**HYPERLEDGER**
BLOCKCHAIN TECHNOLOGIES FOR BUSINESS

# Hyperledger Architecture, Volume 1

Introduction to Hyperledger Business Blockchain Design Philosophy and **Consensus**

*This is the first 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.*

In this initial paper, we aim to:

1. Outline the overarching Hyperledger design philosophy for permissioned blockchain networks.

2. Explain how our approach optimizes the development of flexible, interoperable enterprise blockchain technologies.

3. Identify the core permissioned blockchain network components that the Architecture WG has been and will continue to define through its work.

4. Provide a generalized reference architecture for consensus.

5. Explore how each Hyperledger business blockchain framework manifests the reference architecture.

Forthcoming papers in this series will expand the Hyperledger generalized reference architecture to include the following business blockchain components: Smart Contract Layer, 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.

**HYPERLEDGER**
BLOCKCHAIN TECHNOLOGIES FOR BUSINESS

**HYPERLEDGER MODULAR UMBRELLA APPROACH**

### Infrastructure
**Technical, Legal, Marketing, Organizational**

Ecosystems that accelerate open development and commercial adoption

| Cloud Foundry | Node.js | **Hyperledger** | Open Container Initiative |
|---|---|---|---|

### Frameworks
Meaningfully differentiated approaches to business blockchain frameworks developed by a growing community of communities

| Hyperledger **Indy** | Hyperledger **Fabric** | Hyperledger **Iroha** | Hyperledger **Sawtooth** | Hyperledger **Burrow** |
|---|---|---|---|---|

### Tools
Typically built for one framework, and through common license and community of communities approach, ported to other frameworks

| | Hyperledger **Composer** | Hyperledger **Explorer** | Hyperledger **Cello** | |
|---|---|---|---|---|

## 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. Their 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 the components, and ensuring interoperability between ledgers.

## Introduction

Business blockchain requirements vary. 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. Scalability, confidentiality, compliance, workflow complexity, and even 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's umbrella strategy encourages the re-use of common building blocks via a modular architectural framework. 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, flexibility, and the ability for any component to be modified independently without affecting the rest of the system.

## Architecture Overview

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 the development of a rich and easy-to-use Application Programming Interface (API). 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.

- **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 policy 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.

In this document, we will explore consensus. The goal of consensus is to generate an agreement on the order and to validate the correctness of the set of transactions that constitute the block.

## Consensus

Consensus is the process by which a network of nodes provides a guaranteed ordering of transactions and validates the block of transactions. Consensus must provide the following core functionality:

- Confirms the correctness of all transactions in a proposed block, according to endorsement and consensus policies.

- Agrees on order and correctness and hence on results of execution (implies agreement on global state).

- Interfaces and depends on smart-contract layer to verify correctness of an ordered set of transactions in a block.

## Comparison of Consensus Types

Consensus may be implemented in different ways such as through the use of lottery-based algorithms including Proof of Elapsed Time (PoET) and Proof of Work (PoW) or through the use of voting-based methods including Redundant Byzantine Fault Tolerance (RBFT) and Paxos. Each of these approaches targets different network requirements and fault tolerance models.

The lottery-based algorithms are advantageous in that they can scale to a large number of nodes since the winner of the lottery proposes a block and transmits it to the rest of the network for validation. On the other hand, these algorithms may lead to forking when two "winners" propose a block. Each fork must be resolved, which results in a longer time to finality.

The voting-based algorithms are advantageous in that they provide low-latency finality. When a majority of nodes validates a transaction or block, consensus exists and finality occurs. Because voting-based algorithms typically require nodes to transfer messages to each of the other nodes on the network, the more nodes that exist on the network, the more time it takes to reach consensus. This results in a trade-off between scalability and speed.

The operating assumption for Hyperledger developers is that business blockchain networks will operate in an environment of partial trust. Given this, we are expressly not including standard Proof of Work consensus approaches with anonymous miners. In our assessment, these approaches impose too great a cost in terms of resources and time to be optimal for business blockchain networks.

Table 1 offers an at-a-glance view of the main considerations and pros and cons of different business blockchain approaches to reaching consensus.

**TABLE 1. COMPARISON OF PERMISSIONED CONSENSUS APPROACHES AND STANDARD PoW**

| | Permissioned Lottery-based | Permissioned Voting-based | Standard Proof of Work (Bitcoin) |
|---|---|---|---|
| Speed | GOOD | GOOD | POOR |
| Scalability | GOOD | MODERATE | GOOD |
| Finality | MODERATE | GOOD | POOR |

**FIGURE 1. GENERALIZED HYPERLEDGER CONSENSUS PROCESS FLOW**
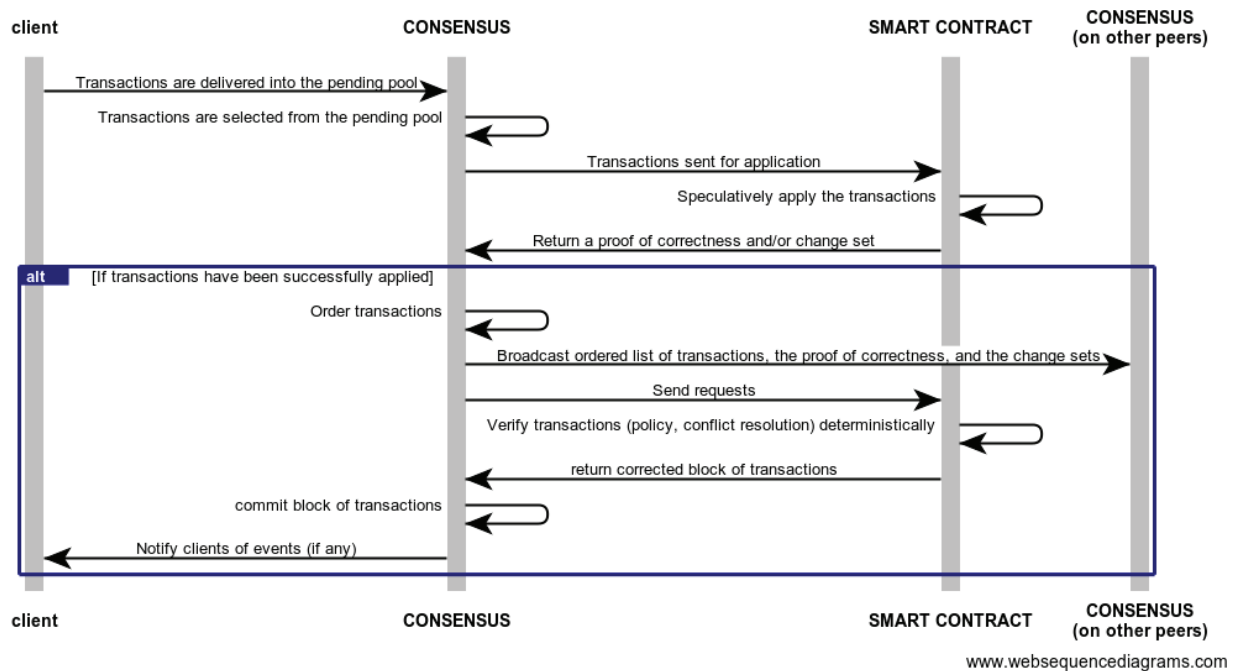


## Consensus and Its Interaction with Other Architectural Layers

While there are many ways in which consensus can be achieved, our analysis of blockchain platforms suggests that the process shown in Figure 1 is commonly used. This is a generalized view, and the different Hyperledger frameworks may choose to implement these steps differently.

Hyperledger business blockchain frameworks reach consensus by performing two separate activities:

1. Ordering of transactions

2. Validating transactions

By logically separating these activities, we ensure that any Hyperledger framework can work with any Hyperledger consensus module.

The first step of the consensus process flow is receiving the transactions from the client application. Consensus depends on an ordering service to order transactions. The ordering service can be implemented in different ways: ranging from a centralized service, which can be used in development and testing, to distributed protocols that target different network and node fault models. To enable confidentiality of the transactions, the ordering service may be agnostic to the transaction; that is, the transaction content can be hashed or encrypted.

Transactions are submitted via an interface to the ordering service. This service collects transactions based on the consensus algorithm and configuration policy, which may define a time limit or specify the number of transactions allowed. Most of the time, for efficiency reasons, instead of outputting individual transactions, the ordering service will group multiple transactions into a single block. In this case, the ordering service must impose and convey a deterministic ordering of the transactions within each block.

To validate transactions, consensus depends on the smart contract layer because it contains the business logic behind what makes a transaction valid. The smart contract layer validates each transaction by ensuring they conform to policy and the contract specified for the transaction. Invalid transactions are rejected and possibly dropped from inclusion within a block.

We can divide potential validation errors into two categories: syntax and logic errors. For syntax errors such as invalid inputs, unverifiable signature, and repeated transaction (due to error or replay attacks), the transaction should be dropped. The second category of errors is more complex and should be policy driven whether to continue processing or not. For example, a transaction that would result in double-spend or versioning control failure. We might want to log these transactions for auditing if the policy requires.

The consensus layer uses the communication layer for communicating with the client and other peers on the network.

## Consensus Properties

Consensus must satisfy two properties to guarantee agreement among nodes: *safety* and *liveness.*

Safety means that each node is guaranteed the same sequence of inputs and results in the same output on each node. When the nodes receive an identical series of transactions, the same state changes will occur on each node. The algorithm must behave identical to a single node system that executes each transaction atomically one at a time.

Liveness means that each non-faulty node will eventually receive every submitted transaction, assuming that communication does not fail.

## Consensus in the Hyperledger Frameworks

Because business blockchain requirements will vary, the Hyperledger community is working on several different consensus mechanisms as well as implementation approaches to ensure modularity.

Table 2 provides a comparison of the consensus algorithms used across Hyperledger frameworks. Apache Kafka in Hyperledger Fabric, RBFT in Hyperledger Indy, and Sumeragi in Hyperledger Iroha use a voting-based approach to consensus that provides fault tolerance and finality within seconds. PoET in Hyperledger Sawtooth uses a lottery-based approach to consensus that provides scale at the cost of finality being delayed due to forks that must be resolved.

**TABLE 2. COMPARISON OF CONSENSUS ALGORITHMS USED IN HYPERLEDGER FRAMEWORKS**

| Consensus Algorithm | Consensus Approach | Pros | Cons |
|---|---|---|---|
| **Kafka in Hyperledger Fabric Ordering Service** | Permissioned voting-based. Leader does ordering. Only in-sync replicas can be voted as leader. ("Kafka," 2017). | Provides crash fault tolerance. Finality happens in a matter of seconds. | While Kafka is crash fault tolerant, it is not Byzantine fault tolerant, which prevents the system from reaching agreement in the case of malicious or faulty nodes. |
| **RBFT in Hyperledger Indy** | Pluggable election strategy set to a permissioned, voting-based strategy by default ("Plenum," 2016). All instances do ordering, but only the requests ordered by the master instance are actually executed. (Aublin, Mokhtar & Quéma, 2013) | Provides Byzantine fault tolerance. Finality happens in a matter of seconds. | The more nodes that exist on the network, the more time it takes to reach consensus. The nodes in the network are known and must be totally connected. |
| **Sumeragi in Hyperledger Iroha** | Permissioned server reputation system. | Provides Byzantine fault tolerance. Finality happens in a matter of seconds. Scale to petabytes of data, distributed across many clusters (Struckhoff, 2016). | The more nodes that exist on the network, the more time it takes to reach consensus. The nodes in the network are known and must be totally connected. |
| **PoET in Hyperledger Sawtooth** | Pluggable election strategy set to a permissioned, lottery-based strategy by default. | Provides scalability and Byzantine fault tolerance. | Finality can be delayed due to forks that must be resolved. |

## Consensus in Hyperledger Fabric

Consensus in Hyperledger Fabric is broken out into 3 phases:  *Endorsement, Ordering,* and *Validation*.

- Endorsement is driven by policy (eg m out of n signatures) upon which participants endorse a transaction.

- Ordering phase accepts the endorsed transactions and agrees to the order to be committed to the ledger.

- Validation takes a block of ordered transactions and validates the correctness of the results, including checking endorsement policy and double-spending.

**FIGURE 2. ILLUSTRATION OF ONE POSSIBLE TRANSACTION FLOW (COMMON-CASE PATH) IN HYPERLEDGER FABRIC**
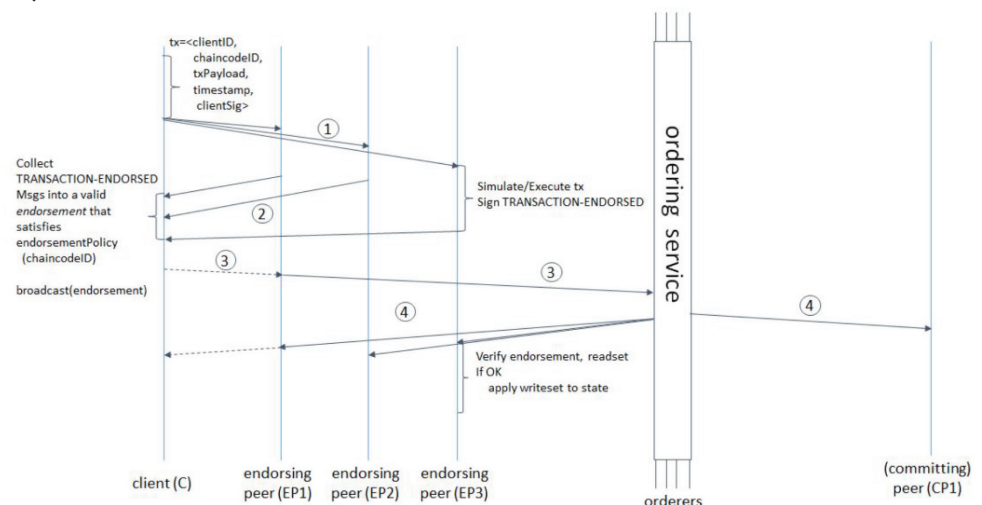


Hyperledger Fabric supports pluggable consensus service for all 3 phases. Applications may plugin different endorsement, ordering, and validation models depending on their requirements. In particular, the ordering service API allows plugging in BFT-based agreement algorithms. The ordering service API consists of two basic operations: broadcast and deliver.

- **broadcast(blob):** a client calls this to broadcast an arbitrary message blob for dissemination over the channel. This is also called request(blob) in the BFT context, when sending a request to a service.

- **deliver(seqno, prevhash, blob):** the ordering service calls this on the peer to deliver the message blob with the specified non-negative integer sequence number (seqno) and hash of the most recently delivered blob (prevhash). In other words, it is an output event from the ordering service. deliver() is also sometimes called notify() in pub-sub systems or commit() in BFT systems.

Multiple ordering plugins are being developed currently, including BFT Smart, Simplified Byzantine Fault Tolerance (SBFT), Honey Badger of BFT, etc. For Fabric v1, Apache Kafka is provided out-of-the-box as a reference implementation. The application use-cases and its fault tolerance model should determine which plugin to use.

## Consensus in Hyperledger Indy

Consensus in Hyperledger Indy is based on Redundant Byzantine Fault Tolerance (RBFT), which is a protocol inspired by Plenum Byzantine Fault Tolerance (Plenum). Think of RBFT as running several instances of Plenum in parallel. Ordered requests from a single instance called the master is used to update the ledger, but the master's performance in terms of throughput and latency is periodically compared to the average performance of other instances. If the master is found to be degraded, a view change occurs that appoints a different instance to the role of master. Like PBFT, RBFT needs at least 3f+1 nodes to handle f faulty nodes. Figure 3 shows a network of 4 nodes that can handle 1 faulty node, has 2 PBFT-like instances running, one master and one backup. Each node can host one primary (leader) instance.

**FIGURE 3. RBFT OVERVIEW**
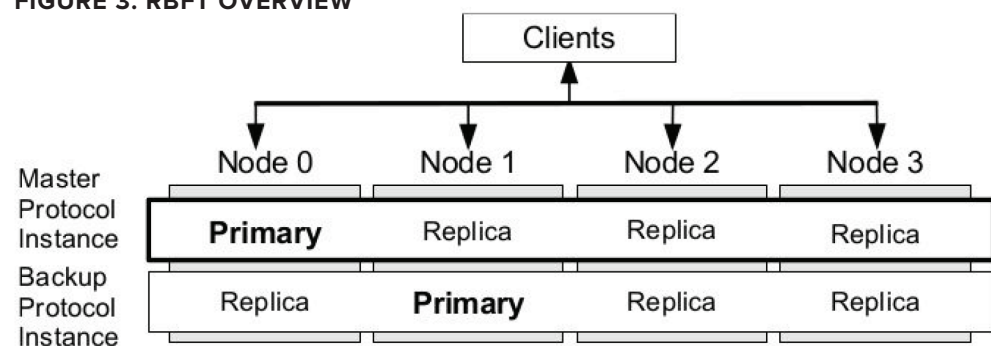


Source: (Aublin, Mokhtar & Quéma, 2013)

Figure 4 shows a different view of RBFT. Here the client is sending a request to nodes. It does not have to send to all nodes because sending to f+1 nodes is sufficient. After receiving the client request, the nodes do a dissemination process through PROPAGATE in which every other node is made aware of the request. Each primary creates a proposal from the received requests called a PRE-PREPARE and sends it to all other nodes. If the nodes accept the primary's proposal, they send an acknowledgement to the proposal by a message called PREPARE. Once a node gets a PRE-PREPARE proposal and 2f PREPARE messages, then it has sufficient information to accept the proposal and sends a COMMIT message. Once a node gets 2f+1 COMMIT messages, then the batch of requests can be ordered and added to the ledger since a sufficient number of nodes have agreed that a majority of nodes have accepted the proposal. The primary does not require one proposal to complete before it can send the next proposal.

**FIGURE 4. RBFT PROTOCOL STEPS**
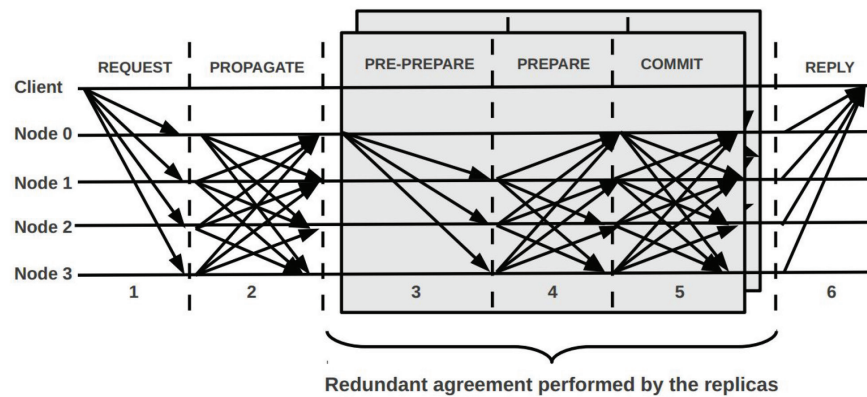


Source: (Aublin et al., 2013)

Hyperledger Indy uses RBFT to handle ordering and validation, which results in a single ledger containing both ordered and validated transactions. This is unlike many blockchain networks that use a Byzantine Fault Tolerance (BFT) protocol only for ordering. These networks leave domain-specific validation to happen after requests are ordered.

Plenum, and therefore RBFT, maintains a projection of the ledger called state. All valid, accepted operations performed may change the state, which is stored in a database as a collection of variables and their values. The state is kept in a cryptographically authenticated data structure called a Merkle Patricia tree, which is specified by Ethereum. Hyperledger Indy stores Decentralized Identifiers (DIDs) as state variables with values including the current verification key and a few other things. The in memory copy of the ledger transactions and resulting state are optimistically updated during the proposal phase. The primary is updated while sending the proposal and non-primaries are updated while accepting a valid proposal.  The proposal gets ordered, then the ledger and the resulting state are committed. If the proposal gets rejected for some reason, then the changes made to the ledger and the resulting state are reverted.

This optimistic update is necessary to have multiple proposals being proposed without the previous proposals being completely ordered. For example, if the first proposal contained a request which was creating a new identity Id1 on the ledger and the second proposal contained a request adding an attribute for Id1, then if the state did not reflect the existence of Id1, the second proposal would be rejected. So, before the second proposal is accepted, the changes in the first proposal should be visible. When changes have been proposed, but not yet committed, it's called an uncommitted change.

# Consensus in Hyperledger Iroha

Hyperledger Iroha introduces a BFT consensus algorithm called Sumeragi, which tolerates f numbers of Byzantine faulty nodes in a network, like all BFT systems. It is heavily inspired by the B-Chain Algorithm described by Duan, Meling, Peisert, & Zhang (2014).

As in B-Chain, we consider the concept of a global order over validating peers and sets A and B of peers, where A consists of the first 2f+1 peers and B consists of the remainder. As 2f+1 signatures are needed to confirm a transaction, under the normal case only 2f+1 peers are involved in transaction validation. The remaining peers only join the validation when faults are exhibited in peers in set A. The 2f+1th peer is called the proxy tail. For normal (non-failure) cases, the transaction flow is shown as in Figure 5.

**FIGURE 5. NORMAL TRANSACTION FLOW IN HYPERLEDGER IROHA WITH SUMERAGI**
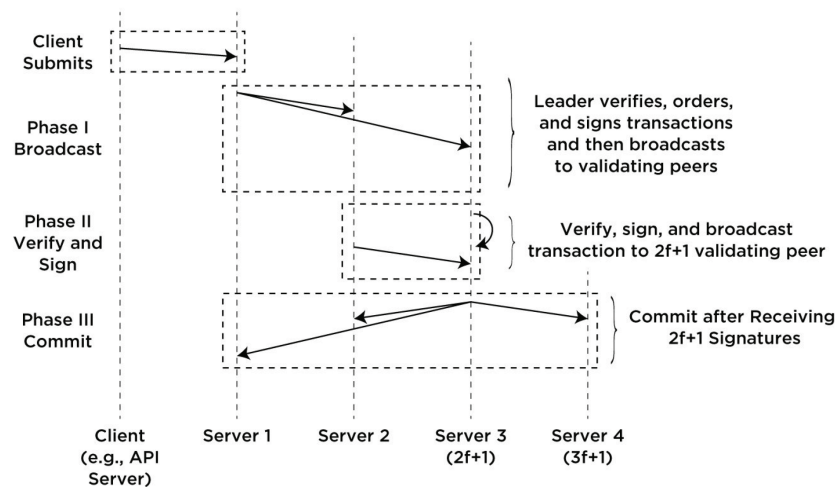


The client, which will typically be an API server interfacing with an end-user client, first submits a transaction to the lead validating peer. This leader then verifies the transaction, orders it into the queue, and signs the transaction. It then broadcasts this transaction to the remaining 2f+1 validating peers.

The order of processing nodes is determined based on the server reputation system called hijiri. Hijiri calculates the reliability of servers based on three factors. First, based on the time each server registered with the membership service. Second, based on the number of successful transactions processed by each server. Third, based on whether failures have been detected.

To detect failures, each server sets a timer when it signs and broadcasts a transaction to the proxy tail. If there is a failure in an intermediate server and a reply is not received before the timer goes off, then the server rebroadcasts the transaction and its signature to the next server in the chain after the proxy tail. The case of a failure in the proxy tail is shown in Figure 6.

**FIGURE 6. TRANSACTION FLOW IN THE CASE OF A SERVER FAILURE IN HYPERLEDGER IROHA WITH SUMERAGI**
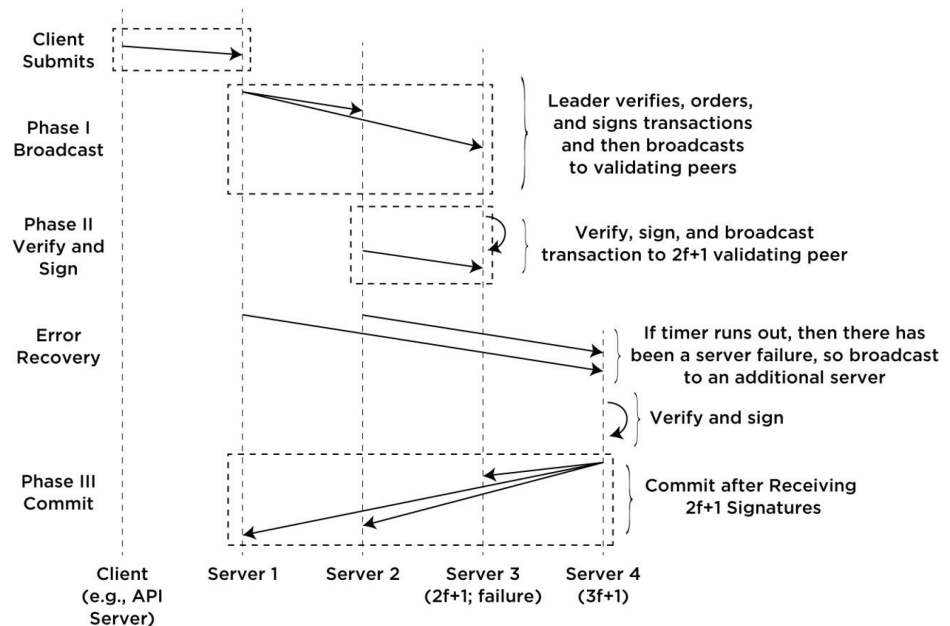


Consensus in Sumeragi is performed on individual transactions and on the global state resulting from the application of the transaction. When a validating peer receives a transaction over the network, it performs the following steps in order:

1. Validate the signature (or signatures, in the case of multi-signature transactions) of the transaction.

2. Validate the contents of the transaction, where applicable. For example, transfer transactions must leave the payer's account with a non-negative balance.

3. Temporarily apply the transaction to the ledger. This involves updating the Merkle root of the global state.

4. Sign the updated Merkle root and the hash of the transaction contents.

5. Broadcast the tuple, which is a finite ordered list of transactions.

6. When syncing nodes with each other, valid parts of the Merkle tree are shared until the roots match.

## Consensus in Hyperledger Sawtooth

Hyperledger Sawtooth facilitates pluggable consensus for both lottery and voting algorithms. By default, Hyperledger Sawtooth uses a lottery-based, Nakamoto consensus algorithm called PoET. As with Bitcoin's PoW, PoET uses a lottery for leader election based on a guaranteed wait time provided through a Trusted Execution Environment (TEE) (Sandell, Bowman, & Shah, 2016). For the purpose of achieving distributed consensus efficiently, a good lottery function has several characteristics that have been defined by Sandell et al. (2016) as:

- **Fairness** - The function should distribute leader election across the broadest possible population of participants.

- **Investment** - The cost of controlling the leader election process should be proportional to the value gained from it.

- **Verification** - It should be relatively simple for all participants to verify that the leader was legitimately selected.

The current implementation of Hyperledger Sawtooth builds on a TEE provided by Intel's Software Guard Extensions (SGX). This ensures the safety and randomness of the leader election process without requiring the costly investment of power and specialized hardware inherent in most "proof" algorithms.

Every PoET validator requests a random time to wait from a trusted function before claiming leadership. The validator with the shortest wait time for a particular transaction block is implicitly elected the leader. The function "CreateTimer" creates a timer for a transaction block that is guaranteed to have been created by the TEE. The "CheckTimer" function verifies that the timer was created by the TEE and, if it has expired, creates an attestation that can be used to verify that validator did, in fact, wait the allotted time before claiming the leadership role.

The PoET leader election algorithm meets the criteria for a good lottery algorithm. It randomly distributes leadership election across the entire population of validators with a distribution that is similar to what is provided by other lottery algorithms. The probability of election is proportional to the resources contributed, where resources are general purpose processors with a TEE. An attestation of execution provides information for verifying that the certificate was created within the TEE and that the validator waited the allotted time. Further, the low cost of participation increases the likelihood that the population of validators will be large, increasing the robustness of the consensus algorithm.

## Conclusion

In this initial paper, we introduced the modular architectural framework used by all Hyperledger projects and explored multiple ways that consensus can be implemented within this modular framework.

**Key takeaways include:**

1. The overarching Hyperledger design philosophy for permissioned blockchain networks follows a modular approach that enables extensibility and flexibility.

2. Within this modular approach, Hyperledger defines common functional components and the interfaces between them, which allows any component to be modified independently without affecting the rest of the system.

3. The Architecture WG has been and 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.

4. The Architecture WG has shared a generalized reference architecture for consensus that can be used by any Hyperledger project.

5. Hyperledger Fabric, Hyperledger Indy, Hyperledger Iroha, and Hyperledger Sawtooth each manifest the reference architecture principles in unique ways. The comparison of consensus algorithms commonly used with these frameworks is provided in Table 1, which can be used to quickly determine the strengths and weaknesses of algorithms including Kafka, RBFT, Sumeragi, and PoET.

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 the Smart Contract Layer. If you are interested in participating in Hyperledger's Architecture working group, please visit the Wiki page for more information.

# References

**Aublin, P., Mokhtar, S.B., & Quéma, V.** (2013). RBFT: Redundant Byzantine Fault Tolerance. Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on, pp. 297-306. doi:10.1109/ICDCS.2013.53 or http://dx.doi.org/10.1109/ICDCS.2013.53

**Cachin, C., & Vukolić, M.** (July 7, 2017). Blockchain Consensus Protocols in the Wild. IBM Research - Zurich. arXiv:1707.01873 or https://arxiv.org/abs/1707.01873v2

**Duan S., Meling H., Peisert S., & Zhang H.** (2014). BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration. In: Aguilera M.K., Querzoni L., & Shapiro M. (eds) Principles of Distributed Systems. OPODIS 2014. Lecture Notes in Computer Science, vol 8878, pp. 91-106. Springer, Cham. doi:10.1007/978-3-319-14472-6_7 or http://dx.doi.org/10.1007/978-3-319-14472-6_7

**Kafka 0.11.0 Documentation.** (2017). Retrieved from https://kafka.apache.org/documentation/

**Plenum Byzantine Fault Tolerant Protocol.** (2016). Retrieved from https://github.com/hyperledger/indy-plenum/wiki

**Sandell, P., Bowman, M., & Shah, P.** (2016). Blockchain and Its Emerging Role in Healthcare Related Research. Intel Corporation – Health and Life Sciences Group. https://oncprojecttracking.healthit.gov/wiki/download/attachments/14582699/19-Blockchain_Whitepaper_Intel_Final.pdf

**Struckhoff, Roger. The Iroha Project to Bring Mobility to Blockchain with Simple APIs.** (December 22, 2016). Retrieved from https://www.altoros.com/blog/the-iroha-project-to-bring-mobility-to-blockchain-with-simple-apis/

**Iroha Whitepaper.** (March 7, 2017). Retrieved from https://github.com/hyperledger/iroha/blob/master/docs/iroha_whitepaper.md