network configuration, Enterprise Ethereum clients *MUST* enforce a size limit on smart contracts of 24,576 bytes.

[C] NODE-010: Enterprise Ethereum clients *MUST* provide the ability to specify at startup a list of static peer nodes to establish peer-to-peer connections with.

[C] NODE-020: Enterprise Ethereum clients *MUST* provide the ability to enable or disable peer-to-peer node discovery.

[P] NODE-030: Enterprise Ethereum clients *MUST* provide the ability to specify a whitelist of the nodes permitted to connect to a node.

[P] NODE-040: Enterprise Ethereum clients *MAY* provide the ability to specify a blacklist of the nodes not permitted to connect to a node.

[P] NODE-050: It *MUST* be possible to specify the node whitelist required by **NODE-030** through a transaction into a smart contract.

[P] NODE-060: It *MUST* be possible to specify the node blacklist allowed by **NODE-040** (if implemented) through a transaction into a smart contract.

[P] NODE-080: NODE-080: Enterprise Ethereum clients *MUST* provide the ability to specify node identities in a way aligned with the concept of groups.

[P] NODE-090: Enterprise Ethereum clients *MUST* document which metadata parameters (if any) can affect transaction ordering, and what the effects are.

[P] PART-010: Enterprise Ethereum clients *MUST* provide the ability to specify a whitelist of accounts that are permitted to transact with the blockchain.

[P] PART-015: Enterprise Ethereum clients *MUST* be able to verify that accounts are present on the whitelist required by **PART-010:** when adding transactions from the account to a block, and when verifying a received block containing transactions created by that account.

[P] PART-020: Enterprise Ethereum clients *MAY* provide the ability to specify a blacklist of accounts that are not permitted to transact with the blockchain.

[P] PART-025: Enterprise Ethereum clients *MUST* be able to verify that accounts are not present on the blacklist allowed by **PART-020:** (if implemented) when adding transactions from the account to a block, and when verifying a received block containing transactions created by that account.

[P] PART-030: It *MUST* be possible to specify the account whitelist required by **PART-010:** through a transaction into a smart contract.

[P] PART-040: It *MUST* be possible to specify the account blacklist allowed by **PART-020:** (if implemented) through a transaction into a smart contract.

[P] PART-050: Enterprise Ethereum clients *MUST* provide a mechanism to identify organizations that participate in the Enterprise Ethereum blockchain.

[P] PART-055 Enterprise Ethereum clients *MUST* support anonymous accounts.

[P] PART-060: Enterprise Ethereum clients *MUST* provide the ability to specify accounts in a way aligned with the concepts of groups and roles.

[P] PART-070: Enterprise Ethereum clients *MUST* be able to authorize the types of transactions an account can submit, providing separate permissioning for the ability to:

- Deploy smart contracts.
- Call functions that change the state of specified smart contracts.
- Perform a simple value transfer between specified accounts.

[C] PERM-075: Enterprise Ethereum clients *MUST* allow organizations to be nested to a minimum of at least 3 levels (i.e. an organization containing an organization that contains another organization).

[C] PERM-020: Enterprise Ethereum clients *SHOULD* provide the ability for network configuration to be updated at run time without the need to restart.

[C] PERM-040: Enterprise Ethereum clients *MAY* support local key management, allowing users to secure their private keys.

[C] PERM-050: Enterprise Ethereum clients *MAY* support secure interaction with an external key management system for key generation and secure key storage.

[P] JRPC-010: Enterprise Ethereum clients *MUST* provide support for the following methods of the Ethereum JSON-RPC API:

- `net_version`
- `net_peerCount`
- `net_listening`
- `eth_protocolVersion`
- `eth_syncing`
- `eth_coinbase`
- `eth_hashrate`
- `eth_gasPrice`
- `eth_accounts`
- `eth_blockNumber`
- `eth_getBalance`
- `eth_getStorageAt`
- `eth_getTransactionCount`
- `eth_getBlockTransactionCountByHash`
- `eth_getBlockTransactionCountByNumber`
- `eth_getCode`
- `eth_sendRawTransaction`
- `eth_call`
- `eth_estimateGas`
- `eth_getBlockByHash`
- `eth_getBlockByNumber`
- `eth_getTransactionByHash`
- `eth_getTransactionByBlockHashAndIndex`
- `eth_getTransactionByBlockNumberAndIndex`
- `eth_getTransactionReceipt`
- `eth_getUncleByBlockHashAndIndex`
- `eth_getUncleByBlockNumberAndIndex`
- `eth_getLogs`.

[P] JRPC-007: Enterprise Ethereum clients *SHOULD* implement [JSON-RPC-API] methods to be backward compatible with the definitions given in version 27e37ee of the Ethereum JSON-RPC API reference [JSON-RPC-API-v27e37ee], unless breaking changes have been made and widely implemented for the health of the ecosystem. For example, to fix a major security or privacy problem.

[C] JRPC-015: Enterprise Ethereum clients *MUST* provide the capability to accept and respond to JSON-RPC method calls over a websocket interface.

[C] JRPC-040: Enterprise Ethereum clients *MUST* provide an implementation of the `debug_traceTransaction` method [debug-traceTransaction] from the Go Ethereum Management API.

[C] JRPC-050: Enterprise Ethereum clients *MUST* implement the [JSON-RPC-PUB-SUB] API.

[P] JRPC-070: Enterprise Ethereum clients implementing additional nonstandard subscription types for the [JSON-RPC-PUB-SUB] API *MUST* prefix their subscription type names with a namespace prefix other than `eea_`.

[P] JRPC-080: The [JSON-RPC] method name prefix `eea_` *MUST* be reserved for future use for RPC methods specific to the EEA.

[P] JRPC-020: Enterprise Ethereum clients *MUST* provide one of the following sets of extensions to create private transaction types defined in Section § 8.1.3 Private Transactions:

- `eea_sendTransactionAsync` and `eea_sendTransaction`, or
- `eea_sendRawTransactionAsync` and `eea_sendRawTransaction`.

[P] JRPC-030: The `eea_sendTransactionAsync`, `eea_sendTransaction`, `eea_sendRawTransactionAsync`, and `eea_sendRawTransaction` methods *MUST* respond with a [JSON-RPC] error response when an unimplemented private transaction type is requested. The error response *MUST* have the *code* `-50100` and the *message* `Unimplemented private transaction type`.

[P] PERM-200: Enterprise Ethereum clients *MUST* call the `connectionAllowed` function, as specified in Section § 7.3.3.1.1 Node Permissioning Functions, to determine whether a connection with another node is permitted, and any restrictions to be placed on that connection.

[P] PERM-210: When checking the response to `connectionAllowed`, if any unknown permissioning bits are found to be zero, Enterprise Ethereum clients *MUST* reject the connection.

[P] PERM-220: On receipt of a `NodePermissionsUpdated` event containing an 'addsRestrictions' property with value 'true', Enterprise Ethereum clients *MUST* close any network connections that are no longer permitted, and impose newly added restrictions on any network connections that have had restrictions added.

[P] PERM-230: On receipt of a `NodePermissionsUpdated` event containing an 'addsRestrictions' property with value 'false', Enterprise Ethereum clients *MUST* allow future actions that are now permitted.

[P] PERM-240: When validating or mining a block, Enterprise Ethereum clients *MUST* call the `transactionAllowed` function, as specified in Section § 7.3.3.2.1 Account Permissioning Smart Contract Interface Function, with worldstate as at the block's parent, to determine if a transaction is permitted in a block.

[C] ORCL-010: Enterprise Ethereum clients *SHOULD* provide the ability to securely interact with oracles to send and receive external off-chain information.

[P] PRIV-010: Enterprise Ethereum clients *MUST* support private transactions using either restricted private transactions or unrestricted private transactions.

When implementing restricted private transactions:

- **[P] PRIV-020:** Enterprise Ethereum clients *MUST* support encryption of the payload data when stored in restricted private transactions.
- **[P] PRIV-030:** Enterprise Ethereum clients *MUST* support encryption of the payload data when in transit in restricted private transactions.
- **[P] PRIV-040:** Enterprise Ethereum clients *MAY* support encryption of the metadata when stored in restricted private transactions.
- **[P] PRIV-050:** Enterprise Ethereum clients *MAY* support encryption of the metadata when in transit in restricted private transactions.
- **[P] PRIV-060:** Nodes that relay a restricted private transaction but are not party to that transaction, *MUST NOT* store payload data.
- **[P] PRIV-070:** Nodes that relay a restricted private transaction but are not party to that transaction *SHOULD NOT* store the metadata.
- **[P] PRIV-080:** The implementation of the `eea_sendTransactionAsync`, `eea_sendTransaction`, `eea_sendRawTransactionAsync`, or `eea_sendRawTransaction` methods (see Section § 7.3.2 Extensions to the JSON-RPC API) with the `restriction` parameter set to `restricted`, *MUST* result in a restricted private transaction.

When implementing unrestricted private transactions:

- **[P] PRIV-090:** Enterprise Ethereum clients *SHOULD* support encryption of the recipient identity when stored in unrestricted private transactions.
- **[P] PRIV-100:** Enterprise Ethereum clients *SHOULD* support encryption of the sender identity when stored in unrestricted private transactions.

- **[P] PRIV-110:** Enterprise Ethereum clients *SHOULD* support encryption of the payload data when stored in unrestricted private transactions.
- **[P] PRIV-120:** Enterprise Ethereum clients *MUST* support encryption of the payload data when in transit in unrestricted private transactions.
- **[P] PRIV-130:** Enterprise Ethereum clients *MAY* support encryption of the metadata when stored in unrestricted private transactions.
- **[P] PRIV-140:** Enterprise Ethereum clients *MAY* support encryption of the metadata when in transit in unrestricted private transactions.
- **[P] PRIV-150:** Nodes that relay an unrestricted private transaction but are not party to that transaction *MAY* store payload data.
- **[P] PRIV-160:** Nodes that relay an unrestricted private transaction but are not party to that transaction, *MAY* store the metadata.
- **[P] PRIV-170:** The implementation of the `eea_sendTransactionAsync`, `eea_sendTransaction`, `eea_sendRawTransactionAsync`, or `eea_sendRawTransaction` methods (see Section § 7.3.2 Extensions to the JSON-RPC API) with the `restriction` parameter set to `unrestricted` *MUST* result in an unrestricted private transaction.
- **[P] PRIV-210:** Enterprise Ethereum clients' implementations of unrestricted private transactions *MUST* provide the ability for nodes to achieve global consensus.

[P] PRIV-180: Enterprise Ethereum clients *SHOULD* be able to extend the set of parties to a private transaction (or forward the private transaction in some way).

[P] PRIV-190: Enterprise Ethereum clients *SHOULD* provide the ability for nodes to achieve consensus on their mutually private transactions.

[C] STOR-030: Enterprise Ethereum clients providing support for multiple blockchains (for example, more than one Enterprise Ethereum blockchain, or a public network) *MUST* store data related to restricted private transactions for those blockchains in private state dedicated to the relevant blockchain.

[P] STOR-040: Enterprise Ethereum clients *SHOULD* permit a smart contract operating on private state to access private state created by other smart contracts involving the same parties to the transaction.

[P] STOR-050: Enterprise Ethereum clients *MUST NOT* permit smart contract operating on private state to access private state created by other smart contracts involving different parties to the transaction.

[P] STOR-070: If an Enterprise Ethereum client stores private state persistently, it *SHOULD* protect the data using an Authenticated Encryption with Additional Data (AEAD) algorithm, such

as one described in [RFC5116].

[P] EXEC-010: Enterprise Ethereum clients *MUST* provide a smart contract execution environment implementing the public Ethereum EVM op-code set [EVM-Opcodes].

[P] EXEC-020: Enterprise Ethereum clients *MAY* provide a smart contract execution environment extending the public Ethereum EVM op-code set [EVM-Opcodes].

[P] EXEC-030: Enterprise Ethereum clients *SHOULD* support the ability to synchronize their public state with the public state held by other public Ethereum nodes.

[P] EXEC-040: Enterprise Ethereum clients *SHOULD* support compilation, storage, and execution of precompiled contracts.

[C] EXEC-050: Enterprise Ethereum clients *MAY* support off-chain Trusted Computing

[C] EXEC-060: Enterprise Ethereum clients *SHOULD* provide configurable encryption options for Enterprise Ethereum blockchains.

[P] FINL-010: When a deterministic consensus algorithm is used, Enterprise Ethereum clients *SHOULD* treat transactions as final after a defined interval or event. For example, after a defined time period has elapsed, or after a defined number of blocks were created since the transaction was included in a block.

[P] CONS-030: One or more consensus algorithms *SHOULD* allow operations as part of an Enterprise Ethereum blockchain.

[P] CONS-050: Enterprise Ethereum clients *MAY* implement multiple consensus algorithms and use them on sidechain networks.

[P] CONS-093: Enterprise Ethereum clients *MUST* support the "Clique" Proof of Authority consensus algorithm. [EIP-225]

[P] CONS-110: Enterprise Ethereum clients *MUST* provide the ability to specify the consensus algorithms, through network configuration, to be used for each public blockchain, private blockchain, and sidechain in use.

[P] PROT-010: Nodes *MUST* be identified and advertised using the Ethereum enode URL format [enode].

[P] PROT-015: Enterprise Ethereum clients *MUST* implement the DEVp2p Node Discovery protocol [DEVp2p-Node-Discovery].

[P] PROT-020: Enterprise Ethereum clients *MUST* use the DEVp2p Wire Protocol [DEVp2p-Wire-Protocol] for messaging between nodes to establish and maintain a communications channel for use by capability protocols.

[P] PROT-040: Enterprise Ethereum clients *MAY* add new protocols or extend existing Ethereum protocols.

[P] PROT-050: To minimize the number of point-to-point connections needed between private nodes, some private nodes *SHOULD* be capable of relaying private transaction data to multiple other private nodes.

[P] PROT-060: Enterprise Ethereum clients *SHOULD* implement the [Whisper-protocol].

[P] SPAM-010: Enterprise Ethereum clients *SHOULD* provide effective anti-spam mechanisms so attacking nodes or accounts (either malicious, buggy, or uncontrolled) can be quickly identified and stopped.

[P] XCLI-005: Features of public Ethereum implemented in Enterprise Ethereum clients *MUST* be compatible with the Constantinople hard fork of Ethereum [EIP-1013], which occured on 28 February, 2019.

[P] XCLI-020: Enterprise Ethereum clients *MAY* extend the public Ethereum APIs. To maintain compatibility, Enterprise Ethereum clients *SHOULD* ensure these new features are a superset of the public Ethereum APIs.

[P] XCLI-030: Enterprise Ethereum clients *MUST* implement the Gas mechanism specified in the [Ethereum-Yellow-Paper].

[P] XCLI-040: Gas price *MAY* be set to zero.

[P] XCLI-050: Enterprise Ethereum clients *MUST* implement the eight precompiled contracts defined in Appendix E of the [Ethereum-Yellow-Paper]:

- `ecrecover`
- `sha256hash`
- `ripemd160hash`
- `dataCopy`
- `bigModExp`
- `bn256Add`
- `bn256ScalarMul`
- `bn256Pairing`

[P] XCLI-060: Enterprise Ethereum clients *MUST* support the Contract Application Binary Interface ([ABI]) for interacting with smart contracts.

[P] XCLI-070: Enterprise Ethereum clients *MUST* support Recursive Length Prefix ([RLP]) encoding for binary data.

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.

We would also like to thank former editors David Hyland-Wood (version 1) and Daniel Burnett (version 2), and former EEA Technical Director, the late and missed Clifton Barber, for their work on previous versions of this specification.

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 Changes

This section outlines substantive changes made to the specification since version 2:

- Substantial update to the text of many conformance requirements to provide consistent terminology and phrasing.
- Add requirement PERM-075 requiring support for at minimum three levels of nesting organizations. Updated PART-050 to require a mechanism to identify organizations.
- Add a new section showing an example permissioning model and example permissioning management smart contracts. This is based on re-instating content from the Network Permissioning Using Smart Contracts section removed with Pull Request 477.
- Minor wording update to ORCL-010 to change 'real world' to 'external off-chain'. Minor wording update to PRIV-080 and PRIV-170 to remove the JSON-RPC-API reference link.
- Changes to resolve Issue 393 including definitions for organizations (intead of enterprises), Enterprise Ethereum client (a client), nodes (an instance of a client), Ethereum account (an account), and users, then appropriate updates for these terms throughout the document.
- Replace Network Permisioning Using Smart Contracts section and **PERM-070** with a new section and add requirement PERM-200 requiring clients to call `connectionAllowed` functions before a connection is established with another node. And add requirement PERM-210 requiring clients to call `transactionAllowed` functions before a transaction is permitted.

- Editorial changes in Sections § 8.1.3 Private Transactions and § 8.1.4 Privacy Levels to clarify the difference between restricted and unrestricted private transactions, and clarify the descriptions for the privacy levels.
- All term definitions removed from the table of definitions and integrated throughout the Specification. The table of definitions, previously located in Appendix A "Additional Information", is removed. Also moved the introductory information for each of the layers in the stack from the "Enterprise Ethereum Concepts" section to the relevant detailed section. Changed the name of "Enterprise Ethereum Concepts" to "Enterprise Ethereum Architecture".
- Editorial changes to PROT-020, and change the requirement from *SHOULD* to *MUST*.
- clarify requirements PRIV-060, PRIV-070, PRIV-150, PRIV-160, and PRIV-180 to refer to "parties to a transaction".
- Editorial changes to PROT-020, and change the requirement from *SHOULD* to *MUST*.
- Remove the unnecessary permissioning prose in the Section 7.1 Privacy Sublayer introduction.
- Remove requirement NODE-070 because it is captured by the updated requirements NODE-080 and PERM-070.
- Add requirement PROT-015 mandating support for [DEVp2p-Node-Discovery] and requiring support for [DEVp2p-Wire-Protocol] changed from *SHOULD* to *MUST*.
- Remove requirement SCAL-010 as untestable, keeping prose explanation of the value of on-chain scalability solutions.
- Remove requirement CONS-100 as unnecessary.
- Remove requirement CONS-020 as unnecessary.
- Updated requirement XCLI-005 to require compatibility with Constantinople hard fork.
- Add requirement CONS-093 to require that EEA clients support the "Clique" Proof of Authority [EIP-225] consensus algorithm.
- Remove requirement STOR-060 as unnecessary.
- Remove requirement CONS-080 as unnecessary.
- Clarified that XCLI-060 and XCLI-070 are protocol requirements.
- Fixes duplication across Trusted Compute requirements by converting OFFCH-010 and SCAL-020 to prose and linking via a cross reference to EXEC-050. Updated EXEC-050 to include the words 'off-chain'.

- Remove unnecessary words 'and smart contracts' from requirements SCAL-010 and SCAL-020.
- Minor edit in PART-060 to remove the word "usual".
- Add requirements PART-015 and PART-025, that require verification of transaction senders against the whitelist and blacklist when including transactions in a block, and when verifying transactions in a received block.
- Remove requirement JRPC-060 which stated Clients *MAY* implement additional subscription types for the [JSON-RPC-PUB-SUB] API. This adds little value.
- Remove requirement JRPC-011 which stated Clients *MAY* provide implementations of other methods. This added little value.
- Update requirement PART-070 to require transaction type permissioning (*MUST* instead of *SHOULD*), and for functions to be able to change the state of specified smart contacts.
- Remove requirement PERM-060 as it is effectively covered by PERM-050. Converted former PERM-060 requirement into prose.
- Update requirements PART-010 and PART-020 for permission to 'transact with the blockchain', instead of 'submit transactions'.
- Change NODE-080 grouping requirement to *MUST* and reword to use "groups" instead of "cluster".
- Change PART-030 and PART-040 to require participant permissioning uses smart contracts.
- Change NODE-050 and NODE-060 to require node permissioning uses smart contracts.
- Move requirements SMRT-010, SMRT-020, ILIB-010, ENTM-010, ENTM-020, ENTM-030, ENTM-040, ENTM-050, ENTM-060, ICHN-010, and STOR-020 to the Enterprise Ethereum Alliance Implementation Guidance document because these requirements are external to an Enterprise Ethereum implementation. Remove requirement STOR-010 because it is superfluous.
- Change NODE-030 and NODE-040 to permit connections "to this node" (instead of saying "join the network").
- Remove requirement PRIV-200 which duplicated PART-070. Updated PART-070 to refer to "Ethereum Accounts" instead of "Participants".
- Remove requirement PERM-030, that allowed generic configuration.
- Remove requirement PERM-010, which contained generic statements on permissioning APIs. These are stated more specifically in NODE, PERM, and PART requirements.

- Remove requirement XCLI-010, that stated Enterprise Ethereum clients *SHOULD* be compatible with the public Ethereum network to the greatest extent possible. This will be replaced with more specific requirements.
- Remove requirements ACCT-010 and ACCT-020 which duplicated PART-010 and PART-030.
- Add `eea_sendRawTransactionAsync` JSON-RPC API method. Updates JRPC-020 to require implementation of `eea_sendTransaction` and `eea_sendTransactionAsync` methods or `eea_sendRawTransactionAsync` and `eea_sendRawTransaction` methods. Updates JPRC-030, PRIV-080, PRIV-170 to include the new `eea_sendRawTransactionAsync` method.
- Add requirements SMRT-030, SMRT-040, and SMRT-050, to require variable contract size limits.
- Remove requirement PROT-030, that stated implementations should support eth/62 and eth/63 protocols.
- Update requirement NODE-090 to require documentation of any metadata that affects transaction ordering.
- Add `eea_sendRawTransaction` JSON-RPC API method. Updates JRPC-020 to require implementation of `eea_sendTransaction` and `eea_sendTransactionAsync` methods or `eea_sendRawTransaction` method. Updates JPRC-030, PRIV-080, PRIV-170 to include the new `eea_sendRawTransaction` method.
- Update requirement JRPC-030 to require a JSON RPC error code for unimplemented Private Transactions, instead of an HTTP 50x response.
- Fixes multiple index parameter name inconsistencies with comments.
- Add requirement PRIV-210 to require that unrestricted private transactions allow all nodes to form consensus.
- Update requirement PRIV-120 to require obfuscation (and not offer "masking" as an alternative.
- Add requirements XCLI-060 and XCLI-070 to require support for Application Binary Interface [ABI] and Recursive Length Prefix [RLP].

B. References

B.1 Normative references

[ABI]

Contract ABI Specification. Ethereum Foundation. URL: https://solidity.readthedocs.io/en/develop/abi-spec.html

[Byzantine-Fault-Tolerant]

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

[debug-traceTransaction]

debug_traceTransaction. URL: https://github.com/ethereum/go-ethereum/wiki/Management-APIs

[DEVp2p-Node-Discovery]

Node Discovery Protocol. URL: https://github.com/ethereum/devp2p/blob/master/rlpx.md

[DEVp2p-Wire-Protocol]

ĐΞVp2p Wire Protocol. URL: https://github.com/ethereum/devp2p/blob/master/caps/eth.md

[EIP-1013]

Hardfork Meta: Constantinople. Ethereum Foundation. URL: https://eips.ethereum.org/EIPS/eip-1013

[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

[EIP-650]

Istanbul Byzantine Fault Tolerance. Ethereum Foundation. URL: https://github.com/ethereum/EIPs/issues/650

[enode]

Ethereum enode URL format. Ethereum Foundation. URL: https://github.com/ethereum/wiki/wiki/enode-url-format

[Ethereum-Wire-Protocol]

Ethereum Wire Protocol. URL: https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol

[Ethereum-Yellow-Paper]

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

[EVM-Opcodes]

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

[eWASM]

Ethereum-flavored WebAssembly. URL: https://github.com/ewasm/design

[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]

The application/json Media Type for JavaScript Object Notation (JSON). D. Crockford. IETF. July 2006. Informational. URL: https://tools.ietf.org/html/rfc4627

[JSON-RPC]

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

[JSON-RPC-API]

Ethereum JSON-RPC API. Ethereum Foundation. URL: https://github.com/ethereum/wiki/wiki/JSON-RPC

[JSON-RPC-API-v27e37ee]

Ethereum JSON-RPC API. Ethereum Foundation. URL: https://github.com/ethereum/wiki/wiki/JSON-RPC/27e37ee31ca29565fd8542ced0814fefc37a94fb

[JSON-RPC-PUB-SUB]

RPC PUB-SUB. Ethereum Foundation. URL: https://github.com/ethereum/go-ethereum/wiki/RPC-PUB-SUB

[LLL]

LLL Introduction. Ben Edgington. 2017. URL: http://lll-docs.readthedocs.io/en/latest/lll_introduction.html

[Nethereum]

Nethereum .NET Integration Library. Nethereum Open Source Community. URL: https://nethereum.com

[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

[RFC5116]

An Interface and Algorithms for Authenticated Encryption. D. McGrew. IETF. January 2008. Proposed Standard. URL: https://tools.ietf.org/html/rfc5116

[RLP]

Recursive Length Prefix. Ethereum Foundation. URL: https://github.com/ethereum/wiki/wiki/RLP

[sharding]

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

[Solidity]
The Solidity Contract-Oriented Programming Language. Ethereum Foundation. URL: https://github.com/ethereum/solidity

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

[web3.js]
Ethereum JavaScript API. Ethereum Foundation. URL: https://github.com/ethereum/web3.js

[web3j]
web3j Lightweight Ethereum Java and Android Integration Library. Conor Svensson. URL: https://web3j.io

[Whisper-protocol]
Whisper. Ethereum Foundation. URL: https://github.com/ethereum/wiki/wiki/Whisper

[X.800]
Security architecture for Open Systems Interconnection for CCITT applications. International Telecommunication Union. March 1991. URL: http://www.itu.int/rec/T-REC-X.800-199103-I/en

B.2 Informative references

[EEA-OC]
EEA Off-Chain Trusted Compute Specification V0.5. Enterprise Ethereum Alliance, Inc. URL: http://entethalliance.org/wp-content/uploads/2018/10/EEA_Off_Chain_Trusted_Compute_Specification_V0_5.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

[geth-repo]

Go-Ethereum. URL: https://github.com/ethereum/go-ethereum/

[implementation-code-in-Golang]

implementation code in Golang. URL: https://github.com/ethereum/go-ethereum/blob/master/core/vm/contracts.go#L50-L360

[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

[PoET]

Proof of Elapsed Time 1.0 Specification. Intel Corporation. 2015-2017. URL: https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html#

[RAFT]

Raft-based Consensus for Ethereum/Quorum. J.P. Morgan. URL: https://github.com/jpmorganchase/quorum/blob/master/raft/doc.md