Enterprise Ethereum Alliance - Enterprise Ethereum Client Specification V2

15 October 2018

Editors:

Daniel Burnett (ConsenSys)
Robert Coote (ConsenSys)
Chaals Nevile (EEA)
Grant Noble (ConsenSys)

Contributors:

The TEE task force within the Enterprise Ethereum Alliance Technical Specification Working Group, and Ashfaq Ahmed (Broadridge), Duarte Aragao (Clearmatics Technologies Limited), Meredith Baxter (ConsenSys), Sanjay Bakshi (Intel), Clifton Barber (Enterprise Ethereum Alliance), Thomas Bertani (Oraclize), Benjamin C Burns (ConsenSys), Jean-Charles Cabelguen (iEx.ec), Alice Corsini (Oraclize), Jeremy Cousins (Clearmatics Technologies Limited), Robert Dawson (ConsenSys), Robert Drost (ConsenSys), Samer Falah (JP Morgan), Sara Feenan (Clearmatics Technologies Limited), Lior Glass (BNY Mellon), Bill Gleim (ConsenSys), Mark Grand (HCL), Dan Guido (Trail of Bits), Timothy Holland (Blockapps Inc), David Hyland-Wood (ConsenSys), Kieren James-Lubin (Blockapps Inc), Shahan Khatchadourian (ConsenSys), John S Lee (Bunz), Cen Liu (Blockapps Inc), Tyrone Lobban (JP Morgan), Martin Michlmayr (Clearmatics Technologies Limited), Mike Myers (Trail of Bits), Immad Naseer (Microsoft), Alex Oberhauser (Cambridge Blockchain), George Ornbo (Clearmatics Technologies Limited), Fernando Pari (ioBuilders), George Polzer (Everyman's AI), Dhyan Raj (Synechron), Ron Resnick (Enterprise Ethereum Alliance), Peter Robinson (ConsenSys), Dan Selman (Clause.io), Przemek Siemion (Banco Santander), Conor Svensson (blk.io), Antoine Toulme (ConsenSys), Ben Towne (SAE ITC), John Whelan (Banco Santander), Tom Willis (Intel), Victor Wong (Blockapps), Jim Zhang (ConsenSys).

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.

Legal Notice

Status of This Document

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

The changes made since version 1 of the Specification, published on 2 May 2018, have been reviewed by the Enterprise Ethereum Alliance (EEA) Technical Specification Working Group (TSWG). The TSWG agreed on 2018-09-27 to request that the EEA Board approve this draft to be published as an EEA Standard obsoleting the Enterprise Ethereum Client Specification version 1.

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 2019.

The TSWG *expects* at time of writing to produce a new revision of this specification for release in the second quarter of 2019 which would obsolete 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 permission management.
- Requirements and techniques to understand and manage network performance.
- 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

1. Introduction §

This section is non-normative.

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

Enterprise Ethereum is based on technologies and concepts of public Ethereum, only extending that as necessary to support the needs of enterprise deployments. The extensions to public Ethereum are designed to satisfy the performance, permissioning, and privacy demands of enterprise deployments, informally known as the “three Ps” of Enterprise Ethereum.

1.1 Why Produce a Client Specification? §

With a growing number of vendors developing Ethereum clients, meeting the requirements for Enterprise Ethereum ensures different clients can communicate with each other and can all work reliably on an Enterprise Ethereum network.

For Ðapp developers, for example, a Client Specification ensures clients provide a set of identical interfaces, so they can be sure their app 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 enterprise blockchain use.

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 enterprise 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 network.

2. Conformance §

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

The key words *MAY*, *MUST*, *MUST NOT*, *SHOULD*, and *SHOULD NOT* are to be interpreted as described in [RFC2119].

This Specification extends the capabilities and interfaces of public Ethereum.

[P] XCLI-005: Features of public Ethereum, if implemented, *MUST* be compatible with the Metropolis phase 1: Byzantium, 16 October 2017 release of Ethereum.

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

2.1 Experimental Requirements §

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

2.2 Requirement Categorization §

All requirements in this Specification are categorized, as described in the table below.

Table 1 Requirement Categorization

Category	**Prefix**	**Description**
Protocol	[P]	The desired properties and correctness of the system will be jeopardized if all clients do not follow this requirement.
Client	[C]	The desired properties and correctness of the system will not be jeopardized if all clients do not follow this requirement. Client requirements are usually those requirements that do not impact global system behavior.
External	[E]	Apply to components other than the EEA client. External requirements are usually further sub-classified, as shown below.
External: Development	[E:D]	An external requirement related to tooling and development of smart contracts.
External: Operations	[E:O]	An external requirement related to operations, including monitoring and infrastructure management.

3. Security Considerations §

This section is non-normative.

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

However some aspects of Ethereum in general, and Enterprise Ethereum in particular, are especially important in the enterprise 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.

3.1 Attacks on Ethereum Clients §

Modeling attacks against an Enterprise Ethereum client 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" 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.

3.2 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.

Ethereum features such as Gas mitigate the risk of resource-consumption attacks by rogue network participants. 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 what risks are of most concern to a client implementation versus which risks are better mitigated by updates to the Ethereum consensus protocol design.

3.3 Handling of Sensitive Data §

The implications of private data stored on the network 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 the Ethereum blockchain) 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, an Ethereum client is also responsible for managing and protecting private keys. Encrypting private keys with a pass phrase 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.

3.4 Security of Client Implementations §

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

- P2P protocol implementation
- Object deserialization routines
- Ethereum Virtual Machine (EVM) implementation
- Key pair generation.
-

The P2P protocol used for communication among nodes in the peer-to-peer Ethereum network 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 (de)serialization is commonly part of the underlying implementation of the P2P protocol, but also a source for complexity that has historically been 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 Ethereum clients. Unless EVMs behave identically for all possibilities of input, there is a serious risk of a hard fork event 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 an Ethereum 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 during the keypair generation step. An insecure pseudo-random number generator is not generally apparent by merely observing its outputs, but enables attackers to break the encryption and reveal users' sensitive data.

4. Enterprise Ethereum Concepts §

This section is non-normative.

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.

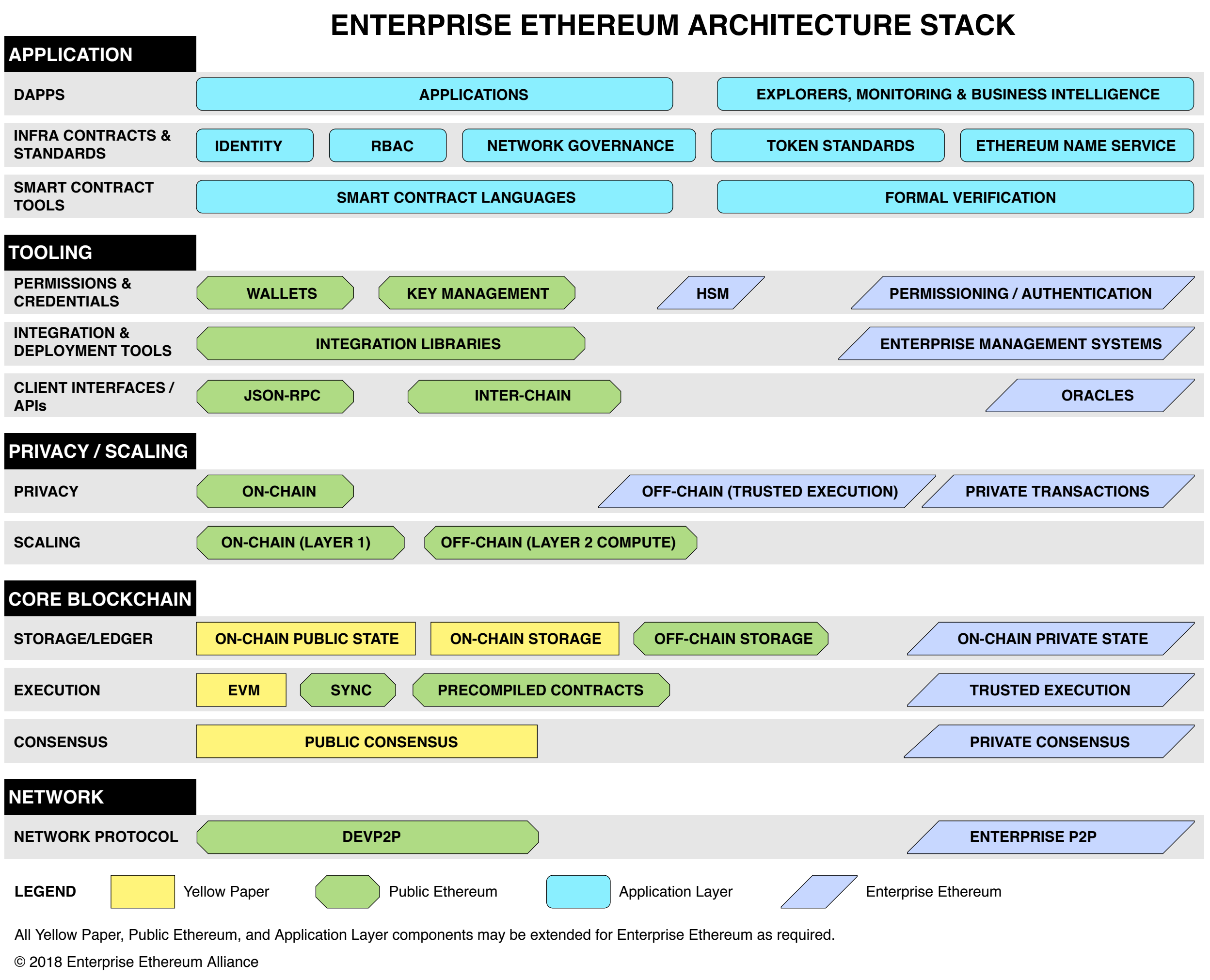

Figure 1. Enterprise Ethereum Architecture Stack

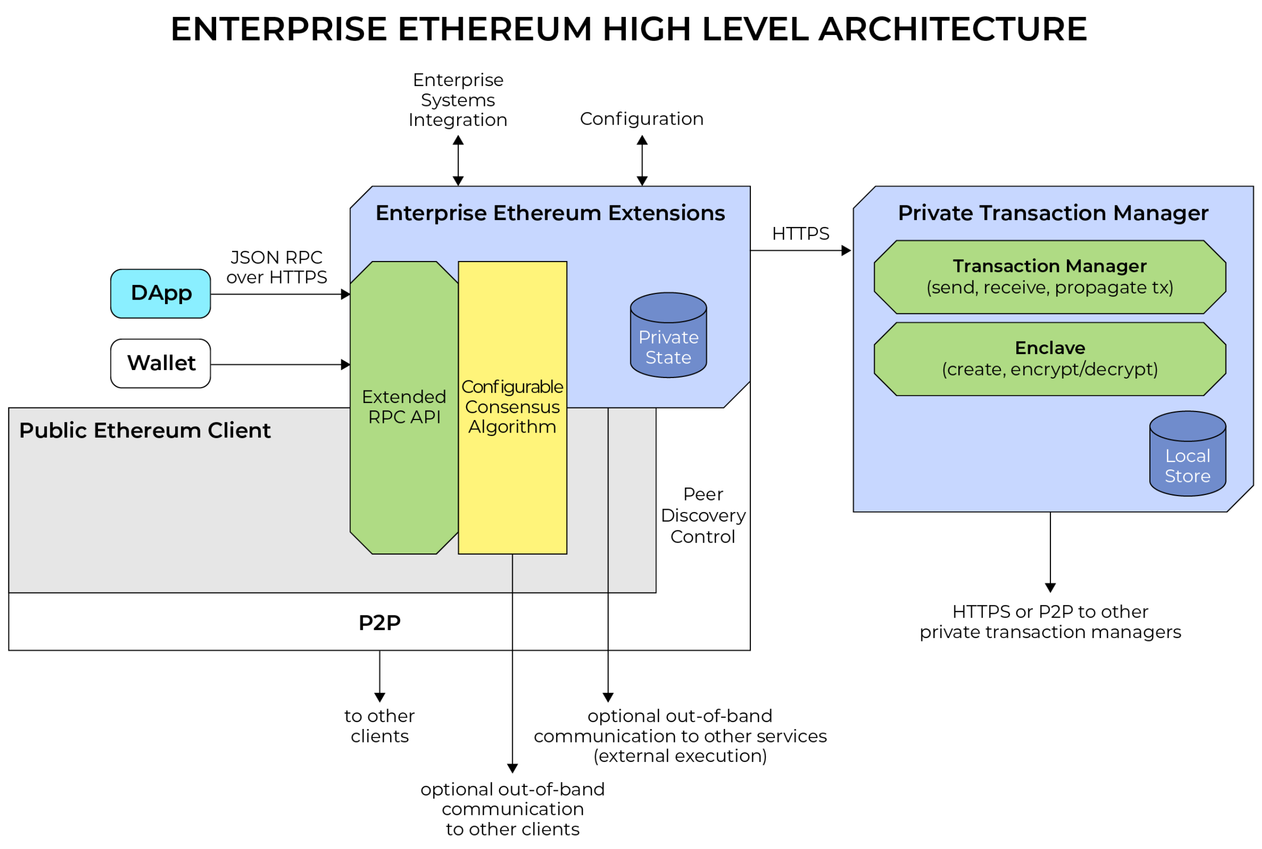
ENTERPRISE ETHEREUM HIGH LEVEL ARCHITECTURE
Enterprise Systems Integration
Configuration
Enterprise Ethereum Extensions
HTTPS
Private Transaction Manager
Transaction Manager (send, receive, propagate tx)
Enclave (create, encrypt/decrypt)
Local Store
JSON RPC over HTTPS
DApp
Wallet
Extended RPC API
Configurable Consensus Algorithm
Private State
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.

4.1 Network Layer §

The Network layer consists of an implementation of a peer-to-peer (P2P) networking protocol allowing Ethereum nodes to communicate with each other using, for example, the DEVp2p protocol. Additional enterprise P2P protocols will be formalized over time to provide the communications needed to supplement higher levels of the stack.

4.2 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 network.

EXAMPLE 1: Consensus Algorithms

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 a client, 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.

4.3 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 clients over time, the exact form of which cannot be known at this time.

Scaling solutions are broadly categorized into layer 1 and layer 2 solutions.

Layer 1 scaling solutions are implemented at the base level protocol layer. On-Chain (Layer 1) scaling might be implemented using techniques such as [sharding] and easy parallelizability [EIP-648].

Layer 2 scaling solutions do not require changes to the base level protocol layer and are implemented at the application protocol layer using smart contracts. Off-Chain (Layer 2) scaling mechanisms might be implemented using techniques such as [Plasma] and [state-channels] 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 can also provide support for off-chain Trusted Computing, enabling privacy during code execution.

4.4 Tooling Layer §

The Tooling layer contains the APIs used to communicate with clients. The primary API is a JSON-RPC API used to submit transactions for execution or to deploy smart contracts to maintain arbitrary state. Other APIs 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 scope of a client. 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.

4.5 Application Layer §

Finally, 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 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.

5. Application Layer §

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

5.1 ĐApps Sublayer §

Decentralized applications (ĐApps) run on top of Ethereum.

[C] DAPP-010: ĐApps *MAY* use the Enterprise Ethereum extensions 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.

5.2 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.

> EXAMPLE 2: Decentralized Identity Standards
>
> Decentralized identity standards are being developed by the Decentralized Identity Foundation [DIF].

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

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

5.3 Smart Contract Tools Sublayer §

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

[E:D] SMRT-010: Implementations *MUST* provide deployment and debugging tools for Enterprise Ethereum smart contracts.

> EXAMPLE 3: Smart Contract Deployment and Debugging Tools
>
> Examples of smart contract deployment and debugging tools used in public Ethereum include [Truffle] and [Remix].

[E:D] SMRT-020: 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 8.2 Execution Sublayer.

6. Tooling Layer §

6.1 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 supporting 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.

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

[P] NODE-030: Implementations *MUST* provide the ability to specify a whitelist of the node identities permitted to join the network.

[P] NODE-040: Implementations *MAY* provide the ability to specify a blacklist of the node identities not permitted to join the network.

[P] NODE-050: It *MUST* be possible to specify the node whitelist through an interface. For example, through a transaction into a smart contract, or through an API.

[P] NODE-060: It *MUST* be possible to specify the node blacklist (if implemented) through an interface. For example, through a transaction into a smart contract, or through an API.

[P] NODE-070: Implementations *MUST* provide a way to certify the identities of nodes.

> EXAMPLE 4: Certifying Node Identities
>
> Whitelisting a validating node by making a suitable entry in a dedicated smart contract, or blacklisting 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.

[P] NODE-080: An Enterprise Ethereum client *SHOULD* provide mechanisms to define clusters of nodes at the organizational level, in the context of permissioning.

6.1.2 Participants §

[P] PART-010: Implementations *MUST* provide the ability to specify a whitelist of participant identities who are permitted to submit transactions.

[P] PART-020: Implementations *MAY* provide the ability to specify a blacklist of participant identities who are not permitted to submit transactions.

[P] PART-030: It *MUST* be possible to specify the participant whitelist through an interface. For example, through a transaction into a smart contract, or through an API.

[P] PART-040: It *MUST* be possible to specify the participant blacklist (if implemented) through an interface. For example, through a transaction into a smart contract, or through an API.

[P] PART-050: Implementations *MUST* provide a mechanism to connect to an enterprise identity management system in order to certify the identities of participants.

[P] PART-055 Implementations *MUST* support anonymous participants.

[P] PART-060: Implementations *MUST* provide the ability to specify participant identities in a way aligned with the usual concepts of groups and roles.

[P] PART-070: Implementations *SHOULD* be able to authorize the types of transactions a participant can submit, providing separate permissioning for the ability to:

- Deploy a smart contract.
- Call a function that changes the state of a smart contract.
- Perform a simple value transfer.

6.1.3 Ethereum Accounts §

[P] ACCT-010: Implementations *SHOULD* provide the ability to specify a whitelist of the Ethereum accounts permitted to be used on the blockchain.

[P] ACCT-020: It *MUST* be possible to specify the Ethereum account whitelist (if implemented) through an interface. For example, through a transaction into a smart contract, or through an API.

6.1.4 Additional Permissioning Requirements §

[P] PERM-010: 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.

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

[C] PERM-030: Implementations *MAY* provide configuration through the use of flat files, command-line options, or configuration management system interfaces.

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

[C] PERM-050: Implementations *MAY* support secure interaction with an external Key Management System for key generation and secure key storage.

[C] PERM-060: Implementations *MAY* support secure interaction with a Hardware Security Module (HSM) for deployments where higher security levels are needed.

6.2 Integration and Deployment Tools Sublayer §

6.2.1 Integration Libraries §

[E:D] ILIB-010: Implementations *MAY* provide integration libraries enabling convenience of interaction through additional language bindings.

> EXAMPLE 5: Integration Libraries
>
> Integration libraries might include [web3j], [web3.js], [Nethereum], [protocol-buffers], or a REST API.

6.2.2 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.

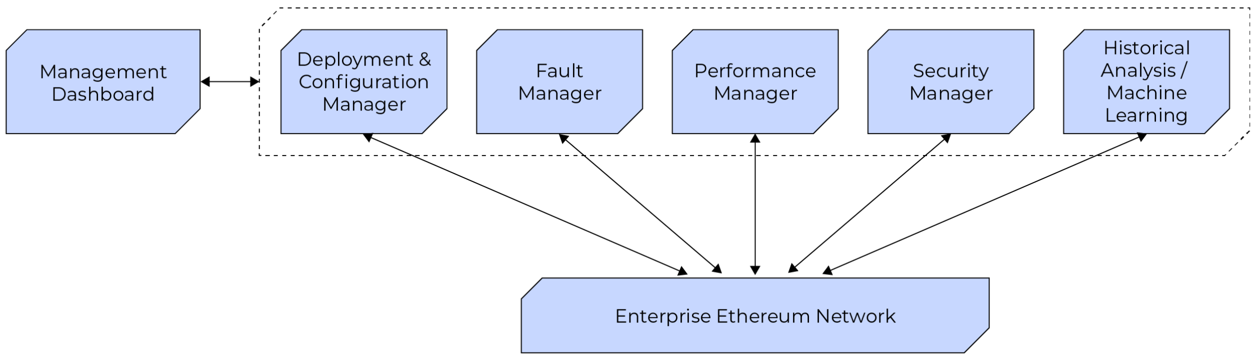
Management Dashboard
Deployment & Configuration Manager
Fault Manager
Performance Manager
Security Manager
Historical Analysis / Machine Learning
Enterprise Ethereum Network

Figure 3 Management Interfaces

[E:O] ENTM-010: 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.

[E:O] ENTM-020: 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.

[E:O] ENTM-030: 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.

[E:O] ENTM-040: Implementations *SHOULD* provide enterprise-ready security management interaction capabilities, including the ability for logs, events, and secure network traffic to be monitored by enterprise security management systems.

[E:O] ENTM-050: 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.

[E:O] ENTM-060: 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.

6.3 Client Interfaces Sublayer §

6.3.1 JSON-RPC §

This section is non-normative.

[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 ÐApps and Ethereum clients.

6.3.2 Compatibility with the Core Ethereum JSON-RPC API §

[P] JRPC-010: Implementations *MUST* provide support for the following methods of the public 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_sign`
- `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: [JSON-RPC-API] methods *SHOULD* be implemented to be backward compatible with the definitions given in revision 328, 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.

[P] JRPC-011: Clients *MAY* provide implementations of other methods.

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

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

[C] JRPC-050: Clients *MUST* provide an implementation of the [JSON-RPC-PUB-SUB] API.

[C] JRPC-060: Clients *MAY* implement additional subscription types for the [JSON-RPC-PUB-SUB] API.

[P] JRPC-070: 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_`.

6.3.3 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: Implementations *MUST* provide the `eea_sendTransactionAsync` and `eea_sendTransaction` Enterprise Ethereum extension methods for at least one of the private transaction types defined in Section 7.1.3 Private Transactions.

[P] JRPC-030: The `eea_sendTransactionAsync` and `eea_sendTransaction` methods *MUST* respond with an HTTP 501 (Not Implemented) status code when an unimplemented private transaction type is requested.

6.3.3.1 eea_sendTransactionAsync §

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

Using this 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 for this call contains:

- `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 – 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 7.1.3 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 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 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"
}
```

6.3.3.2 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 the transaction is sent from.
- `to` DATA, 20 bytes – Optional when creating new contract. 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 – 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 7.1.3 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
}
```

6.3.3.3 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

This call 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"]} }
```

6.3.4 Network Permissioning Using Smart Contracts §

This section is experimental.

This section presents a collection of smart contract interfaces to achieve network permissioning.

This permissioning model consists of networks, participant groups, and participants. A *network* is a collection of enterprises wishing to interact using an Enterprise Ethereum blockchain. Each enterprise is represented as a *participant group* with an accompanying list of client nodes belonging to that enterprise.

A participant group can have client nodes added or removed from it's node list. After a client node is added to a participant group's node list, that client node is allowed to join and become part of the Enterprise Ethereum blockchain network.

Node permissioning on the network is therefore achieved by deciding which participant groups can join and which must leave the network. If a participant group joins the network, the nodes accompanying that participant group are permitted to join. Conversely, if a participant group must leave the network, the nodes accompanying that participant group are disconnected from the network.

Participant groups are a collection of *participants*. Each participant has individually specified permissions. The permissions reflect the different ways the participant can act on behalf of the participant group. A participant corresponds to a single person or agent allowed to administer the network. A participant is represented by one or more Ethereum addresses.

Participants originate changes (mutations), such as adding a new participant to a participant group, adding a new client node to a participant group's node list, or inviting other participant groups to join the network.

A decider function is used to decide, based on the number of invitations for a specific participant group, whether or not the participant group is permitted to join the network. The decider also determines when to evict a member and especially when the decider should be changed, so that the network logic can change.

Node whitelisting is achieved in this model by participant groups (that is, enterprises) determining which client nodes are added to the client node list for their participant group. Node blacklisting can be achieved using the methods described in Section 6.3.4.5 Node Blacklisting.

In this permissioning model there are four fundamental smart contract interfaces:

- `Participant`
- `ParticipantGroup`
- `Network`
- `PermissioningDecider`.

[P] PERM-070: Implementations *MUST* provide the `Participant`, `ParticipantGroup`, `Network`, and `PermissioningDecider` Enterprise Ethereum smart contract interfaces, as described in the following sections.

6.3.4.1 Participant §

The `Participant` smart contract contains participants 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 `Participant`.

```
pragma solidity ^0.4.24;

interface Participant {
// Metadata

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

// Retrieve the participant identifier.
function getId() external view returns (string);

// Authorization mutation.

// Add an Ethereum account address to a participant. Multiple addresses
// can be added.
function addAddress(address _owner) external;

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

// Authorization queries.

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

// Network of trust (reputation) mutators.

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

// Set the participant as vouching for (endorsing) a (child) participant.
function endorse(Participant _child) external;

// Set the participant as no longer vouching for (not endorsing) a (child
// participant.
function unendorse(Participant _child) external;

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

// Set the (parent) participant as vouching for the participant.
function recordEndorsement(Participant _parent) external;

// Set the (parent) participant to no longer vouch for the participant.
function eraseEndorsement(Participant _parent) external;}
```

Each `Participant` can have multiple Ethereum addresses to guard against key loss. A graph of endorsements is present to establish trust.

How authentication happens is up to the users, typically depending on the relationship between the endorser and the endorsee. Some example authentication mechanisms could be:

- alice@a.net sends an email to bob@a.net, asking to confirm Bob's participant address.
- alice@a.net sends an email to bob@b.com (note the different domain address), asking Bob to join a video call to assert his ownership of the Ethereum account.
- Alice walks over to Bob's desk and asks what his participant address is.

> NOTE
>
> Any caller can add any address as a parent of a participant. To authenticate a participant, parent links must be followed and the corresponding child link must be present.

6.3.4.2 ParticipantGroup §

The `ParticipantGroup` smart contract represents a group of participants and their permissions.

```
pragma solidity ^0.4.24;

import "./Participant.sol";
interface ParticipantGroup {
// Metadata

// Retrieve the participant group name.
function getName() external view returns (string);

// Member queries

// Retrieve the permissions for the participant.
function permission(Participant) external view returns (uint);

// Member enumeration

// Retrieve the number of participants in the participant group.
function memberCount() external view returns (uint);

// Retrieve a participant, specified by index, from the participant group
function getMember(uint idx) external view returns (Participant);

// Membership mutation

// Add a participant to the participant group. Requester must have
// `CAN_ADD_PARTICIPANT` permission.
function addParticipant(Participant requester, Participant object,
                               uint _permission) external;

// Remove a participant from the participant group. Requester must have
// `CAN_REMOVE_PARTICIPANT` permission.
function removeParticipant(Participant requester,
                               Participant object) external;

// Events

// Emitted when a participant is added to a participant group.
event MemberAdded(Participant _participant, uint _permission);

// Emitted when a participant is removed from a participant group.
event MemberRemoved(Participant _participant, uint _permission);}
```

The `addParticipant` function grants permissions as follows. The requester chooses which permissions to grant to an object, but cannot grant permissions it does not have itself. Effectively, this means the actual permissions are the bitwise AND of the requester's permissions and the _permission parameter. The permissions reflect the different ways participants in a participant group can act on behalf of the the participant group.

The following is an example permissions list smart contract.

```
pragma solidity ^0.4.24;

contract Permissions {
// Abilities to change the membership of a participant group.
uint constant public CAN_ADD_PARTICIPANT = 0x1;
uint constant public CAN_REMOVE_PARTICIPANT = 0x2;

// Abilities to change the node inventory of a participant group.
uint constant public CAN_ADD_NODE = 0x4;
uint constant public CAN_REMOVE_NODE = 0x8;


// Abilities to vote on behalf of a participant group for other
// participant groups to join the network.
uint constant public CAN_INVITE_PARTICIPANTGROUP = 0x10;
uint constant public CAN_UNINVITE_PARTICIPANTGROUP = 0x20;

// Abilities to vote on behalf of a participant group for a new rule
// engine.
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;
}}
```

6.3.4.3 Network §

As described above, networks are a collection of participant groups. Each participant group supplies a list of client nodes as [enode] URLs, with the intent that the Network smart contract is used as a permissions model for client nodes (servers) allowed to connect to the Enterprise Ethereum blockchain. If a participant group belongs to a network, any participant with the CAN_ADD_NODE permission can add to the client node list for that participant group. Each participant group has autonomy over the nodes it has running in the network.

To add a participant group to the network, every participant group that is already a member of the network can vote to invite or uninvite the participant group. These two functions record the desire of the participant group, but it is up to the permissioning decider to choose when to update the network roster.

Participant groups can have `WRITE` or `READ` permissions in the context of a network, which dictates the permissions of the client nodes belonging to that participant group. Transactions to the actual blockchain network are only to be accepted by client nodes belonging to participant groups with `WRITE` permission in the `Network` smart contract representing it.

```
pragma solidity ^0.4.24;

import "./ParticipantGroup.sol";
import "./Participant.sol";
import "./PermissioningDecider.sol";
interface Network {
// Node queries.

// Retrieve the participant group the node is part of.
function participantGroupOf(string _node) external view
    returns (ParticipantGroup);

// Retrieve the number of nodes in the participant group.
function participantGroupsNodeCount(ParticipantGroup) external view
    returns (uint);

// Retrieve a node, specified by index, from the participant group.
function participantGroupsNode(ParticipantGroup, uint idx) external view
    returns (string);

// Authorization queries.

// Retrieve the permissions for the participant group.
function permission(ParticipantGroup) external view returns (uint);

// Check if the node has `READ` permission in the context of the network.
function checkRead(string _node) external view returns (bool);

// Check if the node has `WRITE` permission in the context of the network
function checkWrite(string _node) external view returns (bool);

// Group specific administration.

// Add a node to a participant group. Participant must have `CAN_ADD_NODE
// permission.
function addNode(ParticipantGroup, Participant, string _node) external;

// Remove a node from a participant group. Participant must have
// `CAN_REMOVE_NODE` permission.
function removeNode(ParticipantGroup, Participant, string _node) external

// Group membership queries.

// Retrieve the number of participant groups.
function participantGroupCount() external view returns (uint);

// Retrieve a participant group, specified by index, from the participant
// groups.
function getParticipantGroup(uint idx) external view
    returns (ParticipantGroup);

// Group membership vote counts.

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

// Retrieve the number of invites for the participant group to have `WRIT
// permissions in the context of the network.
function writeInvitesReceived(ParticipantGroup) external view
    returns (uint);

// Retrieve the number of uninvites for the participant group to leave th
// network.
function uninvitesReceived(ParticipantGroup) external view returns (uint)

// Group membership mutators.

// Invite a participant group to join the network. Participant must have
// `CAN_INVITE_PARTICIPANTGROUP` permission.
function invite(ParticipantGroup _invitee, ParticipantGroup _ginviter,
                      Participant _uinviter, string _node, uint _perm) ex

// Uninvite a participant group from the network. Participant must have
// `CAN_UNINVITE_PARTICIPANTGROUP` permission.
function uninvite(ParticipantGroup _invitee, ParticipantGroup _ginviter,
                      Participant _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 participant group.
// Useful for admin weighting.
function nominatedDecider(ParticipantGroup) external view
    returns (PermissioningDecider);
```

```
// Rule engine mutator.

// Propose a new permissioning decider.
function proposeDecider(PermissioningDecider _next,
                        ParticipantGroup _gproposer, Participant _upr
                        external;
// Emitted events.

// A node was added to a participant group.
event NodeAdded(ParticipantGroup _participant_group, string _node);

// A node was removed from a participant group.
event NodeRemoved(ParticipantGroup _partcipant_group, string _node);

// A participant has invited a participant group to join the network.
event ParticipantGroupInvited(ParticipantGroup _participant_group,
                                uint _permission);

// A participant has uninvited a participant group from the network.
event ParticipantGroupUnInvited(ParticipantGroup _participant_group,
                                uint _permission);

// A participant group was added to the network.
event ParticipantGroupAdded(ParticipantGroup _participant_group,
                                uint _permission);

// A participant group was removed from the network.
event ParticipantGroupRemoved(ParticipantGroup _participant_group,
                                uint _permission);

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

Implementations of `Network` are required to authorize on mutators (`invite`, `uninvite`, `proposeDecider`, `addNode`, `removeNode`). The granularity of permissions is implementation dependent.

6.3.4.4 PermissioningDecider §

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

```
pragma solidity ^0.4.24;

import "./ParticipantGroup.sol";
import "./Network.sol";
interface PermissioningDecider {
// The permission the participant group now has, if approved.
function inviteApproved(Network, ParticipantGroup) external view
    returns (uint8);

// Whether the network should remove the participant group.
function inviteRevoked(Network, ParticipantGroup) 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: `ParticipantGroups` are never removed or added from the `Network`, and the `PermissioningDecider` never changes.
- AutoApprove: `ParticipantGroups` 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 `ParticipantGroup`, or changing the `PermissionDecider`.
- MajorityRules: A prospective `ParticipantGroup` needs more than half of the current `ParticipantGroups` to invite it for membership. A prospective `PermissioningDecider` needs more than half of the current groups to nominate it before this `Decider` relinquishes control.

6.3.4.5 Node Blacklisting §

Node blacklisting at the participant group level (that is, within an enterprise) is achieved by including the following additional functions in the `ParticipantGroup` smart contract.

```
interface ParticipantGroup {

...
// Add a participant within the participant group to the blacklist.
function blacklistNode(Participant, string _node) interface;

// Remove a participant within the participant group from the blacklist.
function unblacklistNode(Participant, string _node) interface;
...}
```

Blacklisting of client nodes belonging to another participant group (that is, another enterprise) in the network is achieved by including the following functions in the `Network` and `PermissioningDecider` smart contracts.

```
interface Network {

...
// Vote to add a participant in another participant group to the blacklis
voteToBlacklist(ParticipantGroup, Participant, string _node) external;

// Vote to remove a participant in another participant group from the
// blacklist.
voteToUnblacklist(ParticipantGroup, Participant, 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(ParticipantGroup _participant_group, string _node);

// Emitted when a node is removed from the blacklist.
event NodeUnblacklisted(ParticipantGroup _participant_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);
...

}
```

6.3.5 Inter-chain §

With the rapid expansion in the number of different blockchains and ledgers, inter-chain mediators are necessary to allow interaction between blockchains. 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.

[E] ICHN-010: Enterprise Ethereum implementations *MAY* provide inter-chain mediation capabilities to enable interaction with different blockchains.

6.3.6 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.

[C] ORCL-010: Enterprise Ethereum implementations *SHOULD* provide the ability to securely interact with oracles to send and receive real-world information.

7. Privacy and Scaling Layer §

7.1 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 7.1.4 Privacy Levels

7.1.1 On-chain §

Various on-chain techniques are proposed to improve the security and privacy capabilities of networks.

> NOTE: On-chain Security Techniques
>
> Future on-chain security techniques could include techniques such as ZK-SNARKS, range proofs, or ring signatures.
>
> Private transactions are an example of future work to support privacy requirements.

7.1.2 Off-chain (Trusted Computing) §

Off-chain Trusted Computing, coupled to a blockchain as an off-chain processing environment, can help provide secure, efficient, and scalable processing for transactions and smart contracts as well as ensuring privacy of sensitive contact data.

[C] OFFCH-010: Enterprise Ethereum implementations *MAY* provide the ability for off-chain, trusted execution of transactions and smart contracts.

7.1.3 Private Transactions §

Many users and operators of Enterprise Ethereum implementations 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. Users of Enterprise Ethereum 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*

- *Unrestricted private transactions*.

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

[P] PRIV-010: 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:

- **[P] PRIV-020:** Implementations *MUST* support masking or obfuscation of the payload data when stored in restricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-030:** Implementations *MUST* support masking or obfuscation of the payload data when in transit in restricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-040:** Implementations *MAY* support masking or obfuscation of the metadata when stored in restricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-050:** Implementations *MAY* support masking or obfuscation of the metadata when in transit in restricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-060:** Nodes that relay a restricted private transaction but are not participants in that transaction *MUST NOT* store transaction payload data.
- **[P] PRIV-070:** Nodes that relay a restricted private transaction but are not participants in that transaction *SHOULD NOT* store transaction metadata.
- **[P] PRIV-080:** The implementation of the [[JSON RPC API]] `eea_sendTransactionAsync` or `eea_sendTransaction` methods (if implemented) with the `restriction` parameter set to restricted, *MUST* result in a restricted private transaction.

EXAMPLE 6: Restricted Private Transactions

Private transactions can 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 are kept private between related parties, so unrelated parties have no access to the content of the transaction, the sending party, or the list of participating addresses. In fact, a private smart contract has a similar relationship to the blockchain that hosts it as a private blockchain network 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.

When implementing unrestricted private transactions:

- **[P] PRIV-090:** 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).
- **[P] PRIV-100:** Implementations *SHOULD* support masking or obfuscation of the sender identity when stored in unrestricted private transactions (for example, using stealth addresses).
- **[P] PRIV-110:** Implementations *SHOULD* support masking or obfuscation of the payload data when stored in unrestricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-120:** Implementations *MUST* support masking or obfuscation of the payload data when in transit in unrestricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-130:** Implementations *MAY* support masking or obfuscation of the metadata when stored in unrestricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-140:** Implementations *MAY* support masking or obfuscation of the metadata when in transit in unrestricted private transactions (for example, using cryptographic encryption).
- **[P] PRIV-150:** Nodes that relay an unrestricted private transaction but are not participants in that transaction *MAY* store payload data.
- **[P] PRIV-160:** Nodes that relay an unrestricted private transaction but are not participants in that transaction *MAY* store transaction metadata.
- **[P] PRIV-170:** The implementation of the [[JSON RPC API]] `eea_sendTransactionAsync` or `eea_sendTransaction` methods (if implemented) with the `restriction` parameter set to unrestricted *MUST* result in an unrestricted private transaction.

EXAMPLE 7: Unrestricted Private Transactions

Obfuscated data that is replicated across all nodes can be reconstructed from any node, albeit in encrypted form. Mathematical transactions on numerical data are intended 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 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: Implementations *SHOULD* be able to extend the set of participants in a private transaction (or forward the private transaction in some way).

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

NOTE: Restricted vs Unrestricted Private Transactions

The differences between restricted 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 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

[P] PRIV-200: Implementations *SHOULD* be able to authorize the types of transactions an Ethereum Account can submit, providing separate permissioning for the ability to:

- Deploy a smart contract.
- Call a function that changes the state of a smart contract.
- Perform a simple value transfer.

7.1.4 Privacy Levels §

Implementations can support different levels of privacy, as outlined in Table 3 below, 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 6.1.1 Nodes and 6.1.2 Participants. 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 7.1.3 Private Transactions 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 6.1.1 Nodes, 6.1.2 Participants, and 6.1.4 Additional Permissioning Requirements. 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 7.1.3 Private Transactions. 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 7.1.3 Private Transactions.
B	Best practice permissioning	Implementations provide support for Privacy Level C and all the *SHOULD* peer node connectivity and permissioning requirements from Sections 6.1.1 Nodes, 6.1.2 Participants, and 6.1.4 Additional Permissioning Requirements.
C	Baseline privacy and permissioning	Implementations provide support for all the *MUST* peer node connectivity and permissioning requirements from Sections 6.1.1 Nodes and 6.1.2 Participants, and either: • All the *MUST* and *MUST NOT* restricted private transaction requirements in Section 7.1.3 Private Transactions. • All the *MUST* and *MUST NOT* unrestricted private transaction requirements in Section 7.1.3 Private Transactions.

7.2 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.

7.2.1 On-chain (Layer 1) §

On-chain scaling at layer 1 improves the capability to handle more transactions by changing the underlying Ethereum protocol.

7.2.2 On-chain (Layer 2) §

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

[P] SCAL-010: Enterprise Ethereum implementations *SHOULD* provide the ability for improved on-chain processing rates of transactions and smart contracts using layer 1 and layer 2 solutions.

7.2.3 Off-chain (Compute) §

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

[C] SCAL-020: Enterprise Ethereum implementations *SHOULD* provide the ability for off-chain processing of transactions and smart contracts.

7.2.4 Performance §

Performance refers to the overall performance of the network. Ideally, increased usage of the network does not degrade its performance.

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

> EXAMPLE 8: EEA Certification
>
> Certificates of Certification might require minimum transaction rates in terms of [ERC-20] smart contract executions per second, or other measures.

8. Core Blockchain Layer §

8.1 Storage and Ledger Sublayer §

[P] STOR-010: Enterprise Ethereum implementations *SHOULD* implement data storage requirements necessary to operate a public Ethereum client.

[C] STOR-020: 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.

[C] STOR-030: 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.

[P] STOR-040: A smart contract operating on private state *SHOULD* be permitted to access private state created by other smart contracts involving the same participants.

[P] STOR-050: A smart contract operating on private state *MUST NOT* be permitted to access private state created by other smart contracts involving different participants.

[P] STOR-060: 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.

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

EXAMPLE 9: Storing File Objects

Implementations might choose to provide additional APIs outside this Specification (such as the WebDAV protocol described in [RFC4918]) for interaction with file objects.

8.1.1 Finality §

[P] FINL-010: When a deterministic consensus algorithm is used, transactions *SHOULD* be considered final after a defined interval or event. For example, a set time period or a set number of blocks created since the transaction was included in a block.

8.2 Execution Sublayer §

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

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

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

[P] EXEC-040: Implementations *SHOULD* provide support for the compilation, storage, and execution of precompiled contracts.

Trusted Computing ensures only authorized parties can execute smart contracts on an execution environment related to a given consortium network.

[C] EXEC-050: Implementations *MAY* offer support for Trusted Computing.

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

[C] EXEC-060: Implementations *SHOULD* provide configurable encryption options for use in conjunction with consortium networks.

8.3 Consensus Sublayer §

[P] CONS-020: Implementations *MUST* be capable of supporting multiple consensus algorithms.

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

[P] CONS-050: One or more consensus algorithms *MAY* support operations on sidechain networks.

[P] CONS-080: 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.

[P] CONS-100: Implementations *MAY* support other consensus algorithms.

[P] CONS-110: 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.

9. Network Layer §

9.1 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-020: 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.

[P] PROT-030: Implementations *SHOULD* support, at a minimum, the eth/62 and eth/63 [Ethereum-Wire-Protocol] implementations.

[P] PROT-040: Implementations *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 10: 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. Common nodes 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: Implementations *SHOULD* implement the Whisper protocol [Whisper-protocol].

10. Anti-Spam §

This section refers to mechanisms to prevent 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 6.2.2 Enterprise Management Systems, or implemented by the Enterprise Ethereum client, as described in the following requirement.

[P] SPAM-010: 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 11: 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.

11. Cross-client Compatibility §

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

[P] XCLI-010: 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 8.2 Execution Sublayer.

[P] XCLI-020: 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 12: Extensions to the Public Ethereum API

Extensions to public Ethereum APIs could include enterprise 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 precompile 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 Ethereum clients.

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

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

12. Synchronization and Disaster Recovery §

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

Various techniques can help do this efficiently. For an Enterprise Ethereum chain 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 Terminology §

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

Table 4 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 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 networks.
Consortium	An Ethereum network, Enterprise or public, not part of the Ethereum MainNet.

Network	
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	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 consists of one or more groups of individual actors with different roles, and a collection of nodes belonging to the enterprise.
Ethereum	An open-source, public blockchain-based, distributed computing platform featuring smart contract (programming) functionality. [Ethereum]
Ethereum Account	An established relationship between a participant and an Ethereum network. Having an Ethereum account allows participants to interact with Ethereum, for example to submit transactions or deploy smart contracts. See also Wallet.
Ethereum MainNet	The public Ethereum blockchain network with the network identifier of 1.
Ethereum Name Service (ENS)	A secure and decentralized way to address resources both on and off the Ethereum blockchain using simple, human-readable names.
Ethereum Virtual Machine (EVM)	A runtime computing environment for the execution of smart contracts on Ethereum. Each node operates an EVM.

Finalized Transaction	A finalized transaction is definitively part of the blockchain and cannot be removed. A transaction reaches this state after some event defined for the relevant blockchain occurs. For example, an elapsed amount of time or a specific number of blocks added.
Formal Verification	Mathematical verification of the logical correctness of a smart contract in the context of the EVM.
Gas	A virtual pricing mechanism for transactions and smart contracts, used by Ethereum to protect against Denial of Service attacks and spam. Gas is defined in the [Ethereum-Yellow-Paper].
Group	A collection 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.
Hardware Security Module (HSM)	A physical device to provide strong and 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].
Inter-chain	Inter-chain mediators allowing interaction between different blockchains and ledgers.
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 Computing services
Off-Chain (Layer 2) Scaling Mechanism	Extensions to public Ethereum using smart contracts, or techniques such as [Plasma], and [state-channels], to facilitate increased transaction speeds. For more information, see [Layer2-Scaling-Solutions].

Off-Chain (Trusted Execution)	Offloading of compute intensive processing for scalability improvements, whilst maintaining transaction privacy.
On-Chain Privacy Mechanism	Extensions to public Ethereum, such as ZK-SNARKS, or privacy-preserving trusted computing compute, enabling private transactions.
On-Chain (Layer 1) Scaling Mechanism	Extensions to the public Ethereum base protocol, such as [sharding], to facilitate increased transaction speeds. For more information, see [Layer2-Scaling-Solutions].
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 User or Enterprise being a party to a transaction within Enterprise Ethereum.
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, this right of privacy also applies to organizations to the extent permitted by law.
Private State	State data that is not shared in the clear in the globally replicated state tree. This data can represent bilateral or multilateral arrangements between counterparties, for example in private transactions.
Private Transaction	A transaction where some information about the transaction, such as the payload data, or the sender and recipient, is only available to the subset of network participants who are parties to that transaction.
Private	A subsystem of an Enterprise Ethereum system for implementing privacy and

Transaction Manager	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 nodes. 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 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.
Smart Contract Language	A programming language and associated tooling used to create smart contracts. For example, [Solidity] and [LLL].
Sync	Synchronization of state with the state held by other nodes.
Transaction	A request to execute operations that change state in a blockchain network. Transactions can involve one or more participants.
Trusted Execution	Enabling privacy during code execution.
Trusted Computing	A system available from the blockchain to execute transactions and smart contracts outside the core blockchain. This can be used to private improved privacy, performance, or security. Such systems can be hardware or software-based, depending on the use case.
Unspent Transaction Output	Output from a transaction that can be spent as an input for a new transaction.
User	A human or an automated process interacting with Enterprise Ethereum through the 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.
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. See also Ethereum account.
Whisper	A network protocol designed for ĐApps to communicate with one another.
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.

A.2 Acknowledgments §

The EEA acknowledges and thanks the many people who contributed to the development of this Specification.

This specification builds on the work of all who contributed to the previous version, whom we hope are all acknolwedged in the Enterprise Ethereum Client Specification v1. The editors would especially like to thank David Hyland-Wood for his incomparable effort as co-editor of that version.

We apologise to anyone whose name was inadvertently ommitted. Please advise us at https://entethalliance.org/contact/ of any errors or omissions.

A.3 Changes §

Full details of all changes since the version 1.0 release of this Specification are available in the GitHub repository for this Specification.

This section outlines substantive changes made to the specification:

- Pull Request 253: Update terminology from "Trusted Execution Environment" to "Trusted Computing", and clarify that it is not a specific hardware solution, but can be based on software.
- Pull Request 212: Add requirement PERM-070 requiring a set of smart contracts for network permissions:
  - `Participant`
  - `ParticipantGroup`
  - `Network`
  - `PermissioningDecider`
- Pull Request 60: Add `eea_sendTransaction`, the synchronous method for sending transaction using RPC.

- Pull Request 286: Add requirement XCLI-050 to clarify that implementations *MUST* support precompiled smart contracts defined for Public Ethereum.
- Pull Request 205: Update requirement JRPC-010 to specify a list of required methods for the JSON-RPC API, and add requirements:
    - JRPC-007 to require compatibility with current method versions.
    - JRPC-011 to allow other methods to be implemented.
    - JRPC-015 to require support for [[JSON RPC]] calls using websockets.
    - JRPC-040 to require support for `debug_traceTransaction`.
    - JRPC-050, JRPC-060, and JRPC-070 describing requirements to support the [JSON-RPC-PUB-SUB] API.
    - JRPC-080 to reserve the `eea_` namespace for Enterprise Ethereum extensions.
- Pull Request 256: Add non-normative Section 3. Security Considerations, and remove requirement ONCH-010 mandating an untestable implementation strategy instead of a requirement on existing implementations.
- Pull Request 261: Update requirement PART-050 and add requirement PART-055 to clarify that implementations *MUST* support connecting to an identity server, but also support anonymous participants.
- Pull Request 218: Remove requirements:
    - CONS-010 and CONS-040 requiring support for the Ethereum Mainnet consensus algorithm.
    - CONS-060 requiring documentation for consensus algorithms.
    - CONS-070 requiring consensus algorithm implementations to be modular and configurable.
- Pull Request 192: Add requirements XCLI-030 and XCLI-040 to implement the Ethereum gas mechanism, allowing for a gas price of zero.
- Pull Request 229: Update requirements OFFCH-010 and EXEC-050 to state clients *MAY* (instead of *SHOULD*) implement Off-chain transactions / Trusted Computing.
- Pull Request 147: Update requirement SCAL-010 about the ability for improved on-chain processing rates of transactions and smart contracts using layer 1 and layer 2 solutions.
- Pull Request 225: Update requirement EXEC-040 to state clients *SHOULD* (instead of *MAY*) support EEA-defined precompiled smart contracts.
- Pull Request 198: Add requirement PROT-060 that clients *SHOULD* implement the [Whisper-protocol].
- Pull Request 216: Remove requirements SYNC-010 and SYNC-020 about backup mechanisms for clients.

- Pull Request 215: Remove requirement PERF-040 about support for changing the genesis block.
- Pull Request 213: Remove requirements PERF-010, PERF-020, and PERF-030 about performance.
- Pull Request 217: Remove requirement CONS-090 about support for IBFT [EIP-650] consensus.
- Pull Request 214: Add requirement XCLI-005 about aligning public Ethereum features with Byzantium release. Remove requirement CONF-010 because we no longer require that clients can operate on the public Ethereum network.
- Pull Request 145: Add requirements PART-070 and PRIV-200 about finer grained permissioning over different transaction types, closing issue 99.
- Pull Request 144: Add requirement ACCT-010 that clients *SHOULD* support Ethereum account whitelisting, and requirement ACCT-020 that if account whitelisting is implemented, it be done through an API.
- Pull Request 67: Add the `Web3_clientcapabilities` RPC method to find information about client capabilities, closing issue 58. The information available through this method is expected to be expanded.

B. References §

B.1 Normative references §

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

[debug-traceTransaction]
debug_traceTransaction. URL: https://github.com/ethereum/go-ethereum/wiki/Management-APIs

[DEVp2p-Wire-Protocol]
ÐΞVp2p Wire Protocol. URL: https://github.com/ethereum/wiki/wiki/ÐΞVp2p-Wire-Protocol

[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

[ERC-20]

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

[ERC-223]

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

[ERC-621]

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

[ERC-721]

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

[ERC-827]

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

[Ethereum]

Ethereum Foundation. URL: https://www.ethereum.org/foundation

[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

[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-PUB-SUB]

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

[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

[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 §

[Byzantine-Fault-Tolerance]

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

[DIF]

Decentralized Identity Foundation. URL: http://identity.foundation/

[EIP-648]

Easy Parallelizability. Ethereum Foundation. URL: https://github.com/ethereum/EIPs/issues/648

[eWASM]

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

[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

[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#

[protocol-buffers]

A language-neutral, platform-neutral extensible mechanism for serializing structured data. Google Developers. URL: https://developers.google.com/protocol-buffers/

[RAFT]

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

[Remix]

Ethereum Tools for the Web. Ethereum Foundation. URL: https://github.com/ethereum/remix

[RFC4918]

HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). L. Dusseault, Ed.. IETF. June 2007. Proposed Standard. URL: https://tools.ietf.org/html/rfc4918

[Truffle]

Ethereum Development Framework. ConsenSys. URL: https://truffleframework.com/

↥