# Architecture Reference

* [Architecture Origins](https://hyperledger-fabric.readthedocs.io/en/release-1.4/arch-deep-dive.html)
* [Transaction Flow](https://hyperledger-fabric.readthedocs.io/en/release-1.4/txflow.html)
* [Hyperledger Fabric CA's User Guide](http://hyperledger-fabric-ca.readthedocs.io/en/latest)
* [Hyperledger Fabric SDKs](https://hyperledger-fabric.readthedocs.io/en/release-1.4/fabric-sdks.html)
* [Service Discovery](https://hyperledger-fabric.readthedocs.io/en/release-1.4/discovery-overview.html)
* [Channels](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channels.html)
* [CouchDB as the State Database](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html)
* [Peer channel-based event services](https://hyperledger-fabric.readthedocs.io/en/release-1.4/peer_event_services.html)
* [Private Data](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data-arch.html)
* [Read-Write set semantics](https://hyperledger-fabric.readthedocs.io/en/release-1.4/readwrite.html)
* [Gossip data dissemination protocol](https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html)

# Architecture Origins

**Note**

This [document](https://hyperledger-fabric.readthedocs.io/en/release-1.4/arch-deep-dive.html) represents the initial architectural proposal for Hyperledger Fabric v1.0. While the Hyperledger Fabric implementation has conceptually followed from the architectural proposal, some [details](https://hyperledger-fabric.readthedocs.io/en/release-1.4/arch-deep-dive.html) have been altered during the implementation. The initial architectural proposal is presented as originally prepared. For a more technically accurate representation of the architecture, please see [Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains](https://arxiv.org/abs/1801.10228v2).

The Hyperledger Fabric architecture delivers the following advantages:

* **Chaincode trust flexibility.** The architecture separates trust assumptions for chaincodes (blockchain applications) from trust assumptions for ordering. In other words, the ordering service may be provided by one set of nodes (orderers) and tolerate some of them to fail or misbehave, and the endorsers may be different for each chaincode.
* **Scalability.** As the endorser nodes responsible for particular chaincode are orthogonal to the orderers, the system may scale better than if these functions were done by the same nodes. In particular, this results when different chaincodes specify disjoint endorsers, which introduces a partitioning of chaincodes between endorsers and allows parallel chaincode execution (endorsement). Besides, chaincode execution, which can potentially be costly, is removed from the critical path of the ordering service.
* **Confidentiality.** The architecture facilitates deployment of chaincodes that have confidentiality requirements with respect to the content and state updates of its transactions.
* **Consensus modularity.** The architecture is modular and allows pluggable consensus (i.e., ordering service) implementations.

**Part I: Elements of the architecture relevant to Hyperledger Fabric v1**

1. System architecture
2. Basic workflow of transaction endorsement
3. Endorsement policies

**Part II: Post-v1 elements of the architecture**

1. Ledger checkpointing (pruning)

## 1. System architecture

The blockchain is a distributed system consisting of many nodes that communicate with each other. The blockchain runs programs called chaincode, holds state and ledger data, and executes transactions. The chaincode is the central element as transactions are operations invoked on the chaincode. Transactions have to be “endorsed” and only endorsed transactions may be committed and have an effect on the state. There may exist one or more special chaincodes for management functions and parameters, collectively called system chaincodes.

### 1.1. Transactions

Transactions may be of two types:

* Deploy transactions [create](https://hyperledger-fabric.readthedocs.io/en/release-1.4/arch-deep-dive.html) new chaincode and take a program as parameter. When a deploy transaction executes successfully, the chaincode has been installed “on” the blockchain.
* Invoke transactions perform an operation in the context of previously deployed chaincode. An invoke transaction refers to a chaincode and to one of its provided functions. When successful, the chaincode executes the specified function - which may involve modifying the corresponding state, and returning an output.

As described later, deploy transactions are special cases of invoke transactions, where a deploy transaction that creates new chaincode, corresponds to an invoke transaction on a system chaincode.

**Remark:** This document currently assumes that a transaction either creates new chaincode or invokes an operation provided by \*one already deployed chaincode. This document does not yet describe: a) optimizations for query (read-only) transactions (included in v1), b) support for cross-chaincode transactions (post-v1 feature).\*

### 1.2. Blockchain datastructures

#### 1.2.1. State

The latest state of the blockchain (or, simply, state) is modeled as a versioned key-value store (KVS), where keys are names and values are arbitrary blobs. These entries are manipulated by the chaincodes (applications) running on the blockchain through put and get KVS-operations. The state is stored persistently and updates to the state are logged. Notice that versioned KVS is adopted as state model, an implementation may use actual KVSs, but also RDBMSs or any other solution.

More formally, state s is modeled as an element of a mapping K -> (V X N), where:

* K is a set of keys
* V is a set of values
* N is an infinite ordered set of version numbers. Injective function next: N -> N takes an element of N and returns the next version number.

Both V and N contain a special element ⊥ (empty type), which is in case of N the lowest element. Initially all keys are mapped to (⊥, ⊥). For s(k)=(v,ver) we denote v by s(k).value, and ver by s(k).version.

KVS operations are modeled as follows:

* put(k,v) for k ∈ K and v ∈ V, takes the blockchain state s and changes it to s' such that s'(k)=(v,next(s(k).version)) with s'(k')=s(k') for all k'!=k.
* get(k) returns s(k).

State is maintained by peers, but not by orderers and clients.

**State partitioning.** Keys in the KVS can be recognized from their name to belong to a particular chaincode, in the sense that only transaction of a certain chaincode may modify the keys belonging to this chaincode. In principle, any chaincode can read the keys belonging to other chaincodes. Support for cross-chaincode transactions, that modify the state belonging to two or more chaincodes is a post-v1 feature.

#### 1.2.2 Ledger

Ledger provides a verifiable history of all successful state changes (we talk about valid transactions) and unsuccessful attempts to change state (we talk about invalid transactions), occurring during the operation of the system.

Ledger is constructed by the ordering service (see Sec 1.3.3) as a totally ordered hashchain of blocks of (valid or invalid) transactions. The hashchain imposes the total [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/arch-deep-dive.html) of blocks in a ledger and each block contains an array of totally ordered transactions. This imposes total order across all transactions.

Ledger is kept at all peers and, optionally, at a subset of orderers. In the context of an orderer we refer to the Ledger as to OrdererLedger, whereas in the context of a peer we refer to the ledger as to PeerLedger. PeerLedger differs from the OrdererLedger in that peers locally maintain a bitmask that tells apart valid transactions from invalid ones (see Section XX for more details).

Peers may prune PeerLedger as described in Section XX (post-v1 feature). Orderers maintain OrdererLedger for fault-tolerance and availability (of the PeerLedger) and may decide to prune it at anytime, provided that properties of the ordering service (see Sec. 1.3.3) are maintained.

The ledger allows peers to replay the history of all transactions and to reconstruct the state. Therefore, state as described in Sec 1.2.1 is an optional datastructure.

### 1.3. Nodes

Nodes are the communication entities of the blockchain. A “node” is only a logical function in the sense that multiple nodes of different types can run on the same physical server. What counts is how nodes are grouped in “trust domains” and associated to logical entities that control them.

There are three types of nodes:

1. **Client** or **submitting-client**: a client that submits an actual transaction-invocation to the endorsers, and broadcasts transaction-proposals to the ordering service.
2. **Peer**: a node that commits transactions and maintains the state and a copy of the ledger (see Sec, 1.2). Besides, peers can have a special **endorser** role.
3. **Ordering-service-node** or **orderer**: a node running the communication service that implements a delivery guarantee, such as atomic or total order broadcast.

The types of nodes are explained next in more detail.

#### 1.3.1. Client

The client represents the entity that acts on behalf of an end-user. It must connect to a peer for communicating with the blockchain. The client may connect to any peer of its choice. Clients create and thereby invoke transactions.

As detailed in Section 2, clients communicate with both peers and the ordering service.

#### 1.3.2. Peer

A peer receives ordered state updates in the form of blocks from the ordering service and maintain the state and the ledger.

Peers can additionally take up a special role of an **endorsing peer**, or an **endorser**. The special function of an endorsing peer occurs with respect to a particular chaincode and consists in endorsing a transaction before it is committed. Every chaincode may specify an endorsement policy that may refer to a set of endorsing peers. The policy defines the necessary and sufficient conditions for a valid transaction endorsement (typically a set of endorsers’ signatures), as described later in Sections 2 and 3. In the special case of deploy transactions that install new chaincode the (deployment) endorsement policy is specified as an endorsement policy of the system chaincode.

#### 1.3.3. Ordering service nodes (Orderers)

The orderers form the ordering service, i.e., a communication fabric that provides delivery guarantees. The ordering service can be implemented in different ways: ranging from a centralized service (used e.g., in development and testing) to distributed protocols that target different network and node fault models.

Ordering service provides a shared communication channel to clients and peers, offering a broadcast service for messages containing transactions. Clients connect to the channel and may broadcast messages on the channel which are then delivered to all peers. The channel supports atomic delivery of all messages, that is, message communication with total-order delivery and (implementation specific) reliability. In other words, the channel outputs the same messages to all connected peers and outputs them to all peers in the same logical order. This atomic communication guarantee is also called total-order broadcast, atomic broadcast, or consensus in the context of distributed systems. The communicated messages are the candidate transactions for inclusion in the blockchain state.

**Partitioning (ordering service channels).** Ordering service may support multiple channels similar to the topics of a publish/subscribe (pub/sub) messaging system. Clients can connect to a given channel and can then send messages and obtain the messages that arrive. Channels can be thought of as partitions - clients connecting to one channel are unaware of the existence of other channels, but clients may connect to multiple channels. Even though some ordering service implementations included with Hyperledger Fabric support multiple channels, for simplicity of presentation, in the rest of this document, we assume ordering service consists of a single channel/topic.

**Ordering service API.** Peers connect to the channel provided by the ordering service, via the interface provided by the ordering service. The ordering service API consists of two basic operations (more generally asynchronous events):

**TODO** add the part of the API for fetching particular blocks under client/peer specified sequence numbers.

* 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.

**Ledger and block formation.** The ledger (see also Sec. 1.2.2) contains all data output by the ordering service. In a nutshell, it is a sequence of deliver(seqno, prevhash, blob) events, which form a hash chain according to the computation of prevhash described before.

Most of the time, for efficiency reasons, instead of outputting individual transactions (blobs), the ordering service will group (batch) the blobs and output blocks within a single deliver event. In this case, the ordering service must impose and convey a deterministic ordering of the blobs within each block. The number of blobs in a block may be chosen dynamically by an ordering service implementation.

In the following, for ease of presentation, we define ordering service properties (rest of this subsection) and explain the workflow of transaction endorsement (Section 2) assuming one blob per deliver event. These are easily extended to blocks, assuming that a deliver event for a block corresponds to a sequence of individual deliver events for each blob within a block, according to the above mentioned deterministic ordering of blobs within a block.

**Ordering service properties**

The guarantees of the ordering service (or atomic-broadcast channel) stipulate what happens to a broadcasted message and what relations exist among delivered messages. These guarantees are as follows:

1. **Safety (consistency guarantees)**: As long as peers are connected for sufficiently long periods of time to the channel (they can disconnect or crash, but will restart and reconnect), they will see an identical series of delivered (seqno, prevhash, blob) messages. This means the outputs (deliver() events) occur in the same order on all peers and according to sequence number and carry identical content (blob and prevhash) for the same sequence number. Note this is only a logical order, and a deliver(seqno, prevhash, blob) on one peer is not required to occur in any real-time relation to deliver(seqno, prevhash, blob) that outputs the same message at another peer. Put differently, given a particular seqno, no two correct peers deliver different prevhash or blob values. Moreover, no value blob is delivered unless some client (peer) actually called broadcast(blob) and, preferably, every broadcasted blob is only delivered once.

Furthermore, the deliver() event contains the cryptographic hash of the data in the previous deliver() event (prevhash). When the ordering service implements atomic broadcast guarantees, prevhash is the cryptographic hash of the parameters from the deliver() event with sequence number seqno-1. This establishes a hash chain across deliver() events, which is used to help verify the integrity of the ordering service output, as discussed in Sections 4 and 5 later. In the special case of the first deliver() event, prevhash has a default value.

1. **Liveness (delivery guarantee)**: Liveness guarantees of the ordering service are specified by a ordering service implementation. The exact guarantees may depend on the network and node fault model.

In principle, if the submitting client does not fail, the ordering service should guarantee that every correct peer that connects to the ordering service eventually delivers every submitted transaction.

To summarize, the ordering service ensures the following properties:

* Agreement. For any two events at correct peers deliver(seqno, prevhash0, blob0) and deliver(seqno, prevhash1, blob1) with the same seqno, prevhash0==prevhash1 and blob0==blob1;
* Hashchain integrity. For any two events at correct peers deliver(seqno-1, prevhash0, blob0) and deliver(seqno, prevhash, blob), prevhash = HASH(seqno-1||prevhash0||blob0).
* No skipping. If an ordering service outputs deliver(seqno, prevhash, blob) at a correct peer p, such that seqno>0, then p already delivered an event deliver(seqno-1, prevhash0, blob0).
* No creation. Any event deliver(seqno, prevhash, blob) at a correct peer must be preceded by a broadcast(blob) event at some (possibly distinct) peer;
* No duplication (optional, yet desirable). For any two events broadcast(blob) and broadcast(blob'), when two events deliver(seqno0, prevhash0, blob) and deliver(seqno1, prevhash1, blob') occur at correct peers and blob == blob', then seqno0==seqno1 and prevhash0==prevhash1.
* Liveness. If a correct client invokes an event broadcast(blob) then every correct peer “eventually” issues an event deliver(\*, \*, blob), where \* denotes an arbitrary value.

## 2. Basic workflow of transaction endorsement

In the following we outline the high-level request flow for a transaction.

**Remark:** Notice that the following protocol \*does not assume that all transactions are deterministic, i.e., it allows for non-deterministic transactions.\*

### 2.1. The client creates a transaction and sends it to endorsing peers of its choice

To invoke a transaction, the client sends a PROPOSE message to a set of endorsing peers of its choice (possibly not at the same time - see Sections 2.1.2. and 2.3.). The set of endorsing peers for a given chaincodeID is made available to client via peer, which in turn knows the set of endorsing peers from endorsement policy (see Section 3). For example, the transaction could be sent to all endorsers of a given chaincodeID. That said, some endorsers could be offline, others may object and choose not to endorse the transaction. The submitting client tries to satisfy the policy expression with the endorsers available.

In the following, we first detail PROPOSE message format and then discuss possible patterns of interaction between submitting client and endorsers.

### 2.1.1. PROPOSE message format

The format of a PROPOSE message is <PROPOSE,tx,[anchor]>, where tx is a mandatory and anchor optional argument explained in the following.

* tx=<clientID,chaincodeID,txPayload,timestamp,clientSig>, where
  + clientID is an ID of the submitting client,
  + chaincodeID refers to the chaincode to which the transaction pertains,
  + txPayload is the payload containing the submitted transaction itself,
  + timestamp is a monotonically increasing (for every new transaction) integer maintained by the client,
  + clientSig is signature of a client on other fields of tx.

The details of txPayload will differ between invoke transactions and deploy transactions (i.e., invoke transactions referring to a deploy-specific system chaincode). For an **invoke transaction**, txPayload would consist of two fields

* + txPayload = <operation, metadata>, where
    - operation denotes the chaincode operation (function) and arguments,
    - metadata denotes attributes related to the invocation.

For a **deploy transaction**, txPayload would consist of three fields

* + txPayload = <source, metadata, policies>, where
    - source denotes the source code of the chaincode,
    - metadata denotes attributes related to the chaincode and application,
    - policies contains policies related to the chaincode that are accessible to all peers, such as the endorsement policy. Note that endorsement policies are not supplied with txPayload in a deploy transaction, but txPayload of a deploy contains endorsement policy ID and its parameters (see Section 3).
* anchor contains read version dependencies, or more specifically, key-version pairs (i.e., anchor is a subset of KxN), that binds or “anchors” the PROPOSE request to specified versions of keys in a KVS (see Section 1.2.). If the client specifies the anchor argument, an endorser endorses a transaction only upon read version numbers of corresponding keys in its local KVS match anchor (see Section 2.2. for more details).

Cryptographic hash of tx is used by all nodes as a unique transaction identifier tid (i.e., tid=HASH(tx)). The client stores tid in memory and waits for responses from endorsing peers.

#### 2.1.2. Message patterns

The client decides on the sequence of interaction with endorsers. For example, a client would typically send <PROPOSE, tx> (i.e., without the anchor argument) to a single endorser, which would then produce the version dependencies (anchor) which the client can later on use as an argument of its PROPOSE message to other endorsers. As another example, the client could directly send <PROPOSE, tx> (without anchor) to all endorsers of its choice. Different patterns of communication are possible and client is free to decide on those (see also Section 2.3.).

### 2.2. The endorsing peer simulates a transaction and produces an endorsement signature

On reception of a <PROPOSE,tx,[anchor]> message from a client, the endorsing peer epID first verifies the client’s signature clientSig and then simulates a transaction. If the client specifies anchor then endorsing peer simulates the transactions only upon read version numbers (i.e., readset as defined below) of corresponding keys in its local KVS match those version numbers specified by anchor.

Simulating a transaction involves endorsing peer tentatively executing a transaction (txPayload), by invoking the chaincode to which the transaction refers (chaincodeID) and the copy of the state that the endorsing peer locally holds.

As a result of the execution, the endorsing peer computes read version dependencies (readset) and state updates (writeset), also called MVCC+postimage info in DB language.

Recall that the state consists of key-value pairs. All key-value entries are versioned; that is, every entry contains ordered version information, which is incremented each time the value stored under a key is updated. The peer that interprets the transaction records all key-value pairs accessed by the chaincode, either for reading or for writing, but the peer does not yet update its state. More specifically:

* Given state s before an endorsing peer executes a transaction, for every key k read by the transaction, pair (k,s(k).version) is added to readset.
* Additionally, for every key k modified by the transaction to the new value v', pair (k,v') is added to writeset. Alternatively, v' could be the delta of the new value to previous value (s(k).value).

If a client specifies anchor in the PROPOSE message then client specified anchor must equal readset produced by endorsing peer when simulating the transaction.

Then, the peer forwards internally tran-proposal (and possibly tx) to the part of its (peer’s) logic that endorses a transaction, referred to as **endorsing logic**. By default, endorsing logic at a peer accepts the tran-proposal and simply signs the tran-proposal. However, endorsing logic may interpret arbitrary functionality, to, e.g., interact with legacy systems with tran-proposal and tx as inputs to reach the decision whether to endorse a transaction or not.

If endorsing logic decides to endorse a transaction, it sends <TRANSACTION-ENDORSED, tid, tran-proposal,epSig> message to the submitting client(tx.clientID), where:

* tran-proposal := (epID,tid,chaincodeID,txContentBlob,readset,writeset),

where txContentBlob is chaincode/transaction specific information. The intention is to have txContentBlob used as some representation of tx (e.g., txContentBlob=tx.txPayload).

* epSig is the endorsing peer’s signature on tran-proposal

Else, in case the endorsing logic refuses to endorse the transaction, an endorser may send a message (TRANSACTION-INVALID, tid, REJECTED) to the submitting client.

Notice that an endorser does not change its state in this step, the updates produced by transaction simulation in the context of endorsement do not affect the state!

### 2.3. The submitting client collects an endorsement for a transaction and broadcasts it through ordering service

The submitting client waits until it receives “enough” messages and signatures on (TRANSACTION-ENDORSED, tid, \*, \*) statements to conclude that the transaction proposal is endorsed. As discussed in Section 2.1.2., this may involve one or more round-trips of interaction with endorsers.

The exact number of “enough” depend on the chaincode endorsement policy (see also Section 3). If the endorsement policy is satisfied, the transaction has been endorsed; note that it is not yet committed. The collection of signed TRANSACTION-ENDORSED messages from endorsing peers which establish that a transaction is endorsed is called an endorsement and denoted by endorsement.

If the submitting client does not manage to collect an endorsement for a transaction proposal, it abandons this transaction with an option to retry later.

For transaction with a valid endorsement, we now start using the ordering service. The submitting client invokes ordering service using the broadcast(blob), where blob=endorsement. If the client does not have capability of invoking ordering service directly, it may proxy its broadcast through some peer of its choice. Such a peer must be trusted by the client not to remove any message from the endorsement or otherwise the transaction may be deemed invalid. Notice that, however, a proxy peer may not fabricate a valid endorsement.

### 2.4. The ordering service delivers a transactions to the peers

When an event deliver(seqno, prevhash, blob) occurs and a peer has applied all state updates for blobs with sequence number lower than seqno, a peer does the following:

* It checks that the blob.endorsement is valid according to the policy of the chaincode (blob.tran-proposal.chaincodeID) to which it refers.
* In a typical case, it also verifies that the dependencies (blob.endorsement.tran-proposal.readset) have not been violated meanwhile. In more complex use cases, tran-proposal fields in endorsement may differ and in this case endorsement policy (Section 3) specifies how the state evolves.

Verification of dependencies can be implemented in different ways, according to a consistency property or “isolation guarantee” that is chosen for the state updates. **Serializability** is a default isolation guarantee, unless chaincode endorsement policy specifies a different one. Serializability can be provided by requiring the version associated with every key in the readset to be equal to that key’s version in the state, and rejecting transactions that do not satisfy this requirement.

* If all these checks pass, the transaction is deemed valid or committed. In this case, the peer marks the transaction with 1 in the bitmask of the PeerLedger, applies blob.endorsement.tran-proposal.writeset to blockchain state (if tran-proposals are the same, otherwise endorsement policy logic defines the function that takes blob.endorsement).
* If the endorsement policy verification of blob.endorsement fails, the transaction is invalid and the peer marks the transaction with 0 in the bitmask of the PeerLedger. It is important to note that invalid transactions do not change the state.

Note that this is sufficient to have all (correct) peers have the same state after processing a deliver event (block) with a given sequence number. Namely, by the guarantees of the ordering service, all correct peers will receive an identical sequence of deliver(seqno, prevhash, blob) events. As the evaluation of the endorsement policy and evaluation of version dependencies in readset are deterministic, all correct peers will also come to the same conclusion whether a transaction contained in a blob is valid. Hence, all peers commit and apply the same sequence of transactions and update their state in the same way.

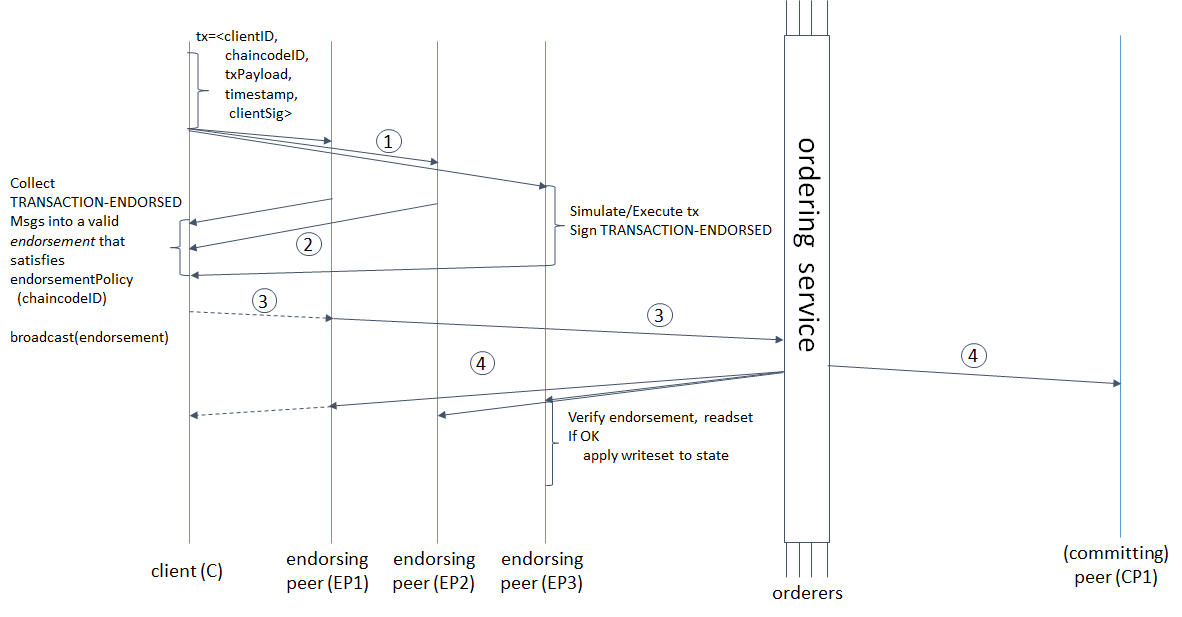


Figure 1. Illustration of one possible transaction flow (common-case path).

## 3. Endorsement policies

### 3.1. Endorsement policy specification

An **endorsement policy**, is a condition on what endorses a transaction. Blockchain peers have a pre-specified set of endorsement policies, which are referenced by a deploy transaction that installs specific chaincode. Endorsement policies can be parametrized, and these parameters can be specified by a deploy transaction.

To guarantee blockchain and security properties, the set of endorsement policies **should be a set of proven policies** with limited set of functions in order to ensure bounded execution time (termination), determinism, performance and security guarantees.

Dynamic addition of endorsement policies (e.g., by deploy transaction on chaincode deploy time) is very sensitive in terms of bounded policy evaluation time (termination), determinism, performance and security guarantees. Therefore, dynamic addition of endorsement policies is not allowed, but can be supported in future.

### 3.2. Transaction evaluation against endorsement policy

A transaction is declared valid only if it has been endorsed according to the policy. An invoke transaction for a chaincode will first have to obtain an endorsement that satisfies the chaincode’s policy or it will not be committed. This takes place through the interaction between the submitting client and endorsing peers as explained in Section 2.

Formally the endorsement policy is a predicate on the endorsement, and potentially further state that evaluates to TRUE or FALSE. For deploy transactions the endorsement is obtained according to a system-wide policy (for example, from the system chaincode).

An endorsement policy predicate refers to certain variables. Potentially it may refer to:

1. keys or identities relating to the chaincode (found in the metadata of the chaincode), for example, a set of endorsers;
2. further metadata of the chaincode;
3. elements of the endorsement and endorsement.tran-proposal;
4. and potentially more.

The above list is ordered by increasing expressiveness and complexity, that is, it will be relatively simple to support policies that only refer to keys and identities of nodes.

**The evaluation of an endorsement policy predicate must be deterministic.** An endorsement shall be evaluated locally by every peer such that a peer does not need to interact with other peers, yet all correct peers evaluate the endorsement policy in the same way.

### 3.3. Example endorsement policies

The predicate may contain logical expressions and evaluates to TRUE or FALSE. Typically the condition will use digital signatures on the transaction invocation issued by endorsing peers for the chaincode.

Suppose the chaincode specifies the endorser set E = {Alice, Bob, Charlie, Dave, Eve, Frank, George}. Some example policies:

* A valid signature from on the same tran-proposal from all members of E.
* A valid signature from any single member of E.
* Valid signatures on the same tran-proposal from endorsing peers according to the condition (Alice OR Bob) AND (any two of: Charlie, Dave, Eve, Frank, George).
* Valid signatures on the same tran-proposal by any 5 out of the 7 endorsers. (More generally, for chaincode with n > 3f endorsers, valid signatures by any 2f+1 out of the n endorsers, or by any group of more than (n+f)/2 endorsers.)
* Suppose there is an assignment of “stake” or “weights” to the endorsers, like {Alice=49, Bob=15, Charlie=15, Dave=10, Eve=7, Frank=3, George=1}, where the total stake is 100: The policy requires valid signatures from a set that has a majority of the stake (i.e., a group with combined stake strictly more than 50), such as {Alice, X} with any X different from George, or {everyone together except Alice}. And so on.
* The assignment of stake in the previous example condition could be static (fixed in the metadata of the chaincode) or dynamic (e.g., dependent on the state of the chaincode and be modified during the execution).
* Valid signatures from (Alice OR Bob) on tran-proposal1 and valid signatures from (any two of: Charlie, Dave, Eve, Frank, George) on tran-proposal2, where tran-proposal1 and tran-proposal2 differ only in their endorsing peers and state updates.

How useful these policies are will depend on the application, on the desired resilience of the solution against failures or misbehavior of endorsers, and on various other properties.

## 4 (post-v1). Validated ledger and PeerLedger checkpointing (pruning)

### 4.1. Validated ledger (VLedger)

To maintain the abstraction of a ledger that contains only valid and committed transactions (that appears in Bitcoin, for example), peers may, in addition to state and Ledger, maintain the Validated Ledger (or VLedger). This is a hash chain derived from the ledger by filtering out invalid transactions.

The construction of the VLedger blocks (called here vBlocks) proceeds as follows. As the PeerLedger blocks may contain invalid transactions (i.e., transactions with invalid endorsement or with invalid version dependencies), such transactions are filtered out by peers before a transaction from a block becomes added to a vBlock. Every peer does this by itself (e.g., by using the bitmask associated with PeerLedger). A vBlock is defined as a block without the invalid transactions, that have been filtered out. Such vBlocks are inherently dynamic in size and may be empty. An illustration of vBlock construction is given in the figure below.

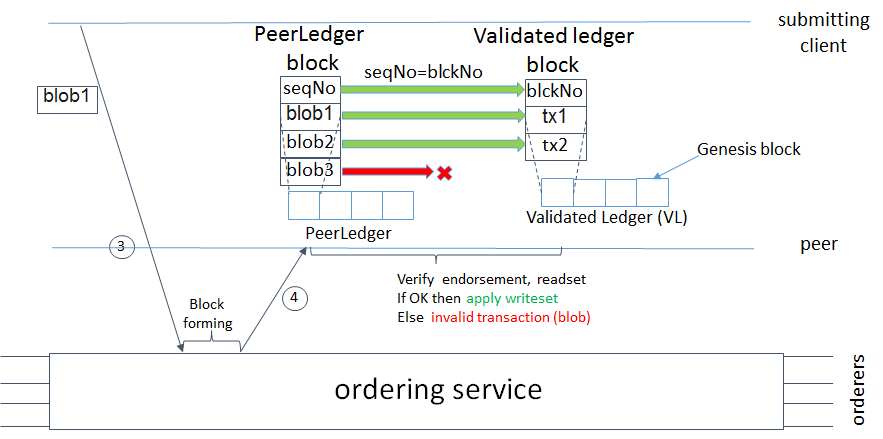


Figure 2. Illustration of validated ledger block (vBlock) formation from ledger (PeerLedger) blocks.

vBlocks are chained together to a hash chain by every peer. More specifically, every block of a validated ledger contains:

* The hash of the previous vBlock.
* vBlock number.
* An ordered list of all valid transactions committed by the peers since the last vBlock was computed (i.e., list of valid transactions in a corresponding block).
* The hash of the corresponding block (in PeerLedger) from which the current vBlock is derived.

All this information is concatenated and hashed by a peer, producing the hash of the vBlock in the validated ledger.

### 4.2. PeerLedger Checkpointing

The ledger contains invalid transactions, which may not necessarily be recorded forever. However, peers cannot simply discard PeerLedger blocks and thereby prune PeerLedger once they establish the corresponding vBlocks. Namely, in this case, if a new peer joins the network, other peers could not transfer the discarded blocks (pertaining to PeerLedger) to the joining peer, nor convince the joining peer of the validity of their vBlocks.

To facilitate pruning of the PeerLedger, this document describes a checkpointing mechanism. This mechanism establishes the validity of the vBlocks across the peer network and allows checkpointed vBlocks to replace the discarded PeerLedger blocks. This, in turn, reduces storage space, as there is no need to store invalid transactions. It also reduces the work to reconstruct the state for new peers that join the network (as they do not need to establish validity of individual transactions when reconstructing the state by replaying PeerLedger, but may simply replay the state updates contained in the validated ledger).

#### 4.2.1. Checkpointing protocol

Checkpointing is performed periodically by the peers every CHK blocks, where CHK is a configurable parameter. To initiate a checkpoint, the peers broadcast (e.g., gossip) to other peers message <CHECKPOINT,blocknohash,blockno,stateHash,peerSig>, where blockno is the current blocknumber and blocknohash is its respective hash, stateHash is the hash of the latest state (produced by e.g., a Merkle hash) upon validation of block blockno and peerSig is peer’s signature on (CHECKPOINT,blocknohash,blockno,stateHash), referring to the validated ledger.

A peer collects CHECKPOINT messages until it obtains enough correctly signed messages with matching blockno, blocknohash and stateHash to establish a valid checkpoint (see Section 4.2.2.).

Upon establishing a valid checkpoint for block number blockno with blocknohash, a peer:

* if blockno>latestValidCheckpoint.blockno, then a peer assigns latestValidCheckpoint=(blocknohash,blockno),
* stores the set of respective peer signatures that constitute a valid checkpoint into the set latestValidCheckpointProof,
* stores the state corresponding to stateHash to latestValidCheckpointedState,
* (optionally) prunes its PeerLedger up to block number blockno (inclusive).

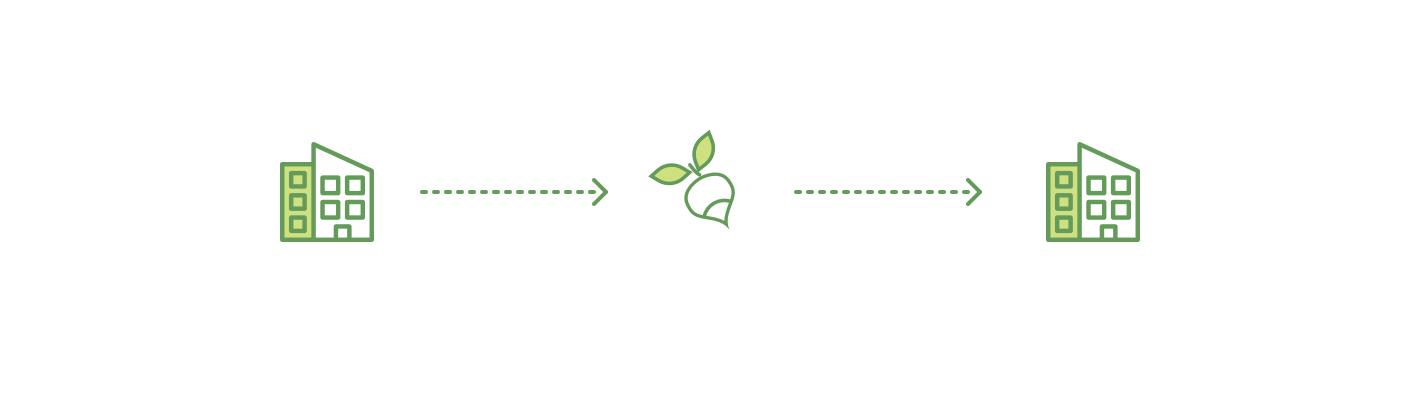
#### 4.2.2. Valid checkpoints

Clearly, the checkpointing protocol raises the following questions: When can a peer prune its ``PeerLedger``? How many ``CHECKPOINT`` messages are “sufficiently many”?. This is defined by a checkpoint validity policy, with (at least) two possible approaches, which may also be combined:

* Local (peer-specific) checkpoint validity policy (LCVP). A local policy at a given peer p may specify a set of peers which peer p trusts and whose CHECKPOINT messages are sufficient to establish a valid checkpoint. For example, LCVP at peer Alice may define that Alice needs to receive CHECKPOINT message from Bob, or from both Charlie and Dave.
* Global checkpoint validity policy (GCVP). A checkpoint validity policy may be specified globally. This is similar to a local peer policy, except that it is stipulated at the system (blockchain) granularity, rather than peer granularity. For instance, GCVP may specify that:
  + each peer may trust a checkpoint if confirmed by 11 different peers.
  + in a specific deployment in which every orderer is collocated with a peer in the same machine (i.e., trust domain) and where up to f orderers may be (Byzantine) faulty, each peer may trust a checkpoint if confirmed by f+1 different peers collocated with orderers.

# Transaction Flow

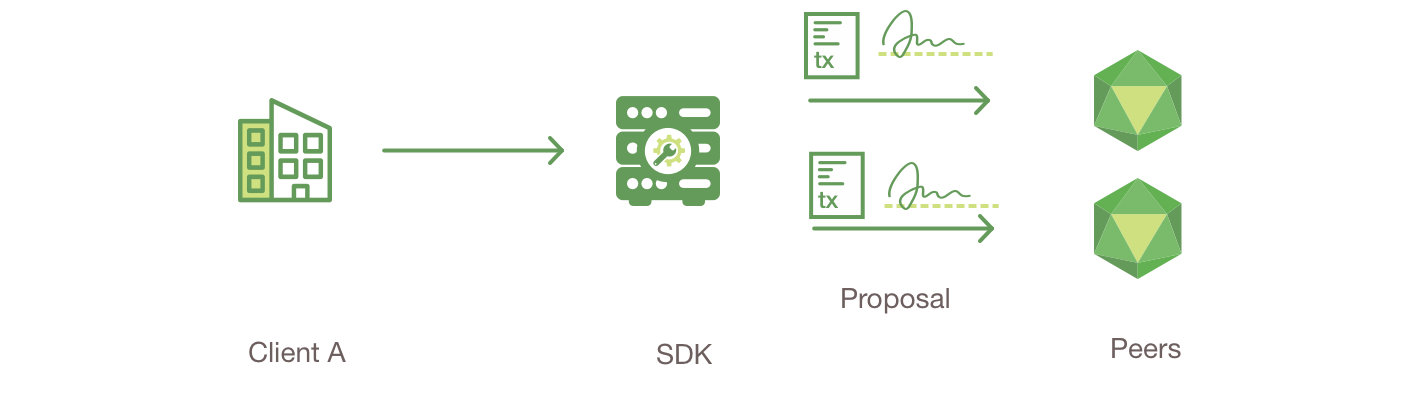
This document outlines the transactional mechanics that take place during a standard asset exchange. The scenario includes two clients, A and B, who are buying and selling radishes. They each have a peer on the network through which they send their transactions and interact with the ledger.



**Assumptions**

This flow assumes that a channel is set up and running. The application user has registered and enrolled with the organization’s Certificate Authority (CA) and received back necessary cryptographic material, which is used to authenticate to the network.

The chaincode (containing a set of key value pairs representing the initial state of the radish market) is installed on the peers and instantiated on the channel. The chaincode contains logic defining a set of transaction instructions and the agreed upon price for a radish. An endorsement policy has also been set for this chaincode, stating that both peerA and peerB must endorse any transaction.

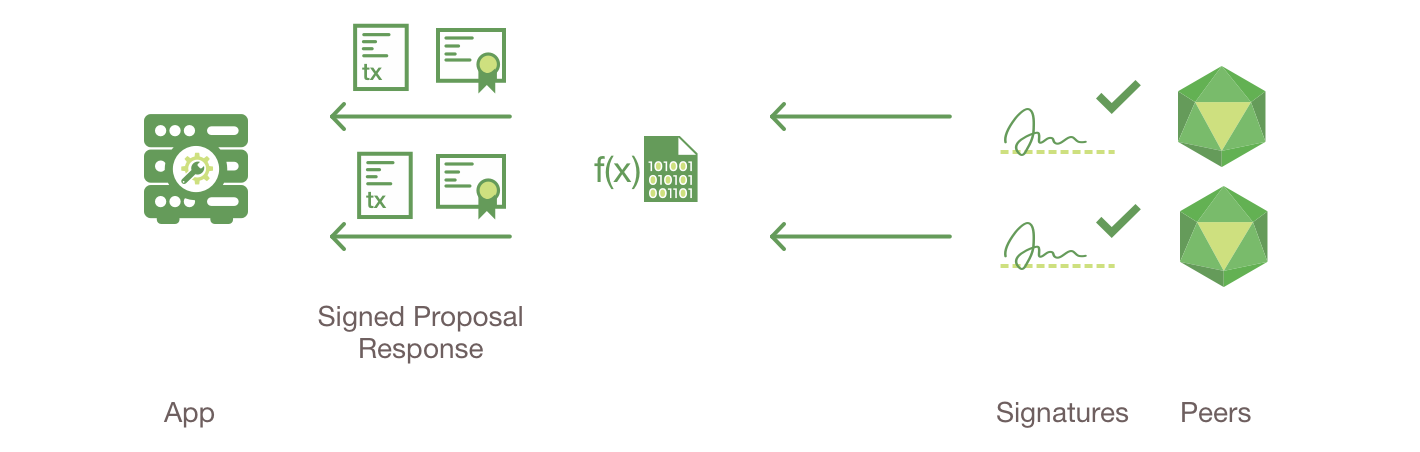


1. **Client A initiates a transaction**

What’s happening? Client A is sending a request to purchase radishes. This request targets peerA and peerB, who are respectively representative of Client A and Client B. The endorsement policy states that both peers must endorse any transaction, therefore the request goes to peerA and peerB.

Next, the transaction proposal is constructed. An application leveraging a supported SDK (Node, Java, Python) utilizes one of the available API’s to generate a transaction proposal. The proposal is a request to invoke a chaincode function with certain input parameters, with the intent of reading and/or updating the ledger.

The SDK serves as a shim to package the transaction proposal into the properly architected format (protocol buffer over gRPC) and takes the user’s cryptographic credentials to produce a unique signature for this transaction proposal.

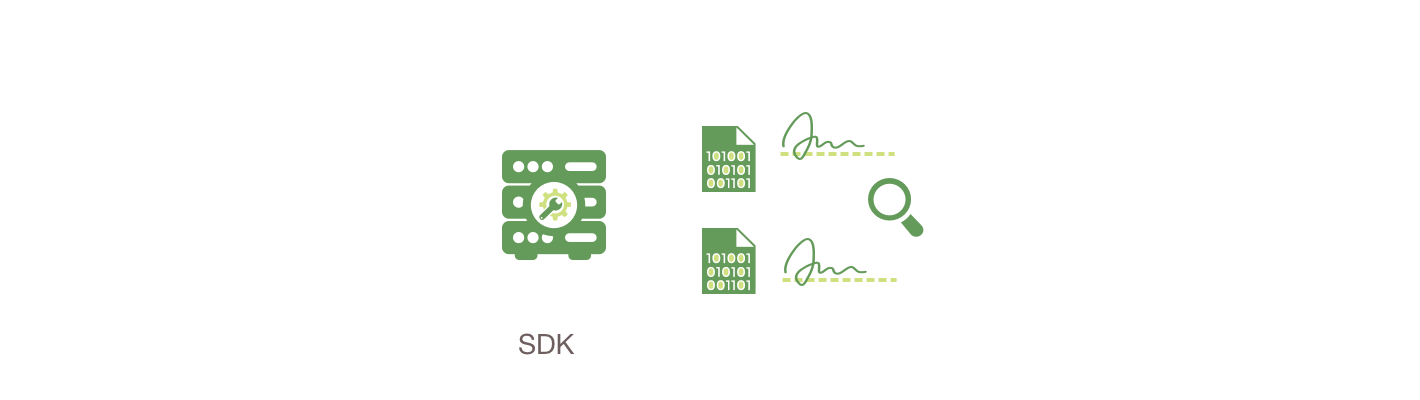


1. **Endorsing peers verify signature & execute the transaction**

The endorsing peers verify (1) that the transaction proposal is well formed, (2) it has not been submitted already in the past (replay-attack protection), (3) the signature is valid (using the MSP), and (4) that the submitter (Client A, in the example) is properly authorized to perform the proposed operation on that channel (namely, each endorsing peer ensures that the submitter satisfies the channel’s Writers policy). The endorsing peers take the transaction proposal inputs as arguments to the invoked chaincode’s function. The chaincode is then executed against the current state database to produce transaction results including a response value, read set, and write set (i.e. key/value pairs representing an asset to create or update). No updates are made to the ledger at this point. The set of these values, along with the endorsing peer’s signature is passed back as a “proposal response” to the SDK which parses the payload for the application to consume.

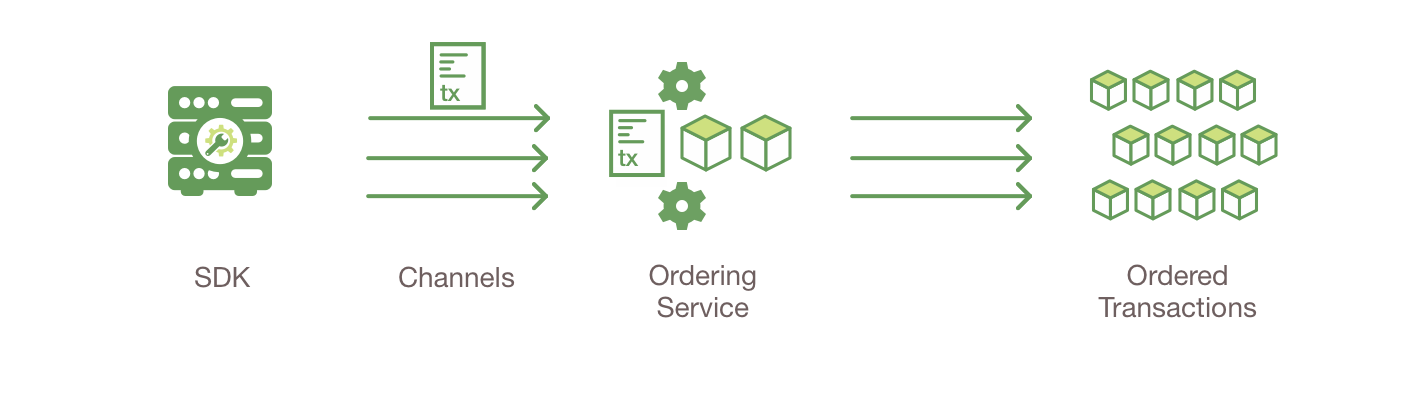
**Note**

The MSP is a peer component that allows peers to verify transaction requests arriving from clients and to sign transaction results (endorsements). The writing policy is defined at channel creation time and determines which users are entitled to submit a transaction to that channel. For more information about membership, check out our [Membership](https://hyperledger-fabric.readthedocs.io/en/release-1.4/membership/membership.html) documentation.



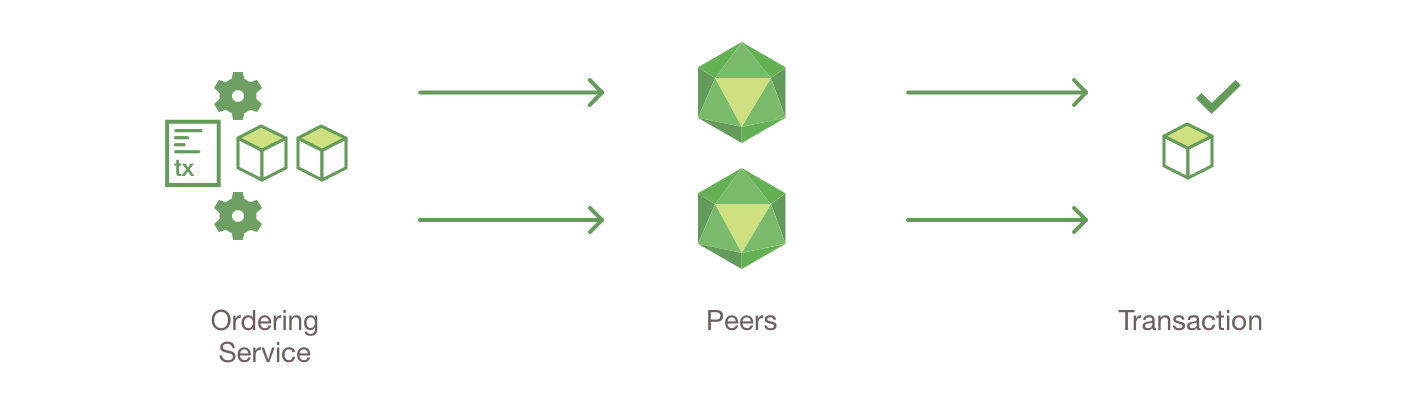
1. **Proposal responses are inspected**

The application verifies the endorsing peer signatures and compares the proposal responses to determine if the proposal responses are the same. If the chaincode is only queried the ledger, the application would inspect the query response and would typically not submit the transaction to the ordering service. If the client application intends to submit the transaction to the ordering service to update the ledger, the application determines if the specified endorsement policy has been fulfilled before submitting (i.e. did peerA and peerB both endorse). The architecture is such that even if an application chooses not to inspect responses or otherwise forwards an unendorsed transaction, the endorsement policy will still be enforced by peers and upheld at the commit validation phase.



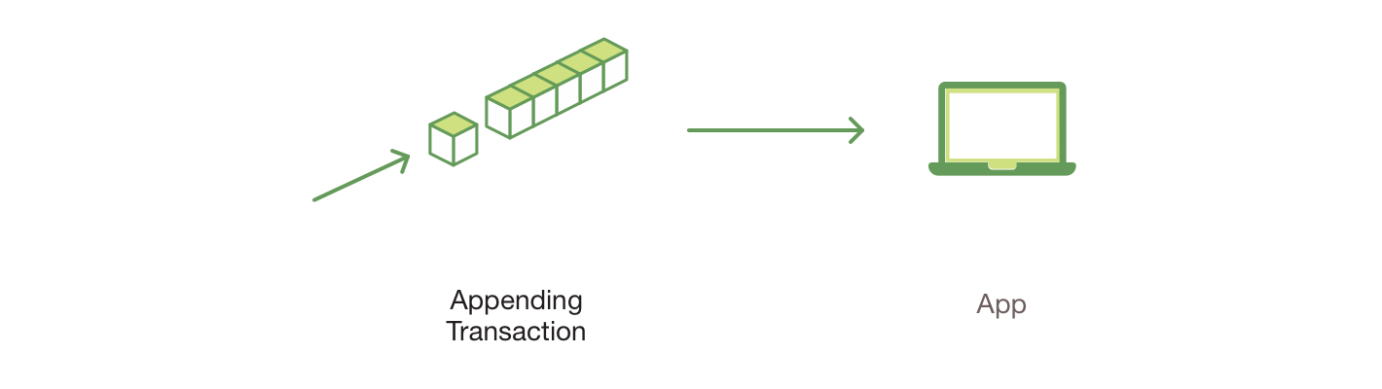
1. **Client assembles endorsements into a transaction**

The application “broadcasts” the transaction proposal and response within a “transaction message” to the ordering service. The transaction will contain the read/write sets, the endorsing peers signatures and the Channel ID. The ordering service does not need to inspect the entire content of a transaction in order to perform its operation, it simply receives transactions from all channels in the network, orders them chronologically by channel, and creates blocks of transactions per channel.



1. **Transaction is validated and committed**

The blocks of transactions are “delivered” to all peers on the channel. The transactions within the block are validated to ensure endorsement policy is fulfilled and to ensure that there have been no changes to ledger state for read set variables since the read set was generated by the transaction execution. Transactions in the block are tagged as being valid or invalid.



1. **Ledger updated**

Each peer appends the block to the channel’s chain, and for each valid transaction the write sets are committed to current state database. An event is emitted, to notify the client application that the transaction (invocation) has been immutably appended to the chain, as well as notification of whether the transaction was validated or invalidated.

**Note**

Applications should listen for the transaction event after submitting a transaction, for example by using the submitTransaction API, which automatically listen for transaction events. Without listening for transaction events, you will not know whether your transaction has actually been ordered, validated, and committed to the ledger.

See the [sequence diagram](https://hyperledger-fabric.readthedocs.io/en/release-1.4/arch-deep-dive.html#swimlane) to better understand the transaction flow.

<https://creativecommons.org/licenses/by/4.0/>

# Hyperledger Fabric SDKs

Hyperledger Fabric intends to offer a number of SDKs for a wide variety of programming languages. The first two delivered are the Node.js and Java SDKs. We hope to provide Python, REST and Go SDKs in a subsequent release.

* [Hyperledger Fabric Node SDK documentation](https://fabric-sdk-node.github.io/).
* [Hyperledger Fabric Java SDK documentation](https://github.com/hyperledger/fabric-sdk-java).

# Service Discovery

## Why do we need service discovery?

In order to execute chaincode on peers, submit transactions to orderers, and to be updated about the status of transactions, applications connect to an API exposed by an SDK.

However, the SDK needs a lot of information in order to allow applications to connect to the relevant network nodes. In addition to the CA and TLS certificates of the orderers and peers on the channel – as well as their IP addresses and port numbers – it must know the relevant endorsement policies as well as which peers have the chaincode installed (so the [application](https://hyperledger-fabric.readthedocs.io/en/release-1.4/discovery-overview.html) knows which peers to send chaincode proposals to).

Prior to v1.2, this information was statically encoded. However, this implementation is not dynamically reactive to network changes (such as the addition of peers who have installed the relevant chaincode, or peers that are temporarily offline). Static configurations also do not allow applications to react to changes of the endorsement policy itself (as might happen when a new organization joins a channel).

In addition, the client application has no way of knowing which peers have updated ledgers and which do not. As a result, the application might submit proposals to peers whose ledger data is not in sync with the rest of the network, resulting in transaction being invalidated upon commit and wasting resources as a consequence.

The **discovery service** improves this process by having the peers compute the needed information dynamically and present it to the SDK in a consumable manner.

## How service discovery works in Fabric

The application is bootstrapped knowing about a group of peers which are trusted by the application developer/administrator to provide authentic responses to discovery queries. A good candidate peer to be used by the client application is one that is in the same organization. Note that in order for peers to be known to the discovery service, they must have an EXTERNAL\_ENDPOINT defined. To see how to do this, check out our [Service Discovery CLI](https://hyperledger-fabric.readthedocs.io/en/release-1.4/discovery-cli.html) documentation.

The application issues a configuration query to the discovery service and obtains all the static information it would have otherwise needed to communicate with the rest of the nodes of the network. This information can be refreshed at any point by sending a subsequent query to the discovery service of a peer.

The service runs on peers – not on the application – and uses the network metadata information maintained by the gossip communication layer to find out which peers are [online](https://hyperledger-fabric.readthedocs.io/en/release-1.4/discovery-overview.html). It also fetches information, such as any relevant endorsement policies, from the peer’s state database.

With service discovery, applications no longer need to specify which peers they need endorsements from. The SDK can simply send a query to the discovery service asking which peers are needed given a channel and a chaincode ID. The discovery service will then compute a descriptor comprised of two objects:

1. **Layouts**: a list of groups of peers and a corresponding amount of peers from each group which should be selected.
2. **Group to peer mapping**: from the groups in the layouts to the peers of the channel. In practice, each group would most likely be peers that represent individual organizations, but because the service API is generic and ignorant of organizations this is just a “group”.

The following is an example of a descriptor from the evaluation of a policy of AND(Org1, Org2) where there are two peers in each of the organizations.

Layouts: [

QuantitiesByGroup: {

“Org1”: 1,

“Org2”: 1,

}

],

EndorsersByGroups: {

“Org1”: [peer0.org1, peer1.org1],

“Org2”: [peer0.org2, peer1.org2]

}

In other words, the endorsement policy requires a signature from one peer in Org1 and one peer in Org2. And it provides the names of available peers in those orgs who can endorse (peer0 and peer1 in both Org1 and in Org2).

The SDK then selects a random layout from the list. In the example above, the endorsement policy is Org1 AND Org2. If instead it was an OR policy, the SDK would randomly select either Org1 or Org2, since a signature from a peer from either Org would satisfy the policy.

After the SDK has selected a layout, it selects from the peers in the layout based on a criteria specified on the client side (the SDK can do this because it has access to metadata like ledger height). For example, it can prefer peers with higher ledger heights over others – or to exclude peers that the application has discovered to be offline – according to the number of peers from each group in the layout. If no single peer is preferable based on the criteria, the SDK will randomly select from the peers that best meet the criteria.

### Capabilities of the discovery service

The discovery service can respond to the following queries:

* **Configuration query**: Returns the MSPConfig of all organizations in the channel along with the orderer endpoints of the channel.
* **Peer membership query**: Returns the peers that have joined the channel.
* **Endorsement query**: Returns an endorsement descriptor for given chaincode(s) in a channel.
* **Local peer membership query**: Returns the local membership information of the peer that responds to the query. By default the client needs to be an administrator for the peer to respond to this query.

### Special requirements

When the peer is running with TLS enabled the client must provide a TLS certificate when connecting to the peer. If the peer isn’t configured to verify client certificates (clientAuthRequired is false), this TLS certificate can be self-signed.

# Channels

A Hyperledger Fabric channel is a private “subnet” of communication between two or more specific network members, for the purpose of conducting private and confidential transactions. A channel is defined by members (organizations), anchor peers per member, the shared ledger, chaincode application(s) and the ordering service node(s). Each transaction on the network is executed on a channel, where each party must be authenticated and authorized to transact on that channel. Each peer that joins a channel, has its own identity given by a membership services provider (MSP), which authenticates each peer to its channel peers and services.

To [create](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channels.html) a new channel, the client SDK calls configuration system chaincode and references properties such as anchor peers, and members (organizations). This request creates a genesis block for the channel ledger, which stores configuration information about the channel policies, members and anchor peers. When adding a new member to an existing channel, either this genesis block, or if applicable, a more recent reconfiguration block, is shared with the new member.

**Note**

See the [Channel Configuration (configtx)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/configtx.html) section for more [details](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channels.html) on the properties and proto structures of config transactions.

The election of a leading peer for each member on a channel determines which peer communicates with the ordering service on behalf of the member. If no leader is identified, an algorithm can be used to identify the leader. The consensus service orders transactions and delivers them, in a block, to each leading peer, which then distributes the block to its member peers, and across the channel, using the gossip protocol.

Although any one anchor peer can belong to multiple channels, and therefore maintain multiple ledgers, no ledger data can pass from one channel to another. This separation of ledgers, by channel, is defined and implemented by configuration chaincode, the identity membership service and the gossip data dissemination protocol. The dissemination of data, which includes information on transactions, ledger state and channel membership, is restricted to peers with verifiable membership on the channel. This isolation of peers and ledger data, by channel, allows network members that require private and confidential transactions to coexist with business competitors and other restricted members, on the same blockchain network.

# CouchDB as the State Database

## State Database options

State database options include LevelDB and CouchDB. LevelDB is the default key-value state database embedded in the peer process. CouchDB is an optional alternative external state database. Like the LevelDB key-value store, CouchDB can store any binary data that is modeled in chaincode (CouchDB attachment functionality is used internally for non-JSON binary data). But as a JSON [document](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html) store, CouchDB additionally enables rich query against the chaincode data, when chaincode values (e.g. assets) are modeled as JSON data.

Both LevelDB and CouchDB support core chaincode operations such as getting and setting a key (asset), and querying based on keys. Keys can be queried by range, and composite keys can be modeled to enable equivalence queries against multiple parameters. For example a composite key of owner,asset\_id can be used to query all assets owned by a certain entity. These key-based queries can be used for read-only queries against the ledger, as well as in transactions that update the ledger.

If you model assets as JSON and use CouchDB, you can also perform complex rich queries against the chaincode data values, using the CouchDB JSON query language within chaincode. These types of queries are excellent for understanding what is on the ledger. Proposal responses for these types of queries are typically useful to the client [application](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html), but are not typically submitted as transactions to the ordering service. In fact, there is no guarantee the result set is stable between chaincode execution and commit time for rich queries, and therefore rich queries are not appropriate for use in update transactions, unless your application can guarantee the result set is stable between chaincode execution time and commit time, or can handle potential changes in subsequent transactions. For example, if you perform a rich query for all assets owned by Alice and transfer them to Bob, a new asset may be assigned to Alice by another transaction between chaincode execution time and commit time, and you would miss this “phantom” item.

CouchDB runs as a separate database process alongside the peer, therefore there are additional considerations in terms of setup, management, and operations. You may consider starting with the default embedded LevelDB, and move to CouchDB if you require the additional complex rich queries. It is a good practice to model chaincode asset data as JSON, so that you have the option to perform complex rich queries if needed in the future.

**Note**

The key for a CouchDB JSON document can only contain valid UTF-8 strings and cannot begin with an underscore (“\_”). Whether you are using CouchDB or LevelDB, you should avoid using U+0000 (nil byte) in keys.

JSON documents in CouchDB cannot use the following values as top level field names. These values are reserved for internal use.

* Any field beginning with an underscore, "\_"
* ~version

## Using CouchDB from Chaincode

### Chaincode queries

Most of the [chaincode shim APIs](https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim#ChaincodeStubInterface) can be utilized with either LevelDB or CouchDB state database, e.g. GetState, PutState, GetStateByRange, GetStateByPartialCompositeKey. Additionally when you utilize CouchDB as the state database and model assets as JSON in chaincode, you can perform rich queries against the JSON in the state database by using the GetQueryResult API and passing a CouchDB query string. The query string follows the [CouchDB JSON query syntax](http://docs.couchdb.org/en/2.1.1/api/database/find.html).

The [marbles02 fabric sample](https://github.com/hyperledger/fabric-samples/blob/master/chaincode/marbles02/go/marbles_chaincode.go) demonstrates use of CouchDB queries from chaincode. It includes a queryMarblesByOwner() function that demonstrates parameterized queries by passing an owner id into chaincode. It then queries the state data for JSON documents matching the docType of “marble” and the owner id using the JSON query syntax:

{"selector":{"docType":"marble","owner":**<**OWNER\_ID**>**}}

#### CouchDB pagination

Fabric supports paging of query results for rich queries and range based queries. APIs supporting pagination allow the use of page size and bookmarks to be used for both range and rich queries. To support efficient pagination, the Fabric pagination APIs must be used. Specifically, the CouchDB limit keyword will not be honored in CouchDB queries since Fabric itself manages the pagination of query results and implicitly sets the pageSize limit that is passed to CouchDB.

If a pageSize is specified using the paginated query APIs (GetStateByRangeWithPagination(), GetStateByPartialCompositeKeyWithPagination(), and GetQueryResultWithPagination()), a set of results (bound by the pageSize) will be returned to the chaincode along with a bookmark. The bookmark can be returned from chaincode to invoking clients, which can use the bookmark in a follow on query to receive the next “page” of results.

The pagination APIs are for use in read-only transactions only, the query results are intended to support client paging requirements. For transactions that need to read and write, use the non-paginated chaincode query APIs. Within chaincode you can iterate through result sets to your desired depth.

Regardless of whether the pagination APIs are utilized, all chaincode queries are bound by totalQueryLimit (default 100000) from core.yaml. This is the maximum number of results that chaincode will iterate through and return to the client, in [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html) to avoid accidental or malicious long-running queries.

**Note**

Regardless of whether chaincode uses paginated queries or not, the peer will query CouchDB in batches based on internalQueryLimit (default 1000) from core.yaml. This behavior ensures reasonably sized result sets are passed between the peer and CouchDB when executing chaincode, and is transparent to chaincode and the calling client.

An example using pagination is included in the [Using CouchDB](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_tutorial.html) tutorial.

### CouchDB indexes

Indexes in CouchDB are required in order to make JSON queries efficient and are required for any JSON query with a sort. Indexes can be packaged alongside chaincode in a /META-INF/statedb/couchdb/indexes directory. Each index must be defined in its own text file with extension \*.json with the index definition formatted in JSON following the [CouchDB index JSON syntax](http://docs.couchdb.org/en/2.1.1/api/database/find.html" \l "db-index). For example, to support the above marble query, a sample index on the docType and owner fields is provided:

{"index":{"fields":["docType","owner"]},"ddoc":"indexOwnerDoc", "name":"indexOwner","type":"json"}

The sample index can be found [here](https://github.com/hyperledger/fabric-samples/blob/master/chaincode/marbles02/go/META-INF/statedb/couchdb/indexes/indexOwner.json).

Any index in the chaincode’s META-INF/statedb/couchdb/indexes directory will be packaged up with the chaincode for deployment. When the chaincode is both installed on a peer and instantiated on one of the peer’s channels, the index will automatically be deployed to the peer’s channel and chaincode specific state database (if it has been configured to use CouchDB). If you [install](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html) the chaincode first and then instantiate the chaincode on the channel, the index will be deployed at chaincode **instantiation** time. If the chaincode is already instantiated on a channel and you later install the chaincode on a peer, the index will be deployed at chaincode **installation** time.

Upon deployment, the index will automatically be utilized by chaincode queries. CouchDB can automatically determine which index to use based on the fields being used in a query. Alternatively, in the selector query the index can be specified using the use\_index keyword.

The same index may exist in subsequent versions of the chaincode that gets installed. To change the index, use the same index name but alter the index definition. Upon installation/instantiation, the index definition will get re-deployed to the peer’s state database.

If you have a large volume of data already, and later install the chaincode, the index creation upon installation may take some time. Similarly, if you have a large volume of data already and instantiate a subsequent version of the chaincode, the index creation may take some time. Avoid calling chaincode functions that query the state database at these times as the chaincode query may time out while the index is getting initialized. During transaction processing, the indexes will automatically get refreshed as blocks are committed to the ledger.

## CouchDB Configuration

CouchDB is enabled as the state database by changing the stateDatabase configuration option from goleveldb to CouchDB. Additionally, the couchDBAddress needs to configured to point to the CouchDB to be used by the peer. The username and password properties should be populated with an admin username and password if CouchDB is configured with a username and password. Additional options are provided in the couchDBConfig section and are documented in place. Changes to the core.yaml will be effective immediately after restarting the peer.

You can also pass in docker environment variables to override core.yaml values, for example CORE\_LEDGER\_STATE\_STATEDATABASE and CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_COUCHDBADDRESS.

Below is the stateDatabase section from core.yaml:

state:

*# stateDatabase - options are "goleveldb", "CouchDB"*

*# goleveldb - default state database stored in goleveldb.*

*# CouchDB - store state database in CouchDB*

stateDatabase: goleveldb

*# Limit on the number of records to return per query*

totalQueryLimit: 10000

couchDBConfig:

*# It is recommended to run CouchDB on the same server as the peer, and*

*# not map the CouchDB container port to a server port in docker-compose.*

*# Otherwise proper security must be provided on the connection between*

*# CouchDB client (on the peer) and server.*

couchDBAddress: couchdb:5984

*# This username must have read and write authority on CouchDB*

username:

*# The password is recommended to pass as an environment variable*

*# during start up (e.g. LEDGER\_COUCHDBCONFIG\_PASSWORD).*

*# If it is stored here, the file must be access control protected*

*# to prevent unintended users from discovering the password.*

password:

*# Number of retries for CouchDB errors*

maxRetries: 3

*# Number of retries for CouchDB errors during peer startup*

maxRetriesOnStartup: 10

*# CouchDB request timeout (unit: duration, e.g. 20s)*

requestTimeout: 35s

*# Limit on the number of records per each CouchDB query*

*# Note that chaincode queries are only bound by totalQueryLimit.*

*# Internally the chaincode may execute multiple CouchDB queries,*

*# each of size internalQueryLimit.*

internalQueryLimit: 1000

*# Limit on the number of records per CouchDB bulk update batch*

maxBatchUpdateSize: 1000

*# Warm indexes after every N blocks.*

*# This option warms any indexes that have been*

*# deployed to CouchDB after every N blocks.*

*# A value of 1 will warm indexes after every block commit,*

*# to ensure fast selector queries.*

*# Increasing the value may improve write efficiency of peer and CouchDB,*

*# but may degrade query response time.*

warmIndexesAfterNBlocks: 1

CouchDB hosted in docker containers supplied with Hyperledger Fabric have the capability of setting the CouchDB username and password with environment variables passed in with the COUCHDB\_USER and COUCHDB\_PASSWORD environment variables using Docker Compose scripting.

For CouchDB installations outside of the docker images supplied with Fabric, the [local.ini file of that installation](http://docs.couchdb.org/en/2.1.1/config/intro.html#configuration-files) must be edited to set the admin username and password.

Docker compose scripts only set the username and password at the creation of the container. The local.ini file must be edited if the username or password is to be changed after creation of the container.

**Note**

CouchDB peer options are read on each peer startup.

## Good practices for queries

Avoid using chaincode for queries that will result in a scan of the entire CouchDB database. Full length database scans will result in long response times and will degrade the performance of your network. You can take some of the following steps to avoid long queries:

* When using JSON queries:
  + Be sure to create indexes in the chaincode package.
  + Avoid query operators such as $or, $in and $regex, which lead to full database scans.
* For range queries, composite key queries, and JSON queries:
  + Utilize paging support (as of v1.3) instead of one large result set.
* If you want to build a dashboard or collect aggregate data as part of your application, you can query an off-chain database that replicates the data from your blockchain network. This will allow you to query and analyze the blockchain data in a data store optimized for your needs, without degrading the performance of your network or disrupting transactions. To achieve this, applications may use block or chaincode events to write transaction data to an off-chain database or analytics engine. For each block received, the block listener application would iterate through the block transactions and build a data store using the key/value writes from each valid transaction’s rwset. The [Peer channel-based event services](https://hyperledger-fabric.readthedocs.io/en/release-1.4/peer_event_services.html) provide replayable events to ensure the integrity of downstream data stores.

# Peer channel-based event services

## General overview

In previous versions of Fabric, the peer event service was known as the event hub. This service sent events any time a new block was added to the peer’s ledger, regardless of the channel to which that block pertained, and it was only accessible to members of the organization running the eventing peer (i.e., the one being connected to for events).

Starting with v1.1, there are two new services which provide events. These services use an entirely different design to provide events on a per-channel basis. This means that registration for events occurs at the level of the channel instead of the peer, allowing for fine-grained control over access to the peer’s data. Requests to receive events are accepted from identities outside of the peer’s organization (as defined by the channel configuration). This also provides greater reliability and a way to receive events that may have been missed (whether due to a connectivity issue or because the peer is joining a network that has already been running).

## Available services

* Deliver

This service sends entire blocks that have been committed to the ledger. If any events were set by a chaincode, these can be found within the ChaincodeActionPayload of the block.

* DeliverFiltered

This service sends “filtered” blocks, minimal sets of information about blocks that have been committed to the ledger. It is intended to be used in a network where owners of the peers wish for external clients to primarily receive information about their transactions and the status of those transactions. If any events were set by a chaincode, these can be found within the FilteredChaincodeAction of the filtered block.

**Note**

The payload of chaincode events will not be included in filtered blocks.

## How to register for events

Registration for events from either service is done by sending an envelope containing a deliver seek info message to the peer that contains the desired start and stop positions, the seek behavior (block until ready or fail if not ready). There are helper variables SeekOldest and SeekNewest that can be used to indicate the oldest (i.e. first) block or the newest (i.e. last) block on the ledger. To have the services send events indefinitely, the SeekInfo message should include a stop position of MAXINT64.

**Note**

If mutual TLS is enabled on the peer, the TLS certificate hash must be set in the envelope’s channel header.

By default, both services use the Channel Readers policy to determine whether to authorize requesting clients for events.

## Overview of deliver response messages

The event services send back DeliverResponse messages.

Each message contains one of the following:

* status – HTTP status code. Both services will return the appropriate failure code if any failure occurs; otherwise, it will return 200 - SUCCESS once the service has completed sending all information requested by the SeekInfo message.
* block – returned only by the Deliver service.
* filtered block – returned only by the DeliverFiltered service.

A filtered block contains:

* channel ID.
* number (i.e. the block number).
* array of filtered transactions.
* transaction ID.
  + type (e.g. ENDORSER\_TRANSACTION, CONFIG.
  + transaction validation code.
* **filtered transaction actions.**
  + **array of filtered chaincode actions.**
    - chaincode event for the transaction (with the payload nilled out).

## SDK event documentation

For further details on using the event services, refer to the [SDK documentation.](https://fabric-sdk-node.github.io/tutorial-channel-events.html)

# Private Data

**Note**

This topic assumes an understanding of the conceptual material in the [documentation on private data](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html).

## Private data collection definition

A collection definition contains one or more collections, each having a policy definition listing the organizations in the collection, as well as properties used to control dissemination of private data at endorsement time and, optionally, whether the data will be purged.

The collection definition gets deployed to the channel at the time of chaincode instantiation (or upgrade). If using the peer CLI to instantiate the chaincode, the collection definition file is passed to the chaincode instantiation using the --collections-config flag. If using a client SDK, check the [SDK documentation](https://fabric-sdk-node.github.io/) for information on providing the collection definition.

Collection definitions are composed of the following properties:

* name: Name of the collection.
* policy: The private data collection distribution policy defines which organizations’ peers are allowed to persist the collection data expressed using the Signature policy syntax, with each member being included in an OR signature policy list. To support read/write transactions, the private data distribution policy must define a broader set of organizations than the chaincode endorsement policy, as peers must have the private data in [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data-arch.html) to endorse proposed transactions. For example, in a channel with ten organizations, five of the organizations might be included in a private data collection distribution policy, but the endorsement policy might call for any three of the organizations to endorse.
* requiredPeerCount: Minimum number of peers (across authorized organizations) that each endorsing peer must successfully disseminate private data to before the peer signs the endorsement and returns the proposal response back to the client. Requiring dissemination as a condition of endorsement will ensure that private data is available in the network even if the endorsing peer(s) become unavailable. When requiredPeerCount is 0, it means that no distribution is **required**, but there may be some distribution if maxPeerCount is greater than zero. A requiredPeerCount of 0 would typically not be recommended, as it could lead to loss of private data in the network if the endorsing peer(s) becomes unavailable. Typically you would want to require at least some distribution of the private data at endorsement time to ensure redundancy of the private data on multiple peers in the network.
* maxPeerCount: For data redundancy purposes, the maximum number of other peers (across authorized organizations) that each endorsing peer will attempt to distribute the private data to. If an endorsing peer becomes unavailable between endorsement time and commit time, other peers that are collection members but who did not yet receive the private data at endorsement time, will be able to pull the private data from peers the private data was disseminated to. If this value is set to 0, the private data is not disseminated at endorsement time, forcing private data pulls against endorsing peers on all authorized peers at commit time.
* blockToLive: Represents how long the data should [live](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data-arch.html) on the private database in terms of blocks. The data will live for this specified number of blocks on the private database and after that it will get purged, making this data obsolete from the network so that it cannot be queried from chaincode, and cannot be made available to requesting peers. To keep private data indefinitely, that is, to never purge private data, set the blockToLive property to 0.
* memberOnlyRead: a value of true indicates that peers automatically enforce that only clients belonging to one of the collection member organizations are allowed read access to private data. If a client from a non-member org attempts to execute a chaincode function that performs a read of a private data, the chaincode invocation is terminated with an error. Utilize a value of false if you would like to encode more granular access control within individual chaincode functions.

Here is a sample collection definition JSON file, containing an array of two collection definitions:

[

{

"name": "collectionMarbles",

"policy": "OR('Org1MSP.member', 'Org2MSP.member')",

"requiredPeerCount": 0,

"maxPeerCount": 3,

"blockToLive":1000000,

"memberOnlyRead": true

},

{

"name": "collectionMarblePrivateDetails",

"policy": "OR('Org1MSP.member')",

"requiredPeerCount": 0,

"maxPeerCount": 3,

"blockToLive":3,

"memberOnlyRead": true

}

]

This example uses the organizations from the BYFN sample network, Org1 and Org2 . The policy in the collectionMarbles definition authorizes both organizations to the private data. This is a typical configuration when the chaincode data needs to remain private from the ordering service nodes. However, the policy in the collectionMarblePrivateDetails definition restricts access to a subset of organizations in the channel (in this case Org1 ). In a real scenario, there would be many organizations in the channel, with two or more organizations in each collection sharing private data between them.

## Private data dissemination

Since private data is not included in the transactions that get submitted to the ordering service, and therefore not included in the blocks that get distributed to all peers in a channel, the endorsing peer plays an important role in disseminating private data to other peers of authorized organizations. This ensures the availability of private data in the channel’s collection, even if endorsing peers become unavailable after their endorsement. To assist with this dissemination, the maxPeerCount and requiredPeerCount properties in the collection definition control the degree of dissemination at endorsement time.

If the endorsing peer cannot successfully disseminate the private data to at least the requiredPeerCount, it will return an error back to the client. The endorsing peer will attempt to disseminate the private data to peers of different organizations, in an effort to ensure that each authorized organization has a copy of the private data. Since transactions are not committed at chaincode execution time, the endorsing peer and recipient peers store a copy of the private data in a local transient store alongside their blockchain until the transaction is committed.

When authorized peers do not have a copy of the private data in their transient data store at commit time (either because they were not an endorsing peer or because they did not receive the private data via dissemination at endorsement time), they will attempt to pull the private data from another authorized peer, for a configurable amount of time based on the peer property peer.gossip.pvtData.pullRetryThreshold in the peer configuration core.yaml file.

**Note**

The peers being asked for private data will only return the private data if the requesting peer is a member of the collection as defined by the private data dissemination policy.

Considerations when using pullRetryThreshold:

* If the requesting peer is able to retrieve the private data within the pullRetryThreshold, it will commit the transaction to its ledger (including the private data hash), and store the private data in its state database, logically separated from other channel state data.
* If the requesting peer is not able to retrieve the private data within the pullRetryThreshold, it will commit the transaction to it’s blockchain (including the private data hash), without the private data.
* If the peer was entitled to the private data but it is missing, then that peer will not be able to endorse future transactions that reference the missing private data - a chaincode query for a key that is missing will be detected (based on the presence of the key’s hash in the state database), and the chaincode will receive an error.

Therefore, it is important to set the requiredPeerCount and maxPeerCount properties large enough to ensure the availability of private data in your channel. For example, if each of the endorsing peers become unavailable before the transaction commits, the requiredPeerCount and maxPeerCount properties will have ensured the private data is available on other peers.

**Note**

For collections to work, it is important to have cross organizational gossip configured correctly. Refer to our documentation on [Gossip data dissemination protocol](https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html), paying particular attention to the “anchor peers” and “external endpoint” configuration.

## Referencing collections from chaincode

A set of [shim APIs](https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim) are available for setting and retrieving private data.

The same chaincode data operations can be applied to channel state data and private data, but in the case of private data, a collection name is specified along with the data in the chaincode APIs, for example PutPrivateData(collection,key,value) and GetPrivateData(collection,key).

A single chaincode can reference multiple collections.

### How to pass private data in a chaincode proposal

Since the chaincode proposal gets stored on the blockchain, it is also important not to include private data in the main part of the chaincode proposal. A special field in the chaincode proposal called the transient field can be used to pass private data from the client (or data that chaincode will use to generate private data), to chaincode invocation on the peer. The chaincode can retrieve the transient field by calling the [GetTransient() API](https://github.com/hyperledger/fabric/blob/8b3cbda97e58d1a4ff664219244ffd1d89d7fba8/core/chaincode/shim/interfaces.go" \l "L315-L321). This transient field gets excluded from the channel transaction.

### Protecting private data content

If the private data is relatively simple and predictable (e.g. transaction dollar amount), channel members who are not authorized to the private data collection could try to guess the content of the private data via brute force hashing of the domain space, in hopes of finding a match with the private data hash on the chain. Private data that is predictable should therefore include a random “salt” that is concatenated with the private data key and included in the private data value, so that a matching hash cannot realistically be found via brute force. The random “salt” can be generated at the client side (e.g. by sampling a secure psuedo-random source) and then passed along with the private data in the transient field at the time of chaincode invocation.

### Access control for private data

Until version 1.3, access control to private data based on collection membership was enforced for peers only. Access control based on the organization of the chaincode proposal submitter was required to be encoded in chaincode logic. Starting in v1.4 a collection configuration option memberOnlyRead can automatically enforce access control based on the organization of the chaincode proposal submitter. For more information about collection configuration definitions and how to set them, refer back to the [Private data collection definition](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data-arch.html#private-data-collection-definition) section of this topic.

**Note**

If you would like more granular access control, you can set memberOnlyRead to false. You can then apply your own access control logic in chaincode, for example by calling the GetCreator() chaincode API or using the client identity [chaincode library](https://github.com/hyperledger/fabric/tree/master/core/chaincode/shim/ext/cid) .

### Querying Private Data

Private data collection can be queried just like normal channel data, using shim APIs:

* GetPrivateDataByRange(collection, startKey, endKey string)
* GetPrivateDataByPartialCompositeKey(collection, objectType string, keys []string)

And for the CouchDB state database, JSON content queries can be passed using the shim API:

* GetPrivateDataQueryResult(collection, query string)

Limitations:

* Clients that call chaincode that executes range or rich JSON queries should be aware that they may receive a subset of the result set, if the peer they query has missing private data, based on the explanation in Private Data Dissemination section above. Clients can query multiple peers and compare the results to determine if a peer may be missing some of the result set.
* Chaincode that executes range or rich JSON queries and updates data in a single transaction is not supported, as the query results cannot be validated on the peers that don’t have access to the private data, or on peers that are missing the private data that they have access to. If a chaincode invocation both queries and updates private data, the proposal request will return an error. If your application can tolerate result set changes between chaincode execution and validation/commit time, then you could call one chaincode function to perform the query, and then call a second chaincode function to make the updates. Note that calls to GetPrivateData() to retrieve individual keys can be made in the same transaction as PutPrivateData() calls, since all peers can validate key reads based on the hashed key version.

### Using Indexes with collections

The topic [CouchDB as the State Database](https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html) describes indexes that can be applied to the channel’s state database to enable JSON content queries, by packaging indexes in a META-INF/statedb/couchdb/indexes directory at chaincode installation time. Similarly, indexes can also be applied to private data collections, by packaging indexes in a META-INF/statedb/couchdb/collections/<collection\_name>/indexes directory. An example index is available [here](https://github.com/hyperledger/fabric-samples/blob/master/chaincode/marbles02_private/go/META-INF/statedb/couchdb/collections/collectionMarbles/indexes/indexOwner.json).

## Considerations when using private data

### Private data purging

Private data can be periodically purged from peers. For more details, see the blockToLive collection definition property above.

Additionally, recall that prior to commit, peers store private data in a local transient data store. This data automatically gets purged when the transaction commits. But if a transaction was never submitted to the channel and therefore never committed, the private data would remain in each peer’s transient store. This data is purged from the transient store after a configurable number blocks by using the peer’s peer.gossip.pvtData.transientstoreMaxBlockRetention property in the peer core.yaml file.

### Updating a collection definition

To update a collection definition or add a new collection, you can upgrade the chaincode to a new version and pass the new collection configuration in the chaincode upgrade transaction, for example using the --collections-config flag if using the CLI. If a collection configuration is specified during the chaincode upgrade, a definition for each of the existing collections must be included.

When upgrading a chaincode, you can add new private data collections, and update existing private data collections, for example to add new members to an existing collection or change one of the collection definition properties. Note that you cannot update the collection name or the blockToLive property, since a consistent blockToLive is required regardless of a peer’s block height.

Collection updates becomes effective when a peer commits the block that contains the chaincode upgrade transaction. Note that collections cannot be deleted, as there may be prior private data hashes on the channel’s blockchain that cannot be removed.

### Private data reconciliation

Starting in v1.4, peers of organizations that are added to an existing collection will automatically fetch private data that was committed to the collection before they joined the collection.

This private data “reconciliation” also applies to peers that were entitled to receive private data but did not yet receive it — because of a network failure, for example — by keeping track of private data that was “missing” at the time of block commit.

Private data reconciliation occurs periodically based on the peer.gossip.pvtData.reconciliationEnabled and peer.gossip.pvtData.reconcileSleepInterval properties in core.yaml. The peer will periodically attempt to fetch the private data from other collection member peers that are expected to have it.

Note that this private data reconciliation feature only works on peers running v1.4 or later of Fabric.

# Read-Write set semantics

This [document](https://hyperledger-fabric.readthedocs.io/en/release-1.4/readwrite.html) discusses the details of the current implementation about the semantics of read-write sets.

## Transaction simulation and read-write set

During simulation of a transaction at an endorser, a read-write set is prepared for the transaction. The read set contains a list of unique keys and their committed versions that the transaction reads during simulation. The write set contains a list of unique keys (though there can be overlap with the keys present in the read set) and their new values that the transaction writes. A delete marker is set (in the place of new value) for the key if the update performed by the transaction is to delete the key.

Further, if the transaction writes a value multiple times for a key, only the last written value is retained. Also, if a transaction reads a value for a key, the value in the committed state is returned even if the transaction has updated the value for the key before issuing the read. In another words, Read-your-writes semantics are not supported.

As noted earlier, the versions of the keys are recorded only in the read set; the write set just contains the list of unique keys and their latest values set by the transaction.

There could be various schemes for implementing versions. The minimal requirement for a versioning scheme is to produce non-repeating identifiers for a given key. For instance, using monotonically increasing numbers for versions can be one such scheme. In the current implementation, we use a blockchain height based versioning scheme in which the height of the committing transaction is used as the latest version for all the keys modified by the transaction. In this scheme, the height of a transaction is represented by a tuple (txNumber is the height of the transaction within the block). This scheme has many advantages over the incremental number scheme - primarily, it enables other components such as statedb, transaction simulation and validation for making efficient design choices.

Following is an illustration of an example read-write set prepared by simulation of a hypothetical transaction. For the sake of simplicity, in the illustrations, we use the incremental numbers for representing the versions.

**<**TxReadWriteSet**>**

**<**NsReadWriteSet name**=**"chaincode1"**>**

**<**read**-**set**>**

**<**read key**=**"K1", version**=**"1"**>**

**<**read key**=**"K2", version**=**"1"**>**

**</**read**-**set**>**

**<**write**-**set**>**

**<**write key**=**"K1", value**=**"V1"

**<**write key**=**"K3", value**=**"V2"

**<**write key**=**"K4", isDelete**=**"true"

**</**write**-**set**>**

**</**NsReadWriteSet**>**

**<**TxReadWriteSet**>**

Additionally, if the transaction performs a range query during simulation, the range query as well as its results will be added to the read-write set as query-info.

## Transaction validation and updating world state using read-write set

A committer uses the read set portion of the read-write set for checking the validity of a transaction and the write set portion of the read-write set for updating the versions and the values of the affected keys.

In the validation phase, a transaction is considered valid if the version of each key present in the read set of the transaction matches the version for the same key in the world state - assuming all the preceding valid transactions (including the preceding transactions in the same block) are committed (committed-state). An additional validation is performed if the read-write set also contains one or more query-info.

This additional validation should ensure that no key has been inserted/deleted/updated in the super range (i.e., union of the ranges) of the results captured in the query-info(s). In other words, if we re-execute any of the range queries (that the transaction performed during simulation) during validation on the committed-state, it should yield the same results that were observed by the transaction at the time of simulation. This check ensures that if a transaction observes phantom items during commit, the transaction should be marked as invalid. Note that the this phantom protection is limited to range queries (i.e., GetStateByRange function in the chaincode) and not yet implemented for other queries (i.e., GetQueryResult function in the chaincode). Other queries are at risk of phantoms, and should therefore only be used in read-only transactions that are not submitted to ordering, unless the [application](https://hyperledger-fabric.readthedocs.io/en/release-1.4/readwrite.html) can guarantee the stability of the result set between simulation and validation/commit time.

If a transaction passes the validity check, the committer uses the write set for updating the world state. In the update phase, for each key present in the write set, the value in the world state for the same key is set to the value as specified in the write set. Further, the version of the key in the world state is changed to reflect the latest version.

## Example simulation and validation

This section helps with understanding the semantics through an example scenario. For the purpose of this example, the presence of a key, k, in the world state is represented by a tuple (k,ver,val) where ver is the latest version of the key k having val as its value.

Now, consider a set of five transactions T1, T2, T3, T4, and T5, all simulated on the same snapshot of the world state. The following snippet shows the snapshot of the world state against which the transactions are simulated and the sequence of read and write activities performed by each of these transactions.

World state: (k1,1,v1), (k2,1,v2), (k3,1,v3), (k4,1,v4), (k5,1,v5)

T1 **->** Write(k1, v1'), Write(k2, v2')

T2 **->** Read(k1), Write(k3, v3')

T3 **->** Write(k2, v2'')

T4 **->** Write(k2, v2'''), read(k2)

T5 -> Write(k6, v6'), read(k5)

Now, assume that these transactions are ordered in the sequence of T1,..,T5 (could be contained in a single block or different blocks)

1. T1 passes validation because it does not perform any read. Further, the tuple of keys k1 and k2 in the world state are updated to (k1,2,v1'), (k2,2,v2')
2. T2 fails validation because it reads a key, k1, which was modified by a preceding transaction - T1
3. T3 passes the validation because it does not perform a read. Further the tuple of the key, k2, in the world state is updated to (k2,3,v2'')
4. T4 fails the validation because it reads a key, k2, which was modified by a preceding transaction T1
5. T5 passes validation because it reads a key, k5, which was not modified by any of the preceding transactions

**Note**: Transactions with multiple read-write sets are not yet supported.

# Gossip data dissemination protocol

Hyperledger Fabric optimizes blockchain network performance, security, and scalability by dividing workload across transaction execution (endorsing and committing) peers and transaction ordering nodes. This decoupling of network operations requires a secure, reliable and scalable data dissemination protocol to ensure data integrity and consistency. To meet these requirements, Fabric implements a **gossip data dissemination protocol**.

## Gossip protocol

Peers leverage gossip to broadcast ledger and channel data in a scalable fashion. Gossip messaging is continuous, and each peer on a channel is constantly receiving current and consistent ledger data from multiple peers. Each gossiped message is signed, thereby allowing Byzantine participants sending faked messages to be easily identified and the distribution of the message(s) to unwanted targets to be prevented. Peers affected by delays, network partitions, or other causes resulting in missed blocks will eventually be synced up to the current ledger state by contacting peers in possession of these missing blocks.

The gossip-based data dissemination protocol performs three primary functions on a Fabric network:

1. Manages peer discovery and channel membership, by continually identifying available member peers, and eventually detecting peers that have gone offline.
2. Disseminates ledger data across all peers on a channel. Any peer with data that is out of sync with the rest of the channel identifies the missing blocks and syncs itself by copying the correct data.
3. Bring newly connected peers up to speed by allowing peer-to-peer state transfer update of ledger data.

Gossip-based broadcasting operates by peers receiving messages from other peers on the channel, and then forwarding these messages to a number of randomly selected peers on the channel, where this number is a configurable constant. Peers can also [exercise](https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html) a pull mechanism rather than waiting for delivery of a message. This cycle repeats, with the result of channel membership, ledger and state information continually being kept current and in sync. For dissemination of new blocks, the **leader** peer on the channel pulls the data from the ordering service and initiates gossip dissemination to peers in its own organization.

## Leader election

The leader election mechanism is used to **elect** one peer per organization which will maintain connection with the ordering service and initiate distribution of newly arrived blocks across the peers of its own organization. Leveraging leader election provides the system with the ability to efficiently utilize the bandwidth of the ordering service. There are two possible modes of operation for a leader election module:

1. **Static** — a system administrator manually configures a peer in an organization to be the leader.
2. **Dynamic** — peers execute a leader election procedure to select one peer in an organization to become leader.

### Static leader election

Static leader election allows you to manually define one or more peers within an organization as leader peers. Please note, however, that having too many peers connect to the ordering service may result in inefficient use of bandwidth. To enable static leader election mode, configure the following parameters within the section of core.yaml:

peer:

*# Gossip related configuration*

gossip:

useLeaderElection: false

orgLeader: true

Alternatively these parameters could be configured and overridden with environmental variables:

export CORE\_PEER\_GOSSIP\_USELEADERELECTION**=**false

export CORE\_PEER\_GOSSIP\_ORGLEADER**=**true

**Note**

The following configuration will keep peer in **stand-by** mode, i.e. peer will not try to become a leader:

export CORE\_PEER\_GOSSIP\_USELEADERELECTION**=**false

export CORE\_PEER\_GOSSIP\_ORGLEADER**=**false

1. Setting CORE\_PEER\_GOSSIP\_USELEADERELECTION and CORE\_PEER\_GOSSIP\_ORGLEADER with true value is ambiguous and will lead to an error.
2. In static configuration organization admin is responsible to provide high availability of the leader node in case for failure or crashes.

### Dynamic leader election

Dynamic leader election enables organization peers to **elect** one peer which will connect to the ordering service and pull out new blocks. This leader is elected for an organization’s peers independently.

A dynamically elected leader sends **heartbeat** messages to the rest of the peers as an evidence of liveness. If one or more peers don’t receive **heartbeats** updates during a set period of time, they will elect a new leader.

In the worst case scenario of a network partition, there will be more than one active leader for organization to guarantee resiliency and availability to allow an organization’s peers to continue making progress. After the network partition has been healed, one of the leaders will relinquish its leadership. In a steady state with no network partitions, there will be **only** one active leader connecting to the ordering service.

Following configuration controls frequency of the leader **heartbeat** messages:

peer:

*# Gossip related configuration*

gossip:

election:

leaderAliveThreshold: 10s

In [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html) to enable dynamic leader election, the following parameters need to be configured within core.yaml:

peer:

*# Gossip related configuration*

gossip:

useLeaderElection: true

orgLeader: false

Alternatively these parameters could be configured and overridden with environment variables:

export CORE\_PEER\_GOSSIP\_USELEADERELECTION**=**true

export CORE\_PEER\_GOSSIP\_ORGLEADER**=**false

## Anchor peers

Anchor peers are used by gossip to make sure peers in different organizations know about each other.

When a configuration block that contains an update to the anchor peers is committed, peers reach out to the anchor peers and learn from them about all of the peers known to the anchor peer(s). Once at least one peer from each organization has contacted an anchor peer, the anchor peer learns about every peer in the channel. Since gossip communication is constant, and because peers always ask to be told about the existence of any peer they don’t know about, a common [view](https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html) of membership can be established for a channel.

For example, let’s assume we have three organizations—A, B, C— in the channel and a single anchor peer—peer0.orgC— defined for organization C. When peer1.orgA (from organization A) contacts peer0.orgC, it will tell it about peer0.orgA. And when at a later time peer1.orgB contacts peer0.orgC, the latter would tell the former about peer0.orgA. From that point forward, organizations A and B would [start](https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html) exchanging membership information directly without any assistance from peer0.orgC.

As communication across organizations depends on gossip in order to work, there must be at least one anchor peer defined in the channel configuration. It is strongly recommended that every organization provides its own set of anchor peers for high availability and redundancy. Note that the anchor peer does not need to be the same peer as the leader peer.

### External and internal endpoints

In order for gossip to work effectively, peers need to be able to obtain the endpoint information of peers in their own organization as well as from peers in other organizations.

When a peer is bootstrapped it will use peer.gossip.bootstrap in its core.yaml to advertise itself and exchange membership information, building a view of all available peers within its own organization.

The peer.gossip.bootstrap property in the core.yaml of the peer is used to bootstrap gossip **within an organization**. If you are using gossip, you will typically configure all the peers in your organization to point to an initial set of bootstrap peers (you can specify a space-separated list of peers). The internal endpoint is usually auto-computed by the peer itself or just passed explicitly via core.peer.address in core.yaml. If you need to overwrite this value, you can export CORE\_PEER\_GOSSIP\_ENDPOINT as an environment variable.

Bootstrap information is similarly required to establish communication **across organizations**. The initial cross-organization bootstrap information is provided via the “anchor peers” setting described above. If you want to make other peers in your organization known to other organizations, you need to set the peer.gossip.externalendpoint in the core.yaml of your peer. If this is not set, the endpoint information of the peer will not be broadcast to peers in other organizations.

To set these properties, issue:

export CORE\_PEER\_GOSSIP\_BOOTSTRAP**=<**a list of peer endpoints within the peer's org>

export CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT**=<**the peer endpoint, **as** known outside the org**>**

## Gossip messaging

Online peers indicate their availability by continually broadcasting “alive” messages, with each containing the **public key infrastructure (PKI)** ID and the signature of the sender over the message. Peers maintain channel membership by collecting these alive messages; if no peer receives an alive message from a specific peer, this “dead” peer is eventually purged from channel membership. Because “alive” messages are cryptographically signed, malicious peers can never impersonate other peers, as they lack a signing key authorized by a root certificate authority (CA).

In addition to the automatic forwarding of received messages, a state reconciliation process synchronizes **world state** across peers on each channel. Each peer continually pulls blocks from other peers on the channel, in order to repair its own state if discrepancies are identified. Because fixed connectivity is not required to maintain gossip-based data dissemination, the process reliably provides data consistency and integrity to the shared ledger, including tolerance for node crashes.

Because channels are segregated, peers on one channel cannot message or share information on any other channel. Though any peer can belong to multiple channels, partitioned messaging prevents blocks from being disseminated to peers that are not in the channel by applying message routing policies based on a peers’ channel subscriptions.

**Note**

1. Security of point-to-point messages are handled by the peer TLS layer, and do not require signatures. Peers are authenticated by their certificates, which are assigned by a CA. Although TLS certs are also used, it is the peer certificates that are authenticated in the gossip layer. Ledger blocks are signed by the ordering service, and then delivered to the leader peers on a channel.

2. Authentication is governed by the membership service provider for the peer. When the peer connects to the channel for the first time, the TLS session binds with the membership identity. This essentially authenticates each peer to the connecting peer, with respect to membership in the network and channel.