HYPERLEDGER
BLOCKCHAIN TECHNOLOGIES FOR BUSINESS

Hyperledger Architecture, Volume II
Smart Contracts

This is the second in a series of papers from the Hyperledger Architecture Working Group (WG). These papers describe a generalized reference architecture for permissioned blockchain networks and share the recommendations of the Hyperledger Architecture WG with the end goal of guiding all Hyperledger projects towards modular designs. These papers also serve as a vendor-neutral resource for technical blockchain users and developers interested in using permissioned blockchain networks.

About this paper

This paper on smart contracts provides a generalized reference architecture for smart contracts. The paper also explores how various Hyperledger blockchain frameworks—namely Burrow, Fabric, Indy, and Sawtooth—implement the reference architecture.

Other papers in this series

The first paper in this series introduced the Hyperledger business blockchain design philosophy and provided a generalized reference architecture for consensus. This paper is available for download from **Hyperledger Architecture Positioning Paper Volume 1: Introduction to Hyperledger Business Blockchain Design Philosophy and Consensus**.

Forthcoming papers in this series will expand on the generalized reference architecture to include the following business blockchain components: Communication Layer, Data Store Abstraction, Crypto Abstraction, Identity Services, Policy Services, APIs, and Interoperation.

ABOUT HYPERLEDGER

Hyperledger is an open source collaborative effort created to advance cross-industry blockchain technologies. It is a global collaboration including leaders in finance, banking, Internet of Things, supply chains, manufacturing, and Technology. The Linux Foundation hosts Hyperledger under the foundation.

Table of Contents

Overview of Hyperledger Architecture

As an umbrella project Hyperledger does not have a single architecture per se. However all Hyperledger projects follow a design philosophy that includes a modular extensible approach, interoperability, an emphasis on highly secure solutions, a token-agnostic approach with no native cryptocurrency, and ease-of-use.

The Hyperledger Architecture WG has distinguished the following business blockchain components:

- **Consensus Layer** - Responsible for generating an agreement on the order and confirming the correctness of the set of transactions that constitute a block. Read the **Hyperledger Architecture Consensus Paper**

- **Smart Contract Layer** - Responsible for processing transaction requests and determining if transactions are valid by executing business logic.

- **Communication Layer** - Responsible for peer-to-peer message transport between the nodes that participate in a shared ledger instance.

- **Data Store Abstraction** - Allows different data-stores to be used by other modules.

- **Crypto Abstraction** - Allows different crypto algorithms or modules to be swapped out without affecting other modules.

- **Identity Services** - Enables the establishment of a root of trust during setup of a blockchain instance, the enrollment and registration of identities or system entities during network operation, and the management of changes like drops, adds, and revocations. Also, provides authentication and authorization.

- **Policy Services** - Responsible for management of various policies specified in the system, such as the endorsement policy, consensus policy, or group management policy. It interfaces and depends on other modules to enforce the various policies.

- **APIs** - Enables clients and applications to interface to blockchains.

- **Interoperation** - Supports the interoperation between different blockchain instances.

Introduction to Smart Contracts

A smart contract is essentially business logic running on a blockchain.

Smart contracts can be as simple as a data update, or as complex as executing a contract with conditions attached. For example, a smart contract can update an account balance, with validation to ensure that enough money is in an account before doing a debit.

A more complex smart contract can be written to stipulate that the cost of shipping an item depends on when it arrives. With the terms agreed by both parties and written to the ledger, the appropriate funds change hands automatically when the item is received.

There are two different types of smart contracts:

- **Installed smart contracts** install business logic on the validators in the network before the network is launched.

- **On-chain smart contracts** deploy business logic as a transaction committed to the blockchain and then called by subsequent transactions. With on-chain smart contracts, the code that defines the business logic becomes part of the ledger.

Which Hyperledger Frameworks Support Smart Contracts?

Four of the Hyperledger blockchain frameworks support smart contracts:

- Hyperledger Burrow

- Hyperledger Fabric

- Hyperledger Iroha

- Hyperledger Sawtooth

Across all these frameworks, the smart contract layer is responsible for processing transaction requests and determining if transactions are valid by executing business logic. Each framework supports smart contracts in a slightly different way, as shown **on page 8** and discussed in later sections of this paper.

How Smart Contracts Are Processed

Figure 1 shows how a request sent to a smart contract is processed. **Inputs** include the contract identifier, the transaction request, any dependencies that may exist, and the current state of the ledger.

The **contract interpreter**, the block in the middle, is loaded with the current state of the ledger and the smart contract code. When the contract interpreter receives a request, it immediately checks and then rejects any invalid requests.
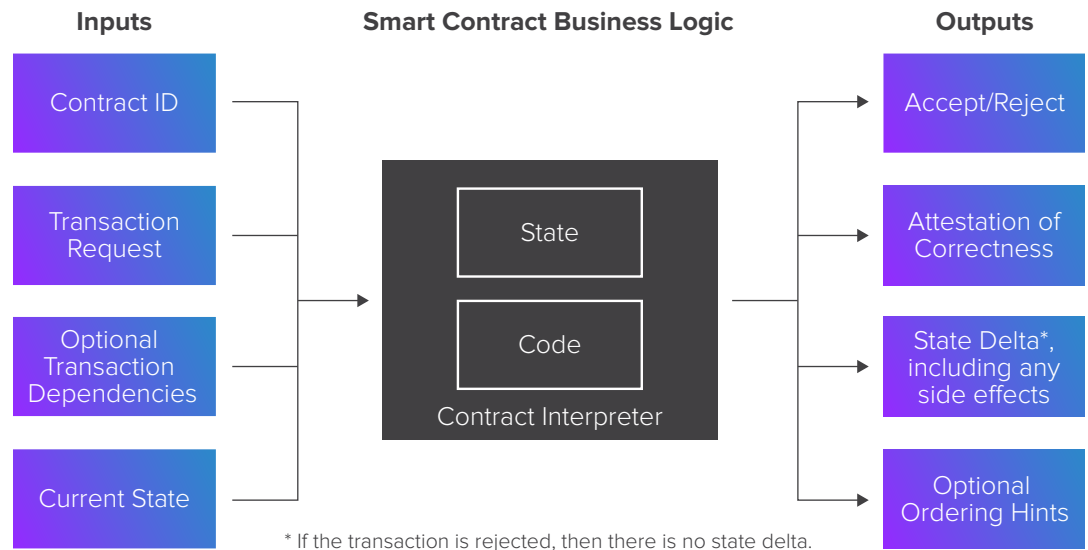
Inputs	Smart Contract Business Logic	Outputs
Contract ID		Accept/Reject
Transaction Request	State	Attestation of Correctness
Optional Transaction Dependencies	Code	State Delta*, including any side effects
Current State	Contract Interpreter	Optional Ordering Hints

* If the transaction is rejected, then there is no state delta.

Figure 1: How Smart Contracts Process Requests

The appropriate **Outputs** are generated if the request is valid and accepted. These outputs include a new state and any side effects.

When all the processing is complete, the interpreter packages the new state, an attestation of correctness, and any required ordering hints for the consensus services. That package is sent to the consensus service for final commitment to the blockchain.

The smart contract layer validates each request by ensuring that it conforms to policy and the contract specified for the transaction. Invalid requests are rejected and may be dropped from inclusion within a block, depending on the framework.

Validation errors can be divided into two categories: syntax errors and logic errors.

For syntax errors such as invalid inputs, unverifiable signature, and any repeated transaction (due to errors or replay attacks), the request should be dropped.

Logic errors are more complex, and the decision whether to continue processing should be policy-driven. For example, if a request would result in double-spend or versioning control failure, you might want to log that request for auditing, if the policy requires.

The result of validating a request in a smart contract is a transaction that captures the transition to the new state. Each Hyperledger framework represents the state transition differently; for example, Hyperledger Fabric uses change sets, while Hyperledger Sawtooth uses cryptographically verifiable states.

This separation between contract execution and consensus complicates the commitment to the blockchain. For example, two requests executed in parallel can result in inconsistencies in the contract state unless they are ordered in a specific way. In general, the consensus layer handles ordering. However, the smart contract has the option to provide hints about properties like ordering.

Processing a request can generate side effects: results that are not captured in the state delta. Side effects can include event notifications, requests to other smart contracts, modifications to global resources, or even irreversible actions like releasing sensitive information or shipping a package.

Transaction Dependencies

All four Hyperledger frameworks that support smart contracts—Hyperledger Burrow, Hyperledger Fabric, Hyperledger Iroha, and Hyperledger Sawtooth—assume that transactions are atomic. In other words, a transaction is considered either not yet started or else completed; a transaction cannot be partly completed. This ensures transaction integrity.

When dependent actions (if this then that) must be committed together, some frameworks assume that the dependent actions are captured in a single smart contract. Others permit batching of transactions across multiple smart contracts. Thus, smart contracts can specify dependencies between multiple transactions which must be committed atomically. These references can be either implicit or explicit.

With an **implicit reference**, it is usually impossible to determine the order in which transactions must be applied. Bitcoin solves this by repeatedly trying to apply the transaction, which results in transactions being gradually executed as their prerequisites are satisfied. This behavior requires a transaction staging component, such as **mempool**, which takes care of any expiring transactions the system could not apply.

With an **explicit reference**, the user submitting the transactions specifies the ordering with some form of transaction identifiers.

Just as implicit and explicit references impact the formation of the block, so does the **dependency graph**. The dependency graph can be either **cyclic** or **acyclic**, in other words, circular or non-circular.

With cyclic dependencies, the entire group containing the cycle must be included in the same block. For example if transaction A is dependent on transaction B and transaction B is dependent on transaction C and transaction C is dependent on transaction A, then all three transactions must be included in the same block.

With acyclic dependencies, any portion of the sequence may be included in the block, as long as transactions are included according to the dependency graph. For example, if transaction B is dependent on transaction A, then transaction A must be committed prior to transaction B.

With implicit dependencies, cycles may be complex to resolve, which can prevent certain transactions from being validated in parallel and committed. On the other hand, explicit

dependency references can allow for parallel execution of non-conflicting transactions within the same block.

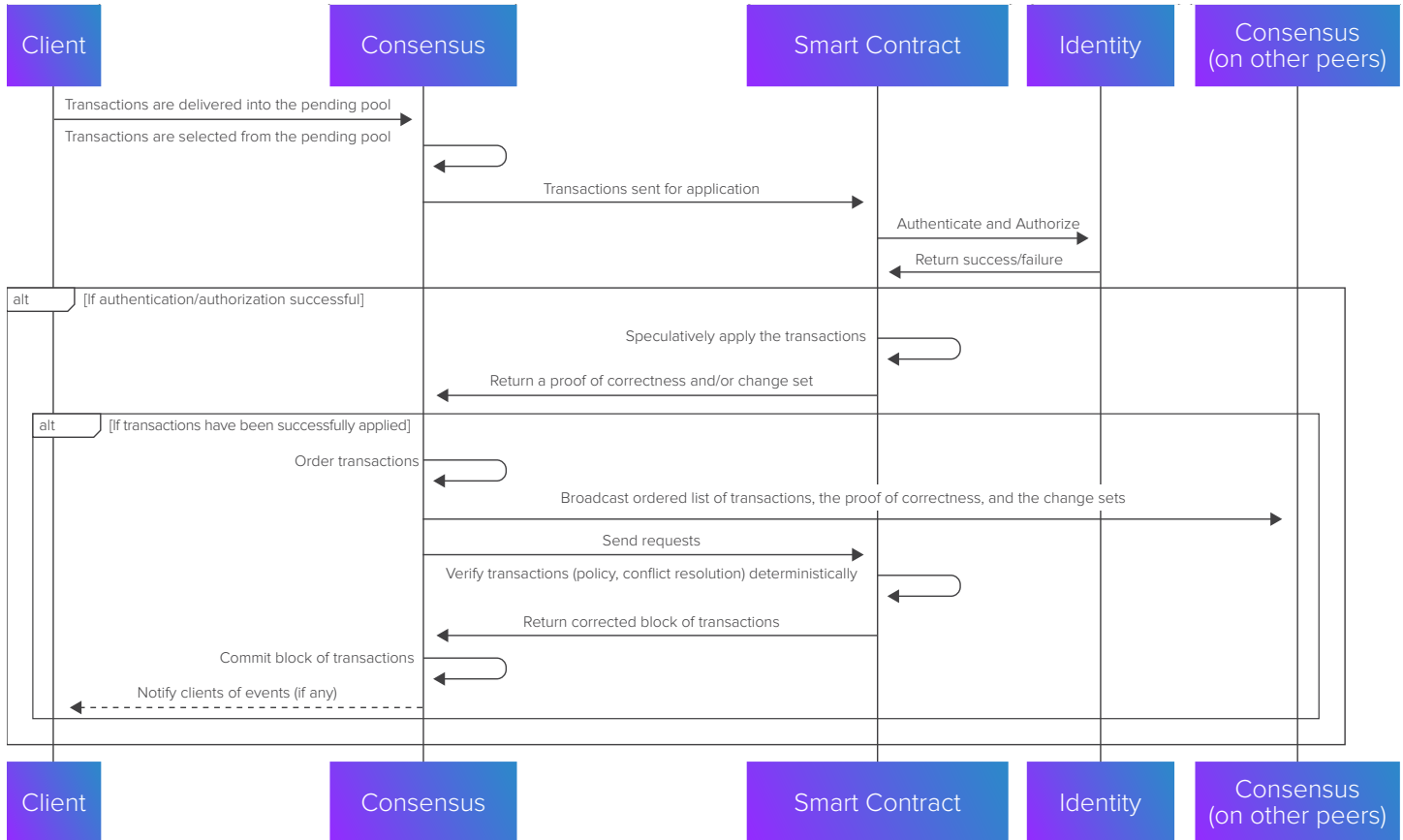## How Smart Contracts Interact with Other Architectural Layers

Figure 2: Hyperledger Smart Contracts and Other Layers

Figure 2 shows how smart contracts fit in with the other blockchain architectural layers. This is a generalized view, since the designers of any Hyperledger framework may choose to implement these steps differently.

In general, the smart contract layer works very closely with the consensus layer. Specifically, the smart contract layer receives a proposal from the consensus layer[1]. This proposal specifies which contract to execute, the details of the transaction including the identity and credentials of the entity asking to execute the contract, and any transaction dependencies.

The smart contract layer uses the current state of the ledger and input from the consensus layer to validate the transaction.

While processing the transaction, the smart contract layer uses the identity services layer to authenticate and authorize the entity asking to execute the smart contract. This ensures two things: that the entity is known on the blockchain network, and that the entity has the appropriate access to execute the smart contract. Identity can be provided through several methods: simple key-based identities, identities and credentials managed through the ledger, anonymous credentials, or managed identity services from an external certificate authority.

[1] For a detailed description of permissioned blockchain consensus and various approaches within Hyperledger, please see the paper: **Hyperledger Architecture Positioning Paper Volume 1: Introduction to Hyperledger Business Blockchain Design Philosophy and Consensus**

After processing the transaction, the smart contract layer returns whether the transaction was accepted or rejected. If the transaction was accepted, the smart contract layer also returns an attestation of correctness, a state delta, and any optional ordering hints needed to ensure the transaction dependencies are taken into account. The state delta includes the change sets and any side effects that should take place when the transaction is successfully committed by the peers.

Smart Contract Integrity and Availability

To ensure the integrity and availability of the blockchain network and the smart contract layer, enterprise blockchains must control access to certain resources. Since smart contracts are programs, they are vulnerable to malicious attack, coding errors, and poor design. A breakdown in any of these areas can compromise the integrity or availability of the blockchain system.

Hyperledger recommends the following four safeguards to use with the smart contract layer to help ensure integrity and availability.

1. Denial-of-Service Protection

In a malicious denial-of-service (DoS) attack, the perpetrator seeks to make a machine or network resource unavailable to its intended users by temporarily disrupting services. This is often done by flooding the computer, server, or network with a series of requests aimed at overloading the system so that it cannot serve legitimate requests.

In permissioned blockchain networks that operate in an environment of partial trust, malicious DoS attacks can be reduced by using identity management and access control policies. These help ensure that only known entities can participate in the blockchain network. If any bad actor gains access to the network through spoofing or other misappropriation of identity, and then starts a DoS attack, the source can be identified, isolated, and removed.

2. Sandboxing

A securely-designed blockchain system must ensure that smart contracts can only execute the functions needed to perform a given transaction. Without safeguards, malicious or erroneous smart contract code could corrupt the network, risking theft or exposure of private data.

A variety of proven techniques for isolating the execution environment can be applied to the smart contract layer, such as partitions and containers.

3. Resource Management / Flow Control

Whether by mistake or malicious intent, a faulty smart contract can consume excessive network resources, prevent other contracts from executing, or otherwise degrade the system. Flow control refers to techniques such as back pressure and overload protection which ensure the availability of resources.

Business blockchain systems can apply time or token-based resource management techniques to ensure that any particular smart contract does not consume excessive resources.

4. Application Lifecycle Management (ALM)

Since smart contracts are application code, they must be managed across the full software lifecycle, including testing, deployment, updates, and decommissioning. Business blockchain systems must provide for these functions either natively or by integrating with ALM tools.

Smart Contracts in the Hyperledger Frameworks

Table 1. Smart Contract Implementations in Hyperledger Frameworks

Framework	Smart Contract Technology	Smart Contract Type	Language(s) for Writing Smart Contracts
Hyperledger Burrow	Smart contract application engine	On-Chain	Native language code
Hyperledger Fabric	Chaincode	Installed	Golang (> v1.0) or Javascript (> v1.1)
Hyperledger Indy	None	None	None
Hyperledger Iroha[2]	Chaincode	On-chain	Native language code
Hyperledger Sawtooth	Transaction families	On-Chain and Installed	C++, Go, Java, JavaScript, Python, Rust, or Solidity (through Seth)

Since different businesses have different requirements for blockchain, the Hyperledger community is working on several different ways to implement smart contracts.

Table 1 compares the smart contract implementations used across Hyperledger frameworks. For each framework, the table lists the smart contract technology used, the type of smart contract (either installed or on-chain), and the main programming language(s) used to write smart contracts.

Smart Contracts in Hyperledger Burrow

Hyperledger Burrow is a permissioned smart contract machine. It provides a modular blockchain client with a permissioned smart contract interpreter built to the specifications of the Ethereum Virtual Machine (EVM), with some extensions and some changes yet to be incorporated.

Hyperledger Burrow couples its EVM execution engine with the Tendermint consensus engine over an application-consensus interface called ABCI. The application state consists of all accounts, the validator set, and the name registry. Accounts in Hyperledger Burrow have permissions and either contain smart contract code or correspond to a public-private key pair.

A transaction that calls on the smart contract code in a given account will activate the execution of that account's code in a permissioned virtual machine.

The smart contract application engine provides much of the value of Hyperledger Burrow. This engine provides the interfaces necessary to be used as a library. Some of its key subcomponents are outlined below.

Application Global State

The application state consists of all accounts, the validator set, and Hyperledger Burrow's built-in name registry. A transaction that calls on the smart contract code in a given account will activate the execution of that account's code in a permissioned virtual machine.

Secure Native Functions

Secure native functions provide the ground rules that all accounts and all smart contract code must follow. They do not reside as EVM code, but are exposed to the permissioned EVM via interface contracts. Permissioning is enforced through secure native functions. And permissioning underlies the execution of all smart contract code.

[2] Preliminary information subject to change

Hyperledger Burrow envisions a secure native functions framework supporting the use of native language code for higher performance and security. Secure native functions can be exposed to the permissioned EVM within Hyperledger Burrow.

Secure native functions can also be structured to support better smart contract performance. Secure native functions can provide a range of privileged-level functions to ecosystem applications which should be built natively and exposed to the permissioned EVM. Efforts are ongoing to systematize this and add advanced capabilities to support a wide variety of users.

Permission Layer

Hyperledger Burrow comes with a capabilities-based, evolvable permissioning layer.

The network is booted with an initial set of accounts with permissions plus a global default set of permissions. Network participants with the correct permission can modify the permissions of other accounts by sending an appropriate transaction type to the network. This transaction is vetted by the network validators before the permissions are updated on the target account. Through the EVM, further sophisticated roles-based permissioning can be leveraged through Hyperledger Burrow's roles on each account. Roles can be updated through discrete transactions or smart contracts.

In addition, Hyperledger Burrow exposes the ability for smart contracts within the permissioned EVM to modify the permission layer and account roles. Once a contract with this functionality has been deployed to a chain, a network participant with the appropriate permissions can grant the contract that capability.

Permissioned EVM

This virtual machine is built to observe the Ethereum operation code specification and assert that the correct permissions have been granted. An arbitrary but finite amount of gas—the execution fee for every operation run on Ethereum—is handed out for every execution. This ensures that the execution will be completed within some finite period of time.

Transactions need to be formulated in a binary format that can be processed by the blockchain node using an Application Binary Interface (ABI). A range of open source tooling from Monax and the Ethereum community enable users to compile, deploy, and link smart contracts compiled for the permissioned EVM, and to formulate transactions that call smart contracts.

The permissioned EVM itself is designed and implemented as a stateless function to deterministically and verifiably transition the EVM state given a transaction. This code package could be integrated in different Hyperledger projects. For example, the extensible transaction families in Hyperledger Sawtooth allow us to think of the permissioned EVM as a transaction processor in Sawtooth's permissioned ledger framework.

Gateway

Hyperledger Burrow exposes RESTful and JSON-RPC endpoints for clients to interact with the blockchain network and the application state either by broadcasting transactions or by querying the current state of the application.

Websockets allow interfacing components to subscribe to events. This is particularly valuable since the consensus engine and smart contract application engine can give unambiguously finalized results to transactions after each block.

Signing

Hyperledger Burrow accepts client-side formulated and signed transactions. An interface for remote signing is available. External signing solutions are crucial for Hyperledger Burrow's users since these allow the blockchain nodes to run on commodity hardware.

Interfaces

Hyperledger Burrow also uses boot and runtime interfaces, largely through files read by the blockchain node at boot. Of course, Burrow also includes a remote procedure call (RPC) which allows for interfacing with the node during runtime.

Smart Contracts in Hyperledger Fabric

A smart contract in Hyperledger Fabric is a program, called chaincode. Chaincode can be written in **Go**, JavaScript (**node.js**), and eventually other programming languages such as Java that implement a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic that members of the network have agreed to. The state created by a chaincode is scoped exclusively to that chaincode and can't be accessed directly by another chaincode. However, with the appropriate permission, a chaincode in the same network can invoke another chaincode to access its state.

There are two different types of chaincode to consider:

- System chaincode
- Application chaincode

System chaincode typically handles system-related transactions such as lifecycle management and policy configuration. However the system chaincode API is open for users to implement their application needs as well.

Application chaincode manages application states on the ledger, including digital assets or arbitrary data records.

A chaincode starts with a package that encapsulates critical metadata about the chaincode, including the name, version, and counterparty signatures to ensure the integrity of the code and metadata. The chaincode package is then installed on the network nodes of the counterparties.

An appropriate member of the network (as controlled by policy configuration) activates the chaincode by submitting an instantiation transaction to the network. If the transaction is approved, the chaincode enters an active state where it can receive transactions from users via client-side applications.

Any chaincode transactions that are validated are appended to the shared ledger. These transactions can then modify the world state accordingly. Any time after a chaincode has been instantiated, it can be upgraded through an upgrade transaction.

Using Chaincodes to Develop Business Contracts and Decentralized Applications

There are generally two ways to develop business contracts for Hyperledger Fabric:

- To code individual contracts into standalone instances of chaincode

- To use one chaincode to handle all contracts (of certain types) and have it expose APIs to manage lifecycle of those contracts. This second approach is probably more efficient.

Using Chaincodes to Define and Manage Assets

Users of Hyperledger Fabric can also use chaincode to define assets and the logic that manages them.

In most blockchain solutions, there are two popular approaches to defining assets:

- The stateless UTXO (unspent transaction output) model, where account balances are encoded into past transaction records

- The account model, where account balances are kept in state storage space on the ledger

Each approach has benefits and drawbacks. Hyperledger Fabric does not require one over the other. Instead, it ensures that both approaches are easy to implement.

Smart Contracts in Hyperledger Indy

Hyperledger Indy does not host smart contracts. Rather than storing data in the ledger and then providing access to that data using smart contracts, Indy enables users to own the data and share it in a way that preserves their privacy.

Hyperledger Indy identities can be referenced in smart contracts from other systems. This allows Indy to provide any distributed ledger system with a first-class decentralized identity system.

Hyperledger Indy does support plugins which enable the support of new transactions without touching the core components of the codebase. An example would be building a simple crypto-currency on Indy.

Smart Contracts in Hyperledger Iroha

At the time of publication, the Hyperledger Iroha documentation was being updated to reflect the latest smart contract functionality. When completed, this description will be available at **hyperledger.org/projects/iroha**.

Smart Contracts in Hyperledger Sawtooth

Hyperledger Sawtooth is a distributed ledger project dedicated to making smart contracts safe, particularly for enterprise use. It supports both types of smart contracts: installed and on-chain. Developers can choose from seven languages to develop smart contracts in Sawtooth.

Installed Smart Contracts with Transaction Families

Any fully programmable language present certain risks. To limit these risks, other blockchain networks will specify fixed transaction semantics. These networks can use families of transactions that support only certain allowed operations. In this context, you can think of a transaction family as a distributed application.

A simple example is the **IntegerKey transaction family**. This provides just three operations: increment, decrement, and set. With just three operations and no looping, this family helps prevents any intentional or accidental transaction script problems.

Another example is the **Settings transaction family**, which can be used to control the blockchain network itself, including specifics like the consensus being used and the inner block time.

A sophisticated example that blocks arbitrary syntax is the **supply chain transaction family**. The semantics of that family include about 20 operations required to trace the provenance and other contextual information of any asset, but no further operations that could be misused, either on purpose or by accident.

Any transaction family can be deployed with Hyperledger Sawtooth, so long as this family supports the transaction family API. This is a simple API that supports a few operations like `get state` to fetch something from the ledger and `set state` to set something in the ledger.

Transaction families can be written in just about any language, including C++, Go, Java, JavaScript, Python, Rust and Solidity via Seth. Existing transaction families that support the API include **Blockinfo**, **Identity**, **IntegerKey**, **Settings**, **Smallbank**, **Validator registry**, and **XO**.

The motivation behind transaction families is to enable businesses to pick the level of versatility and risk that's right for their network. One benefit of the transaction family concept—compared to on-chain transactions—is that a super-critical bug can only kill the transaction family process. The rest of the validator and all the other transaction families can keep running. In contrast, a super-critical bug running on-chain could kill the entire node.

On-Chain Smart Contracts with the Seth Transaction Family

On-chain smart contracts are handled by plugging the Hyperledger Burrow Ethereum Virtual Machine (EVM) onto the Hyperledger Sawtooth validator node. Once this is in place, smart contracts can be written with Solidity code on Hyperledger Sawtooth using the Sawtooth-Ethereum (Seth) transaction family and transaction processor. This allows for fully programmable smart contracts.

The `seth` command can be used to interact with the Seth transaction family via a command line interface. This enables loading and executing smart contracts, querying the data associated with a contract, and generating keys in the format used by seth.

Contracts are compiled in Hyperledger Sawtooth with the `seth load` command, which has an `–init` flag that takes a hex-encoded byte array as an argument. This string is interpreted as the contract creation code. The `solc` compiler can be used to generate this string for a Solidity smart contract.

Contracts in Solidity, which can be compiled in Hyperledger Sawtooth, are similar to classes in object-oriented languages. They contain persistent data in state variables, and functions that can modify these variables. Calling a function on a different contract performs an EVM function call and switches the context so that state variables are inaccessible.

Creating contracts is best done using the JavaScript API web.js. When a contract is created, a function with the same name as the contract called a "constructor" may be executed once. The constructor is optional and only one constructor is allowed, so that overloading is not supported.

If one contract wants to create another contract, the source code (and the binary) of the newly created contract must be known to the creator. This means that cyclic creation dependencies are not possible.

Conclusion

This paper explores the modular architecture used by all Hyperledger projects and looked at the different ways smart contracts can be implemented within this modular framework.

Key takeaways include:

1. The Architecture WG will continue to define the following core components for permissioned blockchain networks: Consensus Layer, Smart Contract Layer, Communication Layer, Data Store Abstraction, Crypto Abstraction, Identity Services, Policy Services, APIs, and Interoperation.

2. The Architecture WG has shared a generalized reference architecture for smart contracts that can be used by any Hyperledger project.

3. Hyperledger Burrow, Hyperledger Fabric, Hyperledger Iroha, and Hyperledger Sawtooth each manifest the reference architecture principles in unique ways. Table 1 compares how smart contracts are used within each of these frameworks.

Forthcoming papers in this series will expand the Hyperledger generalized reference architecture to cover all the core components for permissioned blockchain networks. The next paper in the series will cover Identity Services.

## Permissioned Blockchain Networks

Blockchain requirements vary across different business and industries. Some uses require rapid network consensus systems and short block confirmation times before being added to the chain. For others, a slower processing time may be acceptable in exchange for lower levels of required trust.

Confidentiality, compliance, scalability, workflow, and security requirements differ drastically across industries and uses. Each of these requirements, and many others, represent a potentially unique optimization point for the technology.

For these reasons, Hyperledger incubates and promotes a range of business blockchain technologies including distributed ledgers, smart contract engines, client libraries, graphical interfaces, utility libraries, and sample applications.

Hyperledger Umbrella Strategy

The Hyperledger umbrella strategy encourages re-use of common building blocks within a modular architecture. This enables rapid innovation of distributed ledger technology (DLT), common functional modules, and the interfaces between them. The benefits of this modular approach include extensibility and flexibility, so that any component can be modified independently without affecting the rest of the system.
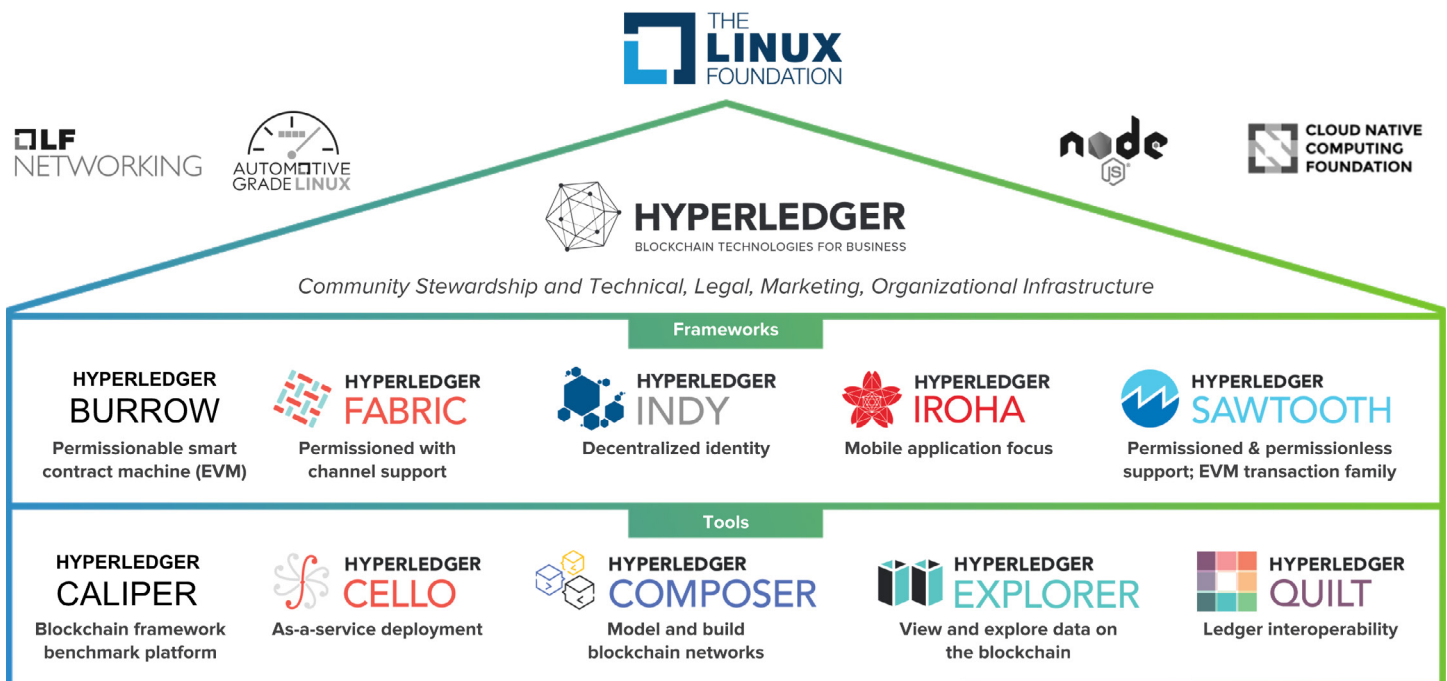
About the Hyperledger Architecture Working Group

The Hyperledger Architecture WG serves as a cross-project forum for architects and technologists from the Hyperledger community to exchange ideas and explore alternate architectural options and tradeoffs.

Our focus is on developing a modular architectural framework for enterprise-class distributed ledgers. This includes identifying common and critical components, providing a functional decomposition of an enterprise blockchain stack into component layers and modules, standardizing interfaces between components, and ensuring interoperability between ledgers.

To learn more about this working group and find out how to participate, visit:
wiki.hyperledger.org/groups/architecture/architecture-wg