ENTERPRISE ETHEREUM ALLIANCE

EEA-TS-0001-0 v1.00

2 May 2018

Enterprise Ethereum Alliance (EEA) – Enterprise Ethereum Client Specification

For additional information on Enterprise Ethereum Alliance (EEA) documents, contact the EEA Administration at https://entethalliance.org/contact/.

Revision History

Revision	Description	Date
v1.00	Initial Publication	2 May 2018

Abstract

This document specifies Enterprise Ethereum, a set of extensions to the public Ethereum blockchain to support the scalability, security, and privacy demands of enterprise deployments.

Status of this Document

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

This document has been reviewed by EEA Membership, Executive and Board, and is endorsed by the EEA Board as an EEA Standard. It is a stable document and may be used as reference material or cited from another document.

This specification was developed by the EEA Release 1 Development Ad Hoc as a designated subset of the EEA Technical Steering Committee for review, improvement, and publication as an EEA Standard.

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

Contents

1 Introduction

This section is informative.

This Specification defines Enterprise Ethereum, a system to enable enterprise-grade transactions on an Ethereum-based blockchain network. Enterprise Ethereum implementations make blockchain operations possible in enterprise production environments. Enterprise Ethereum extends and adapts various technologies and concepts of public Ethereum for enterprise deployments.

Enterprise Ethereum provides a set of extensions to public Ethereum to satisfy the performance, permissioning, and privacy demands of enterprise deployments. These principles are informally known as the "three Ps" of Enterprise Ethereum.

This Specification defines the interfaces to the external-facing components of Enterprise Ethereum (specifically excluding public Ethereum interfaces) and how they are intended to be used. Hence this document combines representative architecture information, and API information.

1.1 Terminology

The following table provides a list of terms and definitions used *in context* in this Specification.

Table 1 Terms and Definitions

Term	Definition
Client	The Enterprise Ethereum client software running on a node in a blockchain network. A client implements Enterprise Ethereum extensions.
Configuration	The settings made by a system operator, such as which consensus algorithm to use or which blockchain to join.
Consensus	Nodes on the blockchain reaching agreement about the current state of the blockchain.
Consensus Algorithm	An algorithm by which a given blockchain achieves consensus prior to an action being taken (for example, adding a block). Different blockchains might use different consensus algorithms, but all nodes of a given blockchain must agree to use the same consensus algorithm. Different consensus algorithms are available for both public Ethereum and Enterprise Ethereum networks.
Consortium Network	An Ethereum network, Enterprise or public, not part of the Ethereum MainNet.
ĐApp (Decentralized Application, or sometimes Distributed Application)	A software application 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.
DEVp2p	The DEV Peer to Peer (DEVp2p) Protocol defines messaging between Ethereum clients to establish and maintain a communications channel for use by higher layer protocols.
Enterprise Ethereum	Enterprise-grade additions to public Ethereum complying with this Specification.
Enterprise Ethereum Client	See client.
Enterprise Ethereum Extension	The portions of an Enterprise Ethereum system implementing the business logic requirements and interfaces of this Specification, over and above the functionality of public Ethereum.
Enterprise Participant	An enterprise participant is an organizational level entity (for example, a bank) that is a member of a network and is likely subject to a legal agreement or a set of rules governing that network. It is a managed group of individual actors with different roles, instead of a set of employees.
Ethereum	An open-source, public blockchain-based, distributed computing platform featuring smart contract (programming) functionality. [Ethereum]
Ethereum MainNet	The public Ethereum blockchain network with the network identifier of 1.

Term	Definition
Ethereum Name Service	A secure name service for public Ethereum to allow identification of Ethereum nodes by name instead of by address. It is conceptually similar to the Domain Name Service (DNS) for Internet-connected machines.
Ethereum Virtual Machine (EVM)	A runtime computing environment for the execution of smart contracts on Ethereum. Each node operates an EVM.
Finality	A guarantee that once a transaction is included in a block it will not be auto-reversed at any point in the future.
Formal Verification	Mathematical verification of the logical correctness of a smart contract in the context of the EVM.
Group	A collection of individual users that share a common set of privileges allowing them to access a specific set of services and functionality. For example, a user in the Change User Password group can change passwords for other users.
Hardware Security Module (HSM)	A physical device to provide strong, secure key generation, key storage, and cryptographic processing.
Integration Library	A software library to implement APIs with different language bindings for interacting with Ethereum nodes, such as the JSON-RPC API. For example, [web3j], [web3.js], and [Nethereum].
JSON-RPC API	The Application Programming Interface (API) implemented by public Ethereum to allow ĐApps and Wallets to interact with the platform. The [JSON-RPC] remote procedure call protocol and format is used for its implementation.
Metadata	The set of data that describes and gives information about the payload data in a transaction.
Node	A peer in a peer-to-peer distributed system of computing resources that together form a blockchain system, each of which runs a client.
Off-Chain (Compute) Scaling Mechanism	Processing executed externally to an Ethereum blockchain to facilitate increased transaction speeds. For example, proofs for ZK-SNARKS, which are verified on-chain, or computationally intensive tasks offloaded to one or more Trusted Execution Environments (TEEs).
On-Chain (Layer 2) Scaling Mechanism	Extensions to public Ethereum, such as [Plasma], state channels, and [sharding], to facilitate increased transaction speeds. For more information, see [Ethereum's Layer 2 Scaling Solutions].
On-Chain Privacy Mechanism	Extensions to public Ethereum, such as ZK-SNARKS, or a privacy-preserving TEE compute, enabling private transactions.
Oracle	A service external to either public Ethereum or an Enterprise Ethereum implementation that is trusted by the creators of smart contracts and called to provide information. For example, services to return a current exchange rate or the result of a mathematical calculation.
Participant	A participant is a user of the system interacting via the JSON-RPC API. A participant may be a human, an automated process, or an enterprise participant.
Payload Data	The content of the data field of a transaction, which is usually obfuscated in private transactions. Payload data is separate from the metadata in the transaction.
Performance	The total effectiveness of the system, including overall throughput, individual transaction time, and availability.
Permissioning	The property of a system to ensure operations are executed by and accessible to designated parties.
Precompiled Contract	A smart contract compiled from its source language to EVM bytecode and stored by an Ethereum node for later execution.
Privacy	As defined in ITU [X.800], privacy 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." For the purposes of this Specification, the rights of individuals can be extended to the rights of organizations.
Private State	The data store of Enterprise Ethereum extensions where information regarding private transactions is kept.

Term	Definition
Private Transaction	A transaction where the metadata, payload data or transaction state are readable only by authorized parties.
Private Transaction Manager	A subsystem of an Enterprise Ethereum system for implementing privacy and permissioning.
Public Ethereum	The Ethereum software developed and released by the [Ethereum Foundation].
Role	A set of administrative tasks, each with an associated set of permissions that apply to users or administrators of a system.
Scaling	Increasing the capability of a system, network, or process to handle more work. In terms of Ethereum, this is about increasing transaction speed using on-chain scaling or off-chain scaling mechanisms, or both.
Sidechain	A separate Ethereum blockchain operating on the Ethereum network. A sidechain can be public or private and can also operate on a consortium network.
Smart Contract	A computer program that, in an Ethereum context, is executable on the Ethereum Virtual Machine (EVM). Smart contracts can be written in several higher-level programming languages but must 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.
Smart Contract Language	A programming language and associated tooling used to create smart contracts. For example, [Solidity] and [LLL].
Transaction	A request to execute operations that change state in a blockchain network. Transactions can involve one or more participants.
Trusted Execution Environment	Hardware-based security capabilities to enable a strong foundation for security.
Unspent Transaction Output	Output from a transaction that can be spent as an input for a new transaction.
Wallet	A software application used to store an individual's credentials (cryptographic private keys) which are associated with the state of that user's account on a blockchain.
Zero-knowledge Proof	In cryptography, a 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.

2 Conformance

As well as sections marked as informative, all authoring guidelines, diagrams, examples, and notes in this Specification are informative (that is, non-normative). Everything else in this Specification is normative.

Examples are shown using the following format.

Example

Example text.

Implementors are encouraged to implement requirements in experimental sections. Certificates of Certification may be subject to implementation of experimental sections.

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this Specification are to be interpreted as described in RFC-2119 [RFC2119].

Conforming implementations of this Specification SHOULD be capable of participating as clients in public Ethereum. This requirement is not intended to infer that transactions are required to be shared across or between public Ethereum and consortium networks.

Because this Specification extends the capabilities and interfaces of public Ethereum, there is a dependency between the versions. This version of the Specification is denoted by the EEA to be interface-compatible with the following public Ethereum versions, or updated versions determined and published by the EEA:

- Homestead, launched 14 March 2016
- Metropolis phase 1: Byzantium, 16 October 2017.

Future versions of this Specification are expected to track and align with later public Ethereum versions.

3 Enterprise Ethereum Concepts

This section is informative.

Enterprise Ethereum implementations are extensions to public Ethereum providing enterprise-focused additions, including the capability to perform private transactions, enforce membership (permissioning) and provide transaction throughput scaling. Private transactions are transactions where the metadata or payload data are readable only by parties participating in those transactions.

The following two diagrams show the relationship between Enterprise Ethereum components that can be part of any EEA compliant 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.

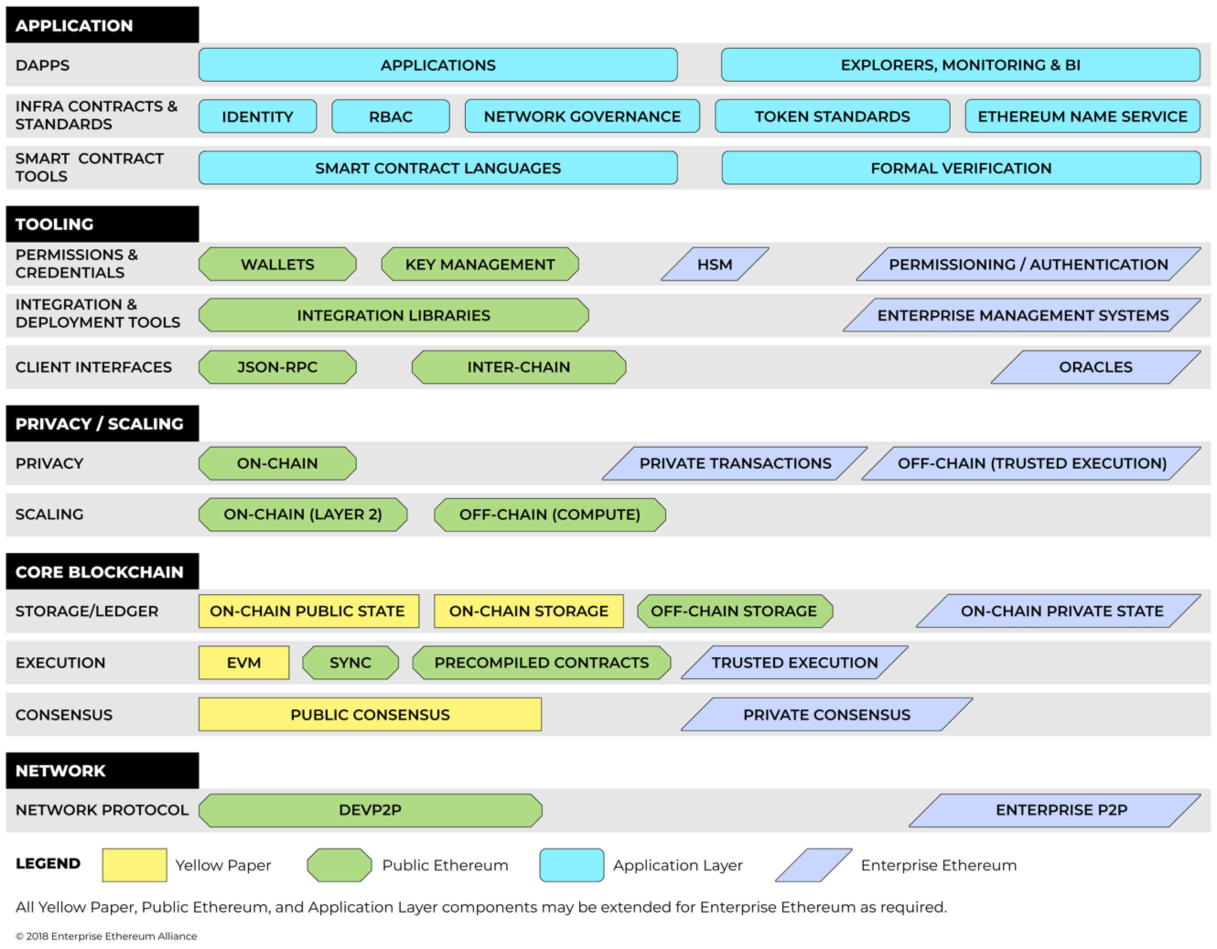

Figure 1 Enterprise Ethereum Architecture Stack

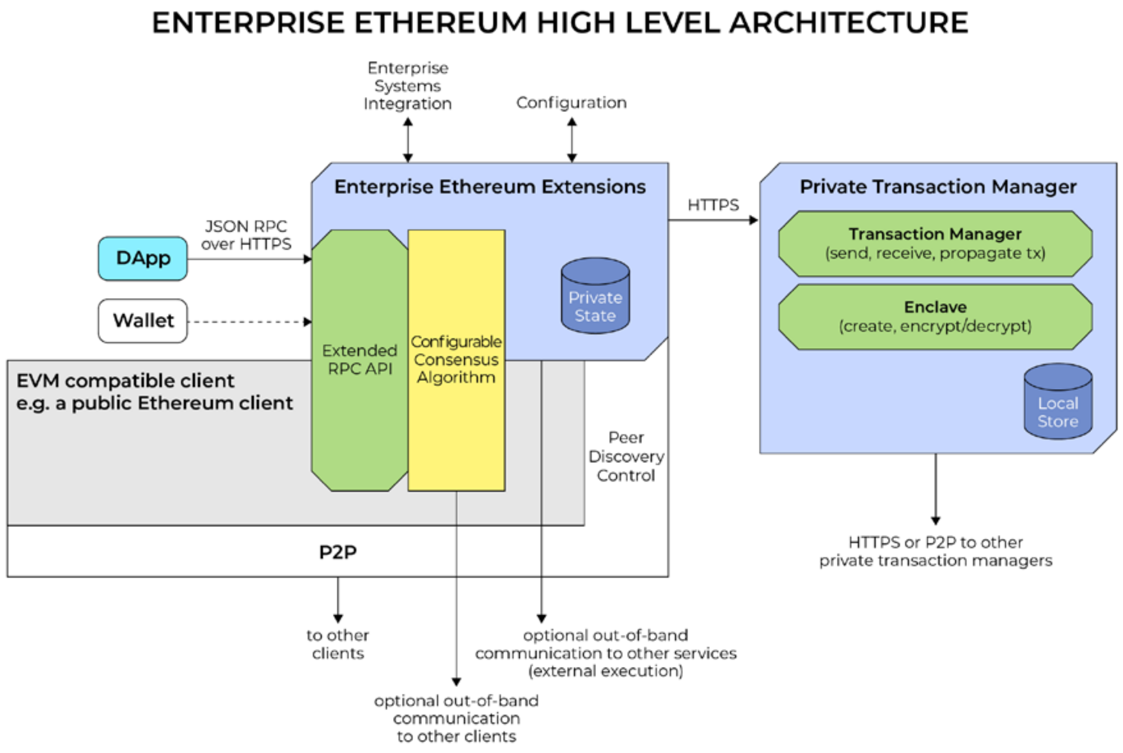
ENTERPRISE ETHEREUM HIGH LEVEL ARCHITECTURE
Enterprise Systems Integration
Configuration
Enterprise Ethereum Extensions
HTTPS
Private Transaction Manager
Transaction Manager (send, receive, propagate tx)
Enclave (create, encrypt/decrypt)
Local Store
JSON RPC over HTTPS
DApp
Wallet
Extended RPC API
Configurable Consensus Algorithm
Private State
EVM compatible client e.g. a public Ethereum client
Peer Discovery Control
P2P
HTTPS or P2P to other private transaction managers
to other clients
optional out-of-band communication to other services (external execution)
optional out-of-band communication to other clients

Figure 2 Representative Enterprise Ethereum High-level Architecture

The architecture stack for Enterprise Ethereum consists of the following five layers:

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

These layers are described in the following sections.

3.1 Network Layer

The Network layer consists of an implementation of the DEVp2p networking protocol. This allows Ethereum nodes to communicate with each other using various protocols running over the DEVp2p connections between the nodes. Enterprise P2P protocols can be used for communications supporting other higher layer functions, such as consensus.

3.2 Core Blockchain Layer

The Core Blockchain layer consists of a mechanism to establish consensus between Ethereum nodes for the acceptance of new blocks. Public consensus algorithms provide a method of doing this when operating with public Ethereum chains. An example of a public consensus algorithm is the Proof of Work (PoW) algorithm, described in the [Ethereum Yellow Paper]. Over time PoW is likely to be phased out from use and replaced with new methods, such as Proof of Stake (PoS).

Enterprise Ethereum implementations provide private consensus algorithms for operations within their private consortium.

Example

Example consensus algorithms include Istanbul [Byzantine Fault Tolerance] (IBFT) [EIP-650], [RAFT], and [Proof of Elapsed Time] (PoET).

The Execution sublayer implements a virtual machine used within an Ethereum node, such as the Ethereum Virtual Machine (EVM) or [Ethereum flavored WebAssembly] (eWASM), its instruction set, and other computational capabilities as required.

Lastly, within the Core Blockchain layer, 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 Unspent Transaction Output (UTXO) model, or at-rest-encrypted key-value stores.

3.3 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.

This Specification does not seek to constrain experimentation to improve the scalability of future implementations of public Ethereum or Enterprise Ethereum. Instead, there is recognition that several forms of scaling improvements will be made to Ethereum nodes over time, the exact form of which cannot be known at this time. Various On-Chain (Layer 2) scaling mechanisms may be implemented, such as [Plasma], [sharding], and easy parallelizability [EIP-648], as well as other Off-Chain (Compute) scaling mechanisms.

Similarly, various On-Chain privacy mechanisms are being explored, such as support for zero-knowledge proofs on public Ethereum.

Enterprise Ethereum implementations are required to provide support for private transactions as described in later sections. Enterprise Ethereum implementations might also provide support for Trusted Execution Environments (TEEs) enabling privacy during code execution.

3.4 Tooling Layer

The Tooling layer critically contains the APIs used to communicate with Ethereum nodes. The primary API is a JSON-RPC API used to submit transactions for execution or deploy smart contracts to maintain arbitrary state. Other API interfaces are allowed, including those intended for inter-blockchain operations and to call external services, such as trusted oracles.

Public Ethereum nodes are often implemented using common integration libraries such as [web3j], [web3.js], or [Nethereum]. Likewise, Enterprise Ethereum implementations are expected to integrate with enterprise management systems using common APIs, libraries, and techniques.

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 purview of an Ethereum node. Enterprise Ethereum implementations enable restricted operations based on user permissions 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. Languages such as [Solidity] and [LLL] are commonly implemented, but support for other languages might be provided without restriction.

3.5 Application Layer

Finally, the Application layer exists, often fully or partially outside of an Ethereum node, whereby 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 can interface with Enterprise Ethereum extensions 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 a public Ethereum client.

4 Application Layer

The Application layer sits at the top of the Enterprise Ethereum stack. This layer contains the components which are built on top of the core Enterprise Ethereum architecture.

4.1 ĐApps Sublayer

Decentralized Applications (ĐApps) run on top of Ethereum.

ĐApps MAY use the Enterprise Ethereum extension to the JSON-RPC API defined in this Specification.

Also at this layer are the blockchain explorers, the tools to monitor the blockchain, and the business intelligence tools.

4.2 Infrastructure Contracts and Standards Sublayer

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

Decentralized identity standards are being developed, for example, by the [Decentralized Identity Foundation].

Role Based Access Control (RBAC) defines methods for authentication and restricting system access to authorized users, potentially realized through smart contracts.

Network Governance methods controlling 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 ENS provides a secure mapping from simple, human-readable names to Ethereum addresses for resources both on and off the blockchain.

4.3 Smart Contract Tools Sublayer

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

Implementations MUST provide deployment and debugging tools for Enterprise Ethereum smart contracts.

Example

Examples of smart contract deployment and debugging tools used in public Ethereum include [Truffle] and [Remix].

Implementations SHOULD extend formal verification methods for use with Enterprise Ethereum smart contracts.

Enterprise Ethereum implementations enable use of these tools and methods through implementation of the Execution sublayer, as described in Section 7.2.

5 Tooling Layer

5.1 Permissions and Credentials Sublayer

Permissioning refers to the ability of an individual node to join the network, and the ability of an individual participant or node to perform specific functions on the Enterprise Ethereum network. For example, only certain nodes can act as validators, while other participants can instantiate smart contracts.

Enterprise Ethereum provides a permissioned implementation of Ethereum that supports transaction privacy. Privacy can be realized at various levels, including peer node connectivity permissioning, participant-level permissioning, controlling which nodes see, relay, and store private transactions, and cryptographically protecting transaction data.

5.1.1 Nodes

Enterprise Ethereum implementations MUST provide the ability to specify at startup a list of static peer nodes to establish peer-to-peer connections with.

Implementations MUST provide the ability to enable or disable peer-to-peer node discovery.

Implementations MUST provide the ability to specify a whitelist of the node identities permitted to join the network.

Implementations MAY provide the ability to specify a blacklist of the node identities not permitted to join the network.

It MUST be possible to specify the node whitelist through an interface or API.

It MUST be possible to specify the node blacklist (if implemented) through an interface or API.

Implementations MUST provide a way to certify the identities of nodes.

Example

White-listing a validating node by making a suitable entry in a dedicated smart contract, or black-listing a node by making a corresponding entry in another dedicated smart contract. An alternative approach could be implementing a cost of gas enabling the private ether to be used as a permissioning token.

An Enterprise Ethereum client SHOULD provide mechanisms to define clusters of nodes at the organizational level, in the context of permissioning.

5.1.2 Participants

Implementations MUST provide the ability to specify a whitelist of participant identities who are permitted to submit transactions.

Implementations MAY provide the ability to specify a blacklist of participant identities who are not permitted to submit transactions.

It MUST be possible to specify the participant whitelist through an interface or API.

It MUST be possible to specify the participant blacklist (if implemented) through an interface or API.

Implementations MUST provide a way to certify the identities of participants.

Implementations MUST provide the ability to specify participant identities in a way aligned with the usual concepts of groups and roles.

5.1.3 Additional Permissioning Requirements

Implementations SHOULD provide permissioning schemes through standard mechanisms, such as smart contracts used in a modular way. That is, permissioning schemes could be implemented to interact with smart contract-based mechanisms.

Implementations SHOULD provide the ability for configuration to be updated at run time without the need to restart.

Implementations MAY provide configuration through the use of flat files, command-line options, or configuration management system interfaces.

Implementations MAY support local key management allowing users to secure their private keys.

Implementations MAY support secure interaction with an external Key Management System for key generation and secure key storage.

Implementations MAY support secure interaction with a Hardware Security Module (HSM) for deployments where higher security levels are needed.

5.2 Integration and Deployment Tools Sublayer

5.2.1 Integration Libraries

Implementations MAY provide integration libraries enabling convenience of interaction through additional language bindings.

Example

Integration libraries might include [web3j], [web3.js], [Nethereum], [protocol buffers], or a REST API.

5.2.2 Enterprise Management Systems

Enterprise-ready capabilities provide the ability to integrate with enterprise management systems using common APIs, libraries, and techniques, as shown in Figure 3.

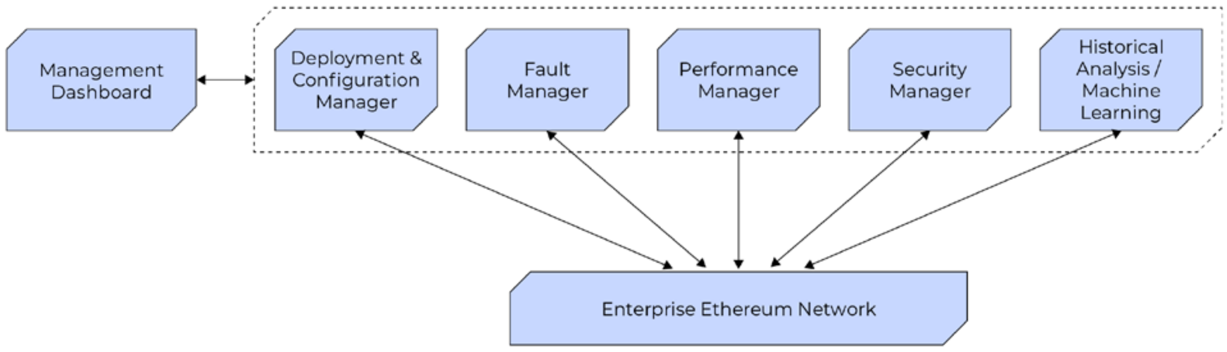
Management Dashboard
Deployment & Configuration Manager
Fault Manager
Performance Manager
Security Manager
Historical Analysis / Machine Learning
Enterprise Ethereum Network

Figure 3 Management Interfaces

Implementations SHOULD provide enterprise-ready software deployment and configuration capabilities, including the ability to easily:

- Deploy through enterprise remote software deployment and configuration systems.
- Modify configurations on already deployed systems.
- Audit configurations on already deployed systems.

Implementations SHOULD provide enterprise-ready software fault reporting capabilities, including the ability to:

- Log software fault conditions.
- Generate events to notify of software fault conditions.
- Accept diagnostic commands from software fault management systems.

Implementations MAY provide enterprise-ready performance management capabilities, including the ability to easily provide relevant performance management metrics for analysis by enterprise performance management systems.

Implementations SHOULD provide enterprise-ready security management interaction capabilities, including the ability for:

- Logs to be easily monitored by enterprise security management systems.
- Events to be easily monitored by enterprise security management systems.
- Secure network traffic to be monitored by enterprise security management systems.

Implementations MAY provide enterprise-ready capabilities to support historical analysis, including the ability for relevant metrics to be easily collected by an enterprise data warehouse system for detailed historical analysis and creating analytical reports.

Implementations MAY include support for other enterprise management systems, as appropriate, such as:

- Common Management Information Protocol (CMIP)
- Web-Based Enterprise Management (WBEM)
- Application Service Management (ASM) instrumentation.

5.3 Client Interfaces Sublayer

5.3.1 Extensions to the JSON-RPC API

This section is experimental.

[JSON] (JavaScript Object Notation) is a lightweight data-interchange format. [JSON] is a language-independent text format that is easy for humans to read and write, and for systems to parse and generate, making it ideal for exchanging data.

[JSON-RPC] 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 JSON-RPC API is used to communicate between ĐApps and Ethereum clients.

Implementations MUST provide support for the public Ethereum JSON-RPC API.

Implementations MUST provide the `eth_sendTransactionAsync` Enterprise Ethereum extension call to the public Ethereum [JSON-RPC API] for at least one of the private transaction types defined in Section 6.1.2. The `eth_sendTransactionAsync` call MUST respond with an HTTP 501 (Not Implemented) status code when an unimplemented private transaction type is requested.

eth_sendTransactionAsync

Creates a transaction, signs it, submits it to the transaction pool, and returns immediately.

Using this function 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 containing:

- `from`: `DATA`, 20 bytes – The address the transaction is sent from.
- `to`: `DATA`, 20 bytes – The address the transaction is sent to.
- `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 function 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 – For private transactions, the public key of the sender.
- `privateFor`: `DATA` – For private transactions, an array of the public keys of the intended recipients.
- `restriction`: `STRING` – Optional. If `restricted`, the transaction is a restricted private transaction. If `unrestricted` the transaction is an unrestricted private transaction. If this parameter is not supplied, the default is `restricted`. For more information, see Section 6.1.2.
- `callbackUrl`: `STRING` – The URL to post the results of the transaction to.

Callback Body

The callback object containing:

- `txHash`: `DATA`, 32 bytes – The transaction hash (if successful).
- `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.

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":"eth_sendTransactionasynch","params":[{
  "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
  "to": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
  "gas": "0x76c0",
  "gasPrice": "0x9184e72a000",
  "value": "0x9184e72a",
  "data":
"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675",
  "privateFrom": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
  "privateTo": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
  "callbackUrl": "http://myserver/id=1",
  "restriction": "restricted"}],
  "id":1}'
```

Response Format

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

Callback Format

```
{
  "txHash": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

5.3.2 Inter-Chain

With the rapid expansion in the number of different blockchains and ledgers, inter-chain mediators are necessary to allow interaction between them. Like other enterprise solutions that include 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.

Enterprise Ethereum implementations MAY provide inter-chain mediation capabilities to enable interaction with different blockchains.

5.3.3 Oracles

In many situations, smart contracts need to interact with real-world information to operate. Oracles securely bridge the data-gap from the smart contract to the real-world information source.

Enterprise Ethereum implementations SHOULD provide the ability to securely interact with oracles to send and receive real-world information.

6 Privacy and Scaling Layer

6.1 Privacy Sublayer

Privacy, in the context of this Specification, refers to the ability to keep data confidential between parties privy to that transaction and to choose which details to provide about a party to one or more other parties.

Enterprise Ethereum implementations are expected to provide some level of transaction privacy. Privacy can be realized at various levels including the peer node connectivity permissioning and user-level permissioning, controlling which nodes see private transactions, and obfuscating transaction data. Options for implementing compliant privacy levels are detailed in Section 6.1.4.

6.1.1 On-Chain

Various on-chain techniques are proposed to improve privacy.

Implementations SHOULD support improved on-chain security techniques as they become available.

Example

Future on-chain security techniques could include techniques such as ZK-SNARKS, range proofs, or ring signatures.

6.1.2 Private Transactions

Many users and operators of Enterprise Ethereum implementations will be 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's revised Payment Services Directive [PSD2] when they provide payment services, and the General Data Protection Regulation [GDPR] when storing personal data regarding individuals. Users of Enterprise Ethereum must signal their intent as to privacy requirements when they send a transaction by utilizing a parameter on the [JSON RPC API] calls. The parameter indicates the preferred transaction type at runtime. This section defines two transaction types to be used for different privacy requirements: *restricted private transactions* and *unrestricted private transactions*.

Transaction information consists of two parts, metadata and payload data. Metadata consists of "envelope" information necessary to execute a transaction. Payload data consists of the transaction contents.

Implementations MUST support private transactions using at least one of the following methods:

- Private transactions where payload data is transmitted to and readable only by the direct participants of a transaction. These transactions are referred to as restricted private transactions.
- Private transactions where payload data is transmitted to all nodes participating in the network but readable only by the direct participants of a transaction. These transactions are referred to as unrestricted private transactions.

When implementing restricted private transactions:

- Implementations MUST support masking or obfuscation of the payload data when stored in restricted private transactions (for example, using cryptographic encryption).
- Implementations MUST support masking or obfuscation of the payload data when in transit in restricted private transactions (for example, using cryptographic encryption).
- Implementations MAY support masking or obfuscation of the metadata when stored in restricted private transactions (for example, using cryptographic encryption).
- Implementations MAY support masking or obfuscation of the metadata when in transit in restricted private transactions (for example, using cryptographic encryption).
- Nodes that relay a restricted private transaction but are not participants in that transaction MUST NOT store transaction payload data.
- Nodes that relay a restricted private transaction but are not participants in that transaction SHOULD NOT store metadata.
- The implementation of the JSON RPC API `eth_sendTransactionAsync` call (if implemented), either without the `restriction` parameter or with the `restriction` parameter set to `restricted`, MUST result in a restricted private transaction.

Example

Private transactions may be implemented by creating private channels, or private smart contracts where the payload data is only stored within the nodes participating in a transaction, and not in any other node (despite that the payload data might be encrypted and only decodable by authorized parties). Private transactions should be transactions to related parties, and unrelated parties should have no access at all to the content of the transaction, the sending party, or the list of participating addresses. In fact, a private smart contract is very similar to a private blockchain network that is only replicated and certified by a subset of participating nodes but is notarized and synchronized through the underlying "public" blockchain. This private blockchain should be able to refer to data in less restrictive private smart contracts, as well as in public smart contracts.

When implementing unrestricted private transactions:

- Implementations SHOULD support masking or obfuscation of the recipient identity when stored in unrestricted private transactions (for example, using cryptographic encryption, or ring signatures and mixing).
- Implementations SHOULD support masking or obfuscation of the sender identity when stored in unrestricted private transactions (for example, using stealth addresses).
- Implementations SHOULD support masking or obfuscation of the payload data when stored in unrestricted private transactions (for example, using cryptographic encryption).
- Implementations MUST support masking or obfuscation of the payload data when in transit in unrestricted private transactions (for example, using cryptographic encryption).

- Implementations MAY support masking or obfuscation of the metadata when stored in unrestricted private transactions (for example, using cryptographic encryption).
- Implementations MAY support masking or obfuscation of the metadata when in transit in unrestricted private transactions (for example, using cryptographic encryption).
- Nodes that relay an unrestricted private transaction but are not participants in that transaction MAY store payload data.
- Nodes that relay an unrestricted private transaction but are not participants in that transaction MAY store transaction metadata.
- The implementation of the JSON RPC API `eth_sendTransactionAsync` call (if implemented) with the `restriction` parameter set to `unrestricted` MUST result in an unrestricted private transaction.

Example

Obfuscated data that is replicated across all nodes can be reconstructed from any node, albeit in encrypted form. Mathematical transactions on numerical data should be able to be validated by the underlying network on a zero-knowledge basis, only to be accessed verbatim by participating parties to the transaction. Specifically, a client should provide 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.

Implementations SHOULD be able to extend the set of participants in a private transaction (or forward the private transaction in some way).

Implementations SHOULD provide the ability for nodes to achieve consensus on their mutually private transactions.

Example

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 Transactions (if implemented)		Unrestricted Private Transactions (if implemented)	
Metadata	**Payload Data**	**Metadata**	**Payload Data**
MAY mask or obfuscate	MUST mask or obfuscate	MAY mask or obfuscate SHOULD mask or obfuscate sender and recipient identity	MUST mask or obfuscate in transit SHOULD mask or obfuscate 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

6.1.3 Off-Chain (Trusted Execution)

TEEs are useful for performing secure, private, efficient, and scalable operations, to provide an additional layer of security, and protecting privacy. When coupled to a blockchain as an off-chain processing environment, TEEs provide the ability for secure, efficient, and scalable transactions, smart contract execution, and privacy of sensitive contact data.

Enterprise Ethereum implementations SHOULD provide the ability for off-chain, trusted execution of transactions and smart contracts.

6.1.4 Privacy Levels

Implementations can support different levels of privacy, as outlined in Table 3, and still comply with this Specification. Because permissioning and privacy are interrelated concepts, the privacy levels specified contain requirements related to both the permissioning and privacy sections of this Specification.

Privacy Level C is the base privacy level for all compliant implementations. To comply with Privacy Level C, implementations have to comply with all MUST and MUST NOT requirements of this Specification. The requirements specifically related to permissioning are the MUST peer node connectivity and user-level permissioning requirements in Sections 5.1.1 and 5.1.2. Implementations have a choice when complying with privacy requirements. To comply with Privacy Level C, implementations are required to comply with all the MUST and MUST NOT requirements in Section 6.1.2 related to either restricted private transactions or unrestricted private transactions.

Supporting specific SHOULD requirements increases the privacy and permissioning abilities for an implementation and are thus recognized as having specific value to users.

Privacy Level B is obtained by providing support for the requirements of Privacy Level C, plus implementing all the SHOULD requirements related to peer node connectivity and user-level permissioning requirements in Sections 5.1.1, 5.1.2, and 5.1.3. Implementations obtaining Privacy Level B demonstrate increased interoperability with the public Ethereum ecosystem and other Enterprise Ethereum implementations.

Privacy Level A is obtained by providing support for Privacy Level B, plus implementing all the SHOULD and SHOULD NOT requirements in Section 6.1.2. Implementations obtaining Privacy Level A demonstrate increased security and privacy protections for their users. Privacy Level A is considered best practice for Enterprise Ethereum implementations and its attainment is highly encouraged.

EEA certification programs will recognize implementations as providing support for Privacy Levels A, B, or C. Certificates of Certification are subject to the unique requirements of EEA-approved vertical business segments.

Table 3 Summary of Privacy Levels

Privacy Level	Description	Definition
A	Best practice privacy and permissioning	Implementations provide support for Privacy Level B and all the SHOULD and SHOULD NOT requirements in Section 6.1.2.
B	Best practice permissioning	Implementations provide support for Privacy Level C and all the SHOULD peer node connectivity and permissioning requirements from Sections 5.1.1, 5.1.2, and 5.1.3.
C	Baseline privacy and permissioning	Implementations provide support for all the MUST peer node connectivity and permissioning requirements from Sections 5.1.1 and 5.1.2 and either: All the MUST and MUST NOT restricted private transaction requirements in Section 6.1.2 OR All the MUST and MUST NOT unrestricted private transaction requirements in Section 6.1.2.

6.2 Scaling Sublayer

Enterprise Ethereum networks 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.

6.2.1 On-Chain (Layer 2)

On-chain scaling at layer 2 improves the capability to handle more transactions but without changing the underlying Ethereum protocol.

Enterprise Ethereum implementations SHOULD provide the ability for improved on-chain processing rates of transactions and smart contracts.

6.2.2 Off-Chain (Compute)

Off-chain scaling moves some of the processing burden from the underlying blockchain network.

Enterprise Ethereum implementations SHOULD provide the ability for off-chain processing of transactions and smart contracts.

6.2.3 Performance

Performance refers to the overall performance of the network, which should not be impaired as usage of the network grows.

EEA certification programs will recognize implementations as providing support for enterprise-appropriate transaction speeds based upon the needs of EEA-approved vertical business segments.

Example

Certificates of Certification may require minimum transaction speeds in terms of [ERC-20] smart contract executions per second, or other measures.

Implementations SHOULD support the ability to have private state data archived from the blockchain while preserving the consistency and validity of the blockchain.

The computing power to validate blocks SHOULD remain constant over time, regardless of the blockchain size or the number of network participants.

The time to access recent blockchain data SHOULD remain constant, regardless of the blockchain size.

Example

By maintaining a parallel repository of recent blocks and transactions not stored as a Merkle trie but optimized for easy reading.

Implementations SHOULD allow network operators to designate a new genesis block to keep the blockchain size from growing perpetually.

Example

Database pruning could be supported, so light and fast applications can be built (understanding that the node might not store the complete blockchain history).

7 Core Blockchain Layer

7.1 Storage and Ledger Sublayer

Enterprise Ethereum implementations SHOULD implement data storage requirements necessary to operate a public Ethereum client.

Implementations MAY implement data storage used for optional off-chain operations. For example, implementations can locally choose to cache the results from a trusted oracle or store information related to systems extensions beyond the scope of this Specification.

Implementations providing support for multiple networks (for example, one or more consortium networks or a public network) MUST store data related to private transactions for those networks in private state dedicated to the relevant network.

A smart contract operating on private state SHOULD be permitted to access private state created by other smart contracts involving the same participants.

A smart contract operating on private state MUST NOT be permitted to access private state created by other smart contracts involving different participants.

Implementations SHOULD provide the ability for private smart contracts to store file objects seamlessly and transparently, so no artificial off-chain file-storage add-ons are needed.

Example

Implementations might choose to provide additional APIs outside this Specification (such as the [WebDAV] protocol) for interaction with file objects.

7.2 Execution Sublayer

Enterprise Ethereum implementations MUST provide a smart contract execution environment implementing the public Ethereum EVM op-code set [EVM Opcodes].

Enterprise Ethereum implementations MAY provide a smart contract execution environment extending the public Ethereum EVM op-code set [EVM Opcodes].

Implementations SHOULD support the ability to synchronize their public state with the public state held by other public Ethereum nodes.

Implementations MAY provide support for the compilation, storage, and execution of precompiled contracts.

TEEs ensure only authorized parties can execute smart contracts on an execution environment related to a given consortium network. Implementations SHOULD provide a TEE.

Multiple encryption techniques could be used to secure TEEs or private state. Implementations SHOULD provide configurable encryption options for use in conjunction with consortium networks.

7.2.1 Settlement Finality

Settlement finality refers to the actions or events required for a transaction to be considered final and irreversible.

When a deterministic consensus algorithm is used, transactions SHOULD be considered final after a defined interval or event. This interval may be a set time period or a set number of blocks being created since the transaction was included in a block.

7.3 Consensus Sublayer

Enterprise Ethereum implementations SHOULD support the ability to form consensus on Ethereum MainNet (public Ethereum) and to form consensus operating as part of an Enterprise Ethereum network.

Implementations MUST be capable of supporting multiple consensus algorithms.

One or more consensus algorithms SHOULD allow operations as part of an Enterprise Ethereum network.

One or more consensus algorithms SHOULD allow operations on the Ethereum MainNet.

One or more consensus algorithms MAY support operations on sidechain networks.

Consensus algorithms MUST be clearly documented for interoperability.

Consensus algorithm implementations SHOULD be modular and configurable.

Example

Some consensus algorithms (for example, [RAFT]) and single-leader validation schemes with multiple validation and block-making nodes simplify consensus processes, favor single-block transaction finality, and enable higher performance.

Consensus algorithms MAY communicate in-band or out-of-band with other clients, as requested. That is, consensus algorithm implementations can make and receive network traffic external to the client-to-client network protocol.

Implementations SHOULD support the Istanbul [Byzantine Fault Tolerance] (IBFT) consensus algorithm [EIP-650], so individual attacked or malfunctioning clients performing voting, block-making, or validation roles do not pose a critical risk to the network.

Implementations MAY support other consensus algorithms.

Implementations MUST provide the ability to specify the consensus algorithms, through configuration, to be used for each public blockchain, private blockchain and sidechain in use.

8 Network Layer

8.1 Network Protocol Sublayer

Network protocols define how nodes communicate with each other.

Nodes MUST be identified and advertised using the Ethereum enode URL format [enode].

Implementations SHOULD use the DEVp2p Wire Protocol [DEVp2p Wire Protocol] for messaging between nodes to establish and maintain a communications channel for use by higher layer protocols. These higher layer protocols are known as capability protocols.

The [Ethereum Wire Protocol] defines the capability protocols for messaging between Ethereum client 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.

Implementations SHOULD support, at a minimum, [Ethereum Wire Protocols] eth/62 and eth/63.

Implementations MAY add new protocols or extend existing Ethereum protocols.

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

Multi-party private contracts and transactions should not require direct connectivity between all parties (because this is very impractical in enterprise settings, especially when many parties are allowed to transact). Common nodes to all parties (for example, voters or blockmakers acting as bootnodes to all parties, and as backup or disaster recovery nodes) should be able to be used 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.

9 Anti-Spam

This section refers to mechanisms for preventing the network 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 5.2.2, or implemented by the Enterprise Ethereum client, as described in the following requirement.

Enterprise Ethereum implementations SHOULD provide effective anti-spam mechanisms so attacking nodes or addresses (either malicious, buggy, or uncontrolled) can be quickly identified and stopped.

Example

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.*

10 Cross-client Compatibility

Cross-client compatibility refers to the ability for a network to operate with different clients.

Enterprise Ethereum clients SHOULD be compatible with the public Ethereum network to the greatest extent possible.

The requirements relating to supporting and extending the public Ethereum opcode set are outlined in Section 7.2.

Implementations MAY extend the public Ethereum APIs. To maintain compatibility, implementations SHOULD ensure these new features are a superset of the public Ethereum APIs.

Example

Extensions to public Ethereum APIs could include Enterprise peer-to-peer APIs, [JSON-RPC APIs] over IPC, HTTP/HTTPS, and websockets.

11 Synchronization and Disaster Recovery

Synchronization and disaster recovery refers to how nodes in a network should behave when connecting for the first time or reconnecting.

Implementations SHOULD support a fast synchronization mode so new clients can be launched quickly and synchronized to long standing, historical blockchains with the understanding that the new client might not have the complete blockchain history.

Implementations SHOULD support a mechanism to back up data and use it later to initialize a node, up to a certain block.

Example

Hard forks might be enabled through this mechanism as well.

Annex A Additional Information

A.1 Acknowledgments

This section is informative.

This Specification was developed with the support of the following editorial team:

- Robert Coote, ConsenSys, robert.coote@consensys.net
- David Hyland-Wood, ConsenSys, david.wood@consensys.net
- Grant Noble, ConsenSys, grant.noble@consensys.net

This Specification is a collaborative effort, so many thanks to the following for their valuable contributions:

- Duarte Aragao, Clearmatics Technologies Limited, da@clearmatics.com
- Sanjay Bakshi, Intel, sanjay.bakshi@intel.com
- Clifton Barber, Enterprise Ethereum Alliance, clif.barber@entethalliance.org
- Jeremey Cousins, Clearmatics Technologies Limited, jc@clearmatics.com
- Robert Dawson, ConsenSys, rob.dawson@consensys.net
- Samer Falah, JP Morgan, samer.falah@jpmorgan.com
- Sara Feenan, Clearmatics Technologies Limited, sf@clearmatics.com
- Lior Glass, BNY Mellon, lior.glass@bnymellon.com
- Kieren James-Lubin, Blockapps Inc, kieren@blockapps.net
- Shahan Khatchadourian, ConsenSys, shahan.khatchadourian@consensus.net
- Tyrone Lobban, JP Morgan, tyrone.lobban@jpmorgan.com
- Martin Michlmayr, Clearmatics Technologies Limited, tbm@clearmatics.com
- George Ornbo, Clearmatics Technologies Limited, go@clearmatics.com
- Ron Resnick, Enterprise Ethereum Alliance, ron.resnick@entethalliance.org
- Peter Robinson, ConsenSys, peter.robinson@consensys.net
- Suresh Shetty, JP Morgan, suresh.shetty@jpmorgan.com
- Conor Svensson, blk.io, conor@blk.io
- Tom Willis, Intel, tom.willis@intel.com
- John Whelan, Santander Digital, john@sanlab.io

The editors would also like to thank the members of the EEA Release 1 Development Ad Hoc:

- Amber Baldet, JP Morgan, amber.baldet@jpmorgan.com
- Peter Broadhurst, ConsenSys, peter.broadhurst@consensys.net
- Shawn Douglass, Amberdata, sdouglas@amberdata.io
- Tom Golway, Hewlett Packard Enterprise, thomas.golway@hpe.com
- Yu-Te Lin, AMIS, yute@am.is
- Alex Liu, AMIS, alex@maicoin.com
- Tom Lombardi, Enterprise Ethereum Alliance, tom.lombardi@entethalliance.org
- Andrew Miller, IC3, socrates1024@gmail.com
- Patrick Nielsen, JP Morgan, patrick.m.nielsen@jpmorgan.com
- Peter Rutgers, ING Bank N.V., peter.rutgers@ing.com
- Joshua Satten, Wipro, johsua.satten@wipro.com
- Przemek Siemion, Santander Digital, przemek@sanlab.io

- Krishnaprasad Shastry, Hewlett Packard Enterprise, krishnaprasad.shastry@hpe.com
- Amanda Stanhaus, JP Morgan, amanda.c.stanhaus@jpmorgan.com
- Cale Teeter, Microsoft, cale.teeter@microsoft.com
- Jim Zhang, ConsenSys, jim.zhang@consensys.net.

A.2 References

A.2.1 Normative References

- [DEVp2p Wire Protocol] URL: https://github.com/ethereum/wiki/wiki/ÐΞVp2p-Wire-Protocol
- [EIP-648] Easy Parallelizability. URL: https://github.com/ethereum/EIPs/issues/648
- [EIP-650] Istanbul Byzantine Fault Tolerance. URL: https://github.com/ethereum/EIPs/issues/650
- [enode] enode URL Format. URL: https://github.com/ethereum/wiki/wiki/enode-url-format
- [Ethereum] URL: https://www.ethereum.org/
- [Ethereum flavored WebAssembly] URL: https://github.com/ewasm/design
- [Ethereum Wire Protocol] URL: https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol
- [EVM Opcodes] URL: https://github.com/trailofbits/evm-opcodes
- [Proof of Elapsed Time] Consensus algorithm. URL: https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html
- [RFC2119] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Internet RFC 2119. URL: http://www.ietf.org/rfc/rfc2119.txt
- [System Requirements] John Whelan, R1AH-18-00002R000-EEASystemRequirements-180226 URL: https://member.entethalliance.org/higherlogic/ws/groups/a4444d30-5f36-4b09-af77-88da97ee1b64/documents/201875/document?document_id=140

A.2.2 Informative References

- [Byzantine Fault Tolerance]. URL: https://en.wikipedia.org/wiki/Byzantine_fault_tolerance
- [Decentralized Identity Foundation]. URL: http://identity.foundation/
- [ERC-20] URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
- [ERC-223] URL: https://github.com/ethereum/EIPs/issues/223
- [ERC-621] URL: https://github.com/ethereum/EIPs/pull/621
- [ERC-721] URL: https://github.com/ethereum/eips/issues/721
- [ERC-827] URL: https://github.com/ethereum/EIPs/issues/827
- [Ethereum Foundation] URL: https://www.ethereum.org/foundation
- [Ethereum's Layer 2 Scaling Solutions] URL: https://medium.com/l4-media/making-sense-of-ethereums-layer-2-scaling-solutions-state-channels-plasma-and-truebit-22cb40dcc2f4
- [Ethereum Yellow Paper] URL: https://ethereum.github.io/yellowpaper/paper.pdf
- [GDPR] European Union General Data Protection Regulation. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A32016R0679
- [JSON] JavaScript Object Notation. URL: http://www.json.org
- [JSON-RPC] JavaScript Object Notation - Remote Procedure Call. URL: http://www.jsonrpc.org/specification
- [JSON-RPC API] URL: https://github.com/ethereum/wiki/wiki/JSON-RPC
- [LLL] URL: http://lll-docs.readthedocs.io/en/latest/lll_introduction.html
- [Nethereum] URL: https://nethereum.com/
- [Plasma] URL: https://plasma.io

- [Protocol Buffers] URL: https://developers.google.com/protocol-buffers/
- [PSD2] European Union Payment services (PSD 2) - Directive (EU) 2015/2366. URL: https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en
- [RAFT] Consensus algorithm. URL: https://github.com/jpmorganchase/quorum/blob/master/raft/doc.md
- [Remix] URL: https://github.com/ethereum/remix
- [Solidity] URL: https://solidity.readthedocs.io
- [Sharding] URL: https://github.com/ethereum/wiki/wiki/Sharding-FAQ
- [Truffle] URL: https://github.com/trufflesuite/truffle
- [web3.js] URL: https://github.com/ethereum/web3.js
- [web3j] URL: https://web3j.io/
- [WebDAV] Web Document Authoring and Versioning. URL: https://tools.ietf.org/html/rfc4918
- [X.800] Security architecture for Open Systems Interconnection for CCITT applications. URL: http://www.itu.int/rec/T-REC-X.800-199103-I/en