Next Consensus Architecture Proposal

Carlos Bruguera edited this page on Jul 22 · 27 revisions

Discuss and post comments here.

Authors: Elli Androulaki, Christian Cachin, Angelo De Caro, Konstantinos Christidis, Chet Murthy, Binh Nguyen, Alessandro Sorniotti, and Marko Vukolić

This page documents the architecture of a blockchain infrastructure with the roles of a blockchain node separated into roles of *peers* (who maintain state/ledger) and *consenters* (who consent on the order of transactions included in the blockchain state). In common blockchain architectures (including Hyperledger fabric as of July 2016) these roles are unified (cf. *validating peer* in Hyperledger fabric). The architecture also introduces *endorsing peers* (endorsers), as special type of peers responsible for simulating execution and *endorsing* transactions (roughly corresponding to executing/validating transactions in HL fabric 0.5-developer-preview).

The architecture has the following advantages compared to the design in which peers/consenters/endorsers are unified.

- **Chaincode trust flexibility.** The architecture separates *trust assumptions* for chaincodes (blockchain applications) from trust assumptions for consensus. In other words, the consensus service may be provided by one set of nodes (consenters) 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 consenters, 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 consensus 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 implementations.

Table of contents

Pages 25

Find a Page…

Home
Access Control Specification
Attribute Certificate Authority
Client SDK Specification
Consensus
Consensus work items for next Architecture proposal
Cumulative Run Time Silos in Go `pprof` CPU Profiles
Custom Events High level specification
Deterministic Transaction
Fabric and Golang Performance Portal
Fabric Next
Fabric Releases
Go Development Portal
How to Reduce CPU Run Time Spent in `runtime.morestack`
Legal

Show 10 more pages…

Clone this wiki locally

https://github.com/hyperled

Clone in Desktop

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: transactions are operations invoked on the chaincode and only chaincode changes the state. Transactions have to be "endorsed" and only endorsed transactions are 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 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) support for cross-chaincode transactions, b) optimizations for query (read-only) transactions.*

1.2. State

Blockchain state. The state of the blockchain ("world state") has a simple structure and 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, blockchain 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 `\bot`, which is in case of `N` the lowest element. Initially all keys are mapped to `(\bot,\bot)`. 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\in K` and `v\in 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 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 (state of the confidential chaincodes cannot be read in clear - see Section 6). *Support for cross-chaincode transactions, that modify the state belonging to two or more chaincodes will be added in future.*

Ledger. Evolution of blockchain state (history) is kept in a *ledger*. Ledger is a hashchain of blocks of transactions. Transactions in the ledger are totally ordered.

Blockchain state and ledger are further detailed in Section 4.

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.

2. **Peer**: a node that commits transactions and maintains the state and a copy of the ledger. Besides, peers can have two special roles:

   a. A **submitting peer** or **submitter**,

   b. An **endorsing peer** or **endorser**.

3. **Consensus-service-node** or **consenter**: a node running the communication service that implements a delivery guarantee (such as atomic broadcast) typically by running consensus.

Notice that consenters and clients do not maintain ledgers and blockchain state, only peers do.

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.

1.3.2. Peer

A peer communicates with the consensus service and maintain the blockchain state and the ledger. Such peers receive ordered state updates from the consensus service and apply them to the locally held state.

Peers can additionally take up one of two roles described next.

- **Submitting peer.** A *submitting peer* is a special role of a peer that provides an interface to clients, such that a client may connect to a submitting peer for invoking transactions and obtaining results. The peer communicates with the other blockchain nodes on behalf of one or more clients for executing the transaction.

- **Endorsing peer.** 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.

To emphasize a peer that does not also have a role of a submitting peer or an endorsing peer, such a peer is sometimes referred to as a *committing peer*.

1.3.3. Consensus service nodes (Consenters)

The *consenters* form the *consensus service*, i.e., a communication fabric that provides delivery guarantees. The consensus 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.

Peers are clients of the consensus service, to which the consensus service provides a shared *communication channel* offering a broadcast service for messages containing transactions. Peers connect to the channel and may send and receive messages on the channel. 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 (consensus channels). Consensus service may support multiple *channels* similar to the *topics* of a publish/subscribe (pub/sub) messaging system. Clients can connects 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. For simplicity, in the rest of this document and unless explicitly mentioned otherwise, we assume consensus service consists of a single channel/topic.

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

- `broadcast(blob)` : the submitting peer 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 consensus 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 consensus service. `deliver()` is also sometimes called `notify()` in pub-sub systems or `commit()` in BFT systems.

  Notice that consensus service clients (i.e., peers) interact with the service only through `broadcast()` and `deliver()` events.

Consensus properties. The guarantees of the consensus service (or atomic-broadcast channel) are as follows. They answer the following question: *What happens to a broadcasted message and what relations exist among delivered messages?*

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 consensus 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 previous `deliver()` event (`prevhash`). When the consensus 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 consensus output, as discussed in Sections 4 and 5 later. In the special case of the first `deliver()` event, `prevhash` has a default value.

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

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

To summarize, the consensus 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 a consensus 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 peer 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 a submitting peer of its choice

To invoke a transaction, the client sends the following message to a submitting peer `spID`.

`<SUBMIT,tx,retryFlag>`, where

- `tx=<clientID,chaincodeID,txPayload,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,
  - `clientSig` is signature of a client on other fields of `tx`.
- `retryFlag` is a boolean that tells the submitting peer whether to retry the submission of the transaction in case transaction fails,

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 one field

- `invocation = <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 two fields

- `chainCode = <source, metadata>`, 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

TODO: Decide whether to include explicitly local/logical time at the client (a timestamp).

2.2. The submitting peer prepares a transaction and sends it to endorsers for obtaining an endorsement

On reception of a `<SUBMIT,tx,retryFlag>` message from a client, the submitting peer first verifies the client's signature `clientSig` and then prepares a transaction. This involves submitting peer tentatively *executing* a transaction (`txPayload`), by invoking the chaincode to which the transaction refers (`chaincodeID`) and the copy of the state that the submitting peer locally holds.

As a result of the execution, the submitting peer computes a *state update* (`stateUpdate`) and *version dependencies* (`verDep`), also called *MVCC+postimage info* in DB language.

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

- `verDep` is a tuple `verDep=(readset,writeset)`. Given state `s` before a submitting peer executes a transaction:
  - for every key `k` read by the transaction, pair `(k,s(k).version)` is added to `readset`.

    - for every key `k` modified by the transaction, pair `(k,s(k).version)` is added to `writeset`.
- additionally, for every key `k` modified by the transaction to the new value `v'`, pair `(k,v')` is added to `stateUpdate`. Alternatively, `v'` could be the delta of the new value to previous value (`s(k).value`).

An implementation may combine `verDep.writeset` with `stateUpdate` into a single data structure.

Then, `tran-proposal := (spID,chaincodeID,txContentBlob,stateUpdate,verDep)`,

where `txContentBlob` is chaincode/transaction specific information. The intention is to have `txContentBlob` used as some representation of `tx` (e.g., `txContentBlob=tx.txPayload`). More details are given in Section 6.

Cryptographic hash of `tran-proposal` is used by all nodes as a unique transaction identifier `tid` (i.e., `tid=HASH(tran-proposal)`).

The submitting peer then sends the transaction (i.e., `tran-proposal`) to the endorsers for the chaincode concerned. The endorsing peers are selected according to the interpretation of the policy and the availability of peers, known by the peers. 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 peer tries to satisfy the policy expression with the endorsers available.

The submitting peer `spID` sends the transaction to an endorsing peer `epID` using the following message:

`<PROPOSE,tx,tran-proposal>`

Possible optimization: An implementation may optimize duplication of `chaincodeID` in `tx.chaincodeID` and `tran-proposal.chaincodeID`, as well as possible duplication of `txPayload` in `tx.txPayload` and `tran-proposal.txContentBlob`.

Finally, the submitting peer stores `tran-proposal` and `tid` in memory and waits for responses from endorsing peers.

Alternative design: *As described here the submitting peer communicates directly with the endorsers. This could also be a function performed by the consensus service; in this case it should be determined whether fabric has to follow atomic broadcast delivery guarantee for this or use simple peer-to-peer communication. In that case the consensus service would also be responsible to collect the endorsements according to the policy and to return them to the submitting peer.*

TODO: Decide on communication between submitting peers and endorsing peers: peer-to-peer or using the consensus service.

2.3. An endorser receives and endorses a transaction

When a transaction is delivered to a connected endorsing peer for the chaincode `tran-proposal.chaincodeID` by means of a `PROPOSE` message, the endorsing peer performs the following steps:

- The endorser verifies `tx.clientSig` and ensures `tx.chaincodeID==tran-proposal.chaincodeID`.

- The endorser simulates the transaction (using `tx.txPayload`) and verifies that the state update and dependency information are correct. If everything is valid, the peer digitally signs the statement `(TRANSACTION-VALID, tid)` producing signature `epSig`. The endorsing peer then sends `<TRANSACTION-VALID, tid,epSig>` message to the submitting peer (`tran-proposal.spID`).

- Else, in case the transaction simulation at endorsers fails to produce results from `tran-proposal`, we distinguish the following cases:

  a. if the endorser obtains different state updates than those in `tran-proposal.stateUpdates`, the peer signs a statement `(TRANSACTION-INVALID, tid, INCORRECT_STATE)` and sends the signed statement to the submitting peer.

  b. if the endorser is aware of more advanced data versions than those referred to in `tran-proposal.verDeps`, it signs a statement `(TRANSACTION-INVALID, tid, STALE_VERSION)` and sends the signed statement to the submitting peer.

  c. if the endorser does not want to endorse a transaction for any other reason (internal endorser policy, error in a transaction, etc.) it signs a statement `(TRANSACTION-INVALID, tid, REJECTED)` and sends the signed statement to the submitting peer.

Notice that an endorser does not change its state in this step, the updates are not logged!

Alternative design: *An endorsing peer may omit to inform the submitting peer about an invalid transaction altogether, without sending explicit `TRANSACTION-INVALID` notifications.*

Alternative design: *The endorsing peer submits the `TRANSACTION-VALID` / `TRANSACTION-INVALID` statement and signature to the consensus service for delivery.*

TODO: Decide on alternative designs above.

2.4. The submitting peer collects an endorsement for a transaction and broadcasts it through consensus

The submitting peer waits until it receives enough messages and signatures on `(TRANSACTION-VALID, tid)` statements to conclude that the transaction proposal is endorsed (including possibly its own signature). This will 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 signatures from endorsing peers which establish that a transaction is endorsed is called an *endorsement*, the peer stores them in `endorsement`.

If the submitting peer does not manage to collect an endorsement for a transaction proposal, it abandons this transaction and notifies the submitting client. If `retryFlag` has been originally set by the submitting client (see step 1 and the `SUBMIT` message) the submitting peer may (depending on submitting peer policies) retry the transaction (from step 2).

For transaction with a valid endorsement, we now start using the consensus-fabric. The submitting peer invokes consensus service using the `broadcast(blob)`, where `blob=(tran-proposal, endorsement)`.

2.5. The consensus 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. (This step might be performed even without waiting for applying the state updates with sequence numbers smaller than `seqno` .)

- It verifies that the dependencies (`blob.tran-proposal.verDep`) have not been violated meanwhile.

Verification of dependencies can be implemented in different ways, according to a consistency property or "isolation guarantee" that is chosen for the state updates. For example, **serializability** can be provided by requiring the version associated with each key in the `readset` or `writeset` to be equal to that key's version in the state, and rejecting transactions that do not satisfy this requirement. As another example, one can provide **snapshot isolation** when all keys in the `writeset` still have the same version as in the state as in the dependency data. The database literature contains many more isolation guarantees.

TODO: Decide whether to insist on serializability or allow chaincode to specify isolation level.

- If all these checks pass, the transaction is deemed *valid* or *committed*. This means that a peer appends the transaction to the ledger and subsequently applies `blob.tran-proposal.stateUpdates` to blockchain state. Only committed transactions may change the state.

- If any of these checks fail, the transaction is invalid and the peer drops the transaction. It is important to note that invalid transactions are not committed, do not change the state, and are not recorded.

Additionally, the submitting peer notifies the client of a dropped transaction. If `retryFlag` has been originally set by the submitting client (see step 1 and the `SUBMIT` message) the submitting peer may (depending on submitting peer policies) retry the transaction (from step 2).

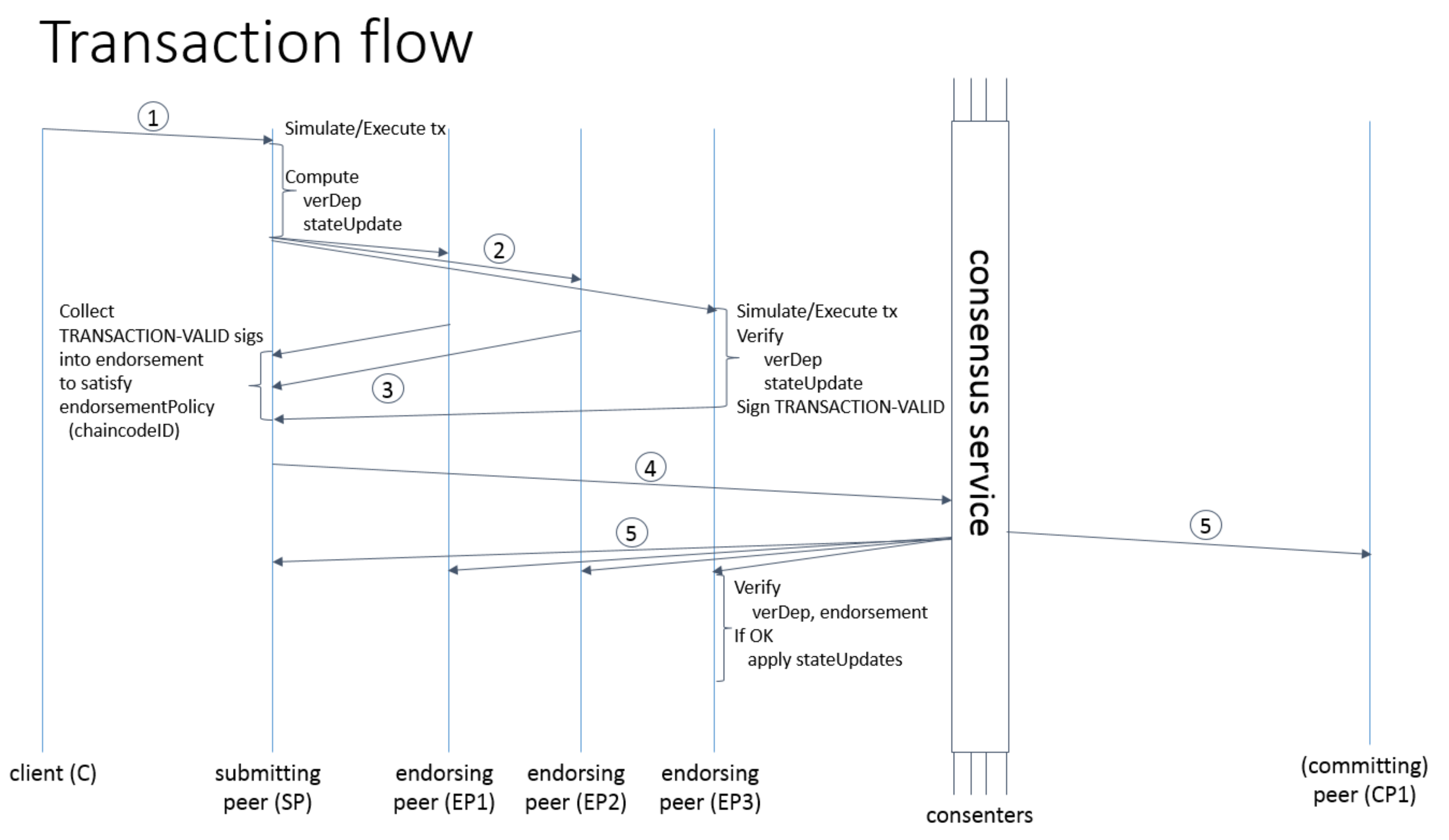

Figure 1. Illustration of the transaction flow (common-case path).

3. Endorsement policies

3.1. Endorsement policy specification

An **endorsement policy**, is a condition on what *endorses* a transaction. An endorsement policy is specified by a `deploy` transaction that installs specific chaincode. 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 peer and endorsing peers as explained in Section 2.

Formally the endorsement policy is a predicate on the transaction, 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).

Formally an endorsement policy is a predicate referring 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 transaction itself;
4. and potentially more.

The evaluation of an endorsement policy predicate must be deterministic. Endorsement policies must not be complex and cannot be "mini chaincode". The endorsement policy specification language must be limited and enforce determinism.

The 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.

TODO: Decide on parameters of the endorsement policy.

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 all members of E.
- A valid signature from any single member of E.
- Valid signatures from endorsing peers according to the condition `(Alice OR Bob) AND (any two of: Charlie, Dave, Eve, Frank, George)`.
- Valid signatures 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).

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.

3.2. Implementation

Typically, endorsement policies will be formulated in terms of signatures required from endorsing peers. The metadata of the chaincode must contain the corresponding signature verification keys.

An endorsement will typically consist of a set of signatures. It can be evaluated locally by every peer or by every consenter node with access to the chaincode metadata (which includes these keys), such that this node does *not* require interaction with another node. Neither does a node need access to the state for verifying an endorsement.

Endorsements that refer to other metadata of the chaincode can be evaluated in the same way.

TODO: Formalize endorsement policies and design an implementation.

---

4. Blockchain data structures

The blockchain consists of three data structures: a) *raw ledger*, b) *blockchain state* and c) *validated ledger*). Blockchain state and validated ledger are maintained for efficiency - they can be derived from the raw ledger.

- *Raw ledger (RL)*. The raw ledger contains all data output by the consensus service at peers. It is a sequence of `deliver(seqno, prevhash, blob)` events, which form a hash chain according to the computation of `prevhash` described before. The raw ledger contains information about both *valid* and *invalid* transactions and provides a verifiable history of all successful and unsuccessful state changes and attempts to change state, occurring during the operation of the system.

  The RL allows peers to replay the history of all transactions and to reconstruct the blockchain state (see below). It also provides submitting peer with information about *invalid* (uncommitted) transactions, on which submitting peers can act as described in Section 2.5.

- *(Blockchain) state*. The state is maintained by the peers (in the form of a KVS) and is derived from the raw ledger by **filtering out invalid transactions** as described in Section 2.5 (Step 5 in Figure 1) and applying valid transactions to state (by executing `put(k,v)` for every `(k,v)` pair in `stateUpdate` or, alternatively, applying state deltas with respect to previous state).

  Namely, by the guarantees of consensus, 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 of a state update (Section 2.5) 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.

- *Validated ledger (VL)*. 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 raw ledger, maintain the *validated ledger*. This is a hash chain derived from the raw ledger by filtering out invalid transactions.

4.1. Batch and block formation

Instead of outputting individual transactions (blobs), the consensus service may output *batches* of blobs. In this case, the consensus service must impose and convey a deterministic ordering of the blobs within each batch. The number of blobs in a batch may be chosen dynamically by a consensus implementation.

Consensus batching does not impact the construction of the raw ledger, which remains a hash chain of blobs. But with batching, the raw ledger becomes a hash chain of batches rather than hash chain of individual blobs.

With batching, the construction of the (optional) validated ledger *blocks* proceeds as follows. As the batches in the raw ledger 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 in a batch becomes committed in a block. Every peer does this by itself. A block is defined as a consensus batch without the invalid transactions, that have been filtered out. Such blocks are inherently dynamic in size and may be empty. An illustration of block construction is given in the figure below.

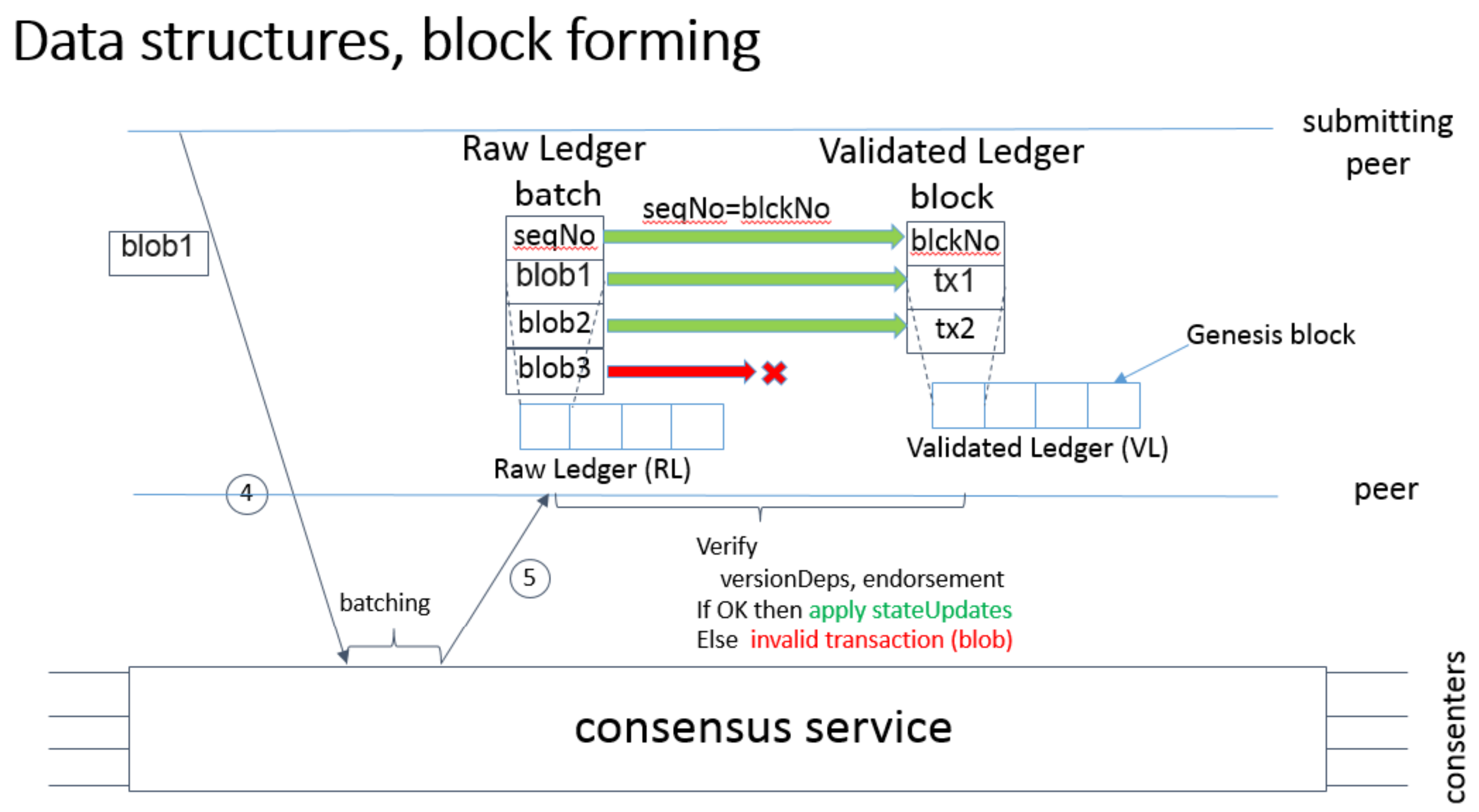

Figure 2. Illustration of validated ledger block formation from raw ledger batches.

4.2. Chaining the blocks

The batches of the raw ledger output by the consensus service form a hash chain, as described in Section 1.3.3.

The blocks of the validated ledger are chained together to a hash chain by every peer. The valid and committed transactions from the batch form a block; all blocks are chained together to form a hash chain.

More specifically, every block of a validated ledger contains:

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

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

5. State transfer and checkpointing

In the common case, during normal operation, a peer receives a sequence of `deliver()` events (containing batches of transactions) from the consensus service. It appends these batches to its raw ledger and updates the blockchain state and the validated ledger accordingly.

However, due to network partitions or temporary outages of peers, a peer may miss several batches in the raw ledger. In this case, a peer must *transfer state* of the raw ledger from other peers in order to catch up with the rest of the network. This section describes a way to implement this.

5.1. Raw ledger state transfer (batch transfer)

To illustrate how basic *state transfer* works, assume that the last batch in a local copy of the raw ledger at a peer *p* has sequence number 25 (i.e., `seqno` of the last received `deliver()` event equals `25`). After some time peer *p* receives `deliver(seqno=54,hash53,blob)` event from the consensus service.

At this moment, peer *p* realizes that it misses batches 26-53 in its copy of the raw ledger. To obtain the missing batches, *p* resorts to peer-to-peer communication with other peers. It calls out and asks other peers for returning the missing batches. While it transfers the missing state, *p* continues listening to the consensus service for new batches.

Notice that *p does not need to trust any of its peers* from which it obtains the missing batches via state transfer. As *p* has the hash of the batch *53* (namely, *hash53*), which *p* trusts since it obtained that directly from the consensus service, *p* can verify the integrity of the missing batches, once all of them have arrived. The verification checks that they form a proper hash chain.

As soon as *p* has obtained all missing batches and has verified the missing batches 26-53, it can proceed to the steps of section 2.5 for each of the batches 26-54. From this *p* then constructs the blockchain state and the validated ledger.

Notice that *p* can start to speculatively reconstruct the blockchain state and the validated ledger as soon as it receives batches with lower sequence numbers, even if it still misses some batches with higher sequence numbers. However, before externalizing the state and committing blocks to the validated ledger, *p* must complete the state transfer of all missing batches (in our example, up to batch 53, inclusively) and processing of individual transferred batches as described in Section 2.5.

5.2. Checkpointing

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

To facilitate pruning of the raw ledger, this document describes a *checkpointing* mechanism. This mechanism establishes the validity of the validated-ledger blocks across the peer network and allows checkpointed validated-ledger blocks to replace the discarded raw-ledger batches. 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 from the raw ledger, but simply replay the state updates contained in the validated ledger).

Notice that checkpointing facilitates pruning of the raw ledger and, as such, it is only performance optimization. Checkpointing is not necessary to make the design correct.

5.2.1. 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,peerSig>`, where `blockno` is the current blocknumber and `blocknohash` is its respective hash, and `peerSig` is peer's signature on `(CHECKPOINT,blocknohash,blockno)`, referring to the validated ledger.

A peer collects `CHECKPOINT` messages until it obtains enough correctly signed messages with matching `blockno` and `blocknohash` to establish a *valid checkpoint* (see Section 5.2.2.).

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`.
- (optionally) prunes its raw ledger up to batch number `blockno` (inclusive).

5.2.2. Valid checkpoints

Clearly, the checkpointing protocol raises the following questions: *When can a peer prune its Raw Ledger? 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 *7* different peers.
    - in a deployment where every consenter is also a peer and where up to *f* consenters may be (Byzantine) faulty, each peer may trust a checkpoint if confirmed by *f+1* different consenters.

5.2.3. Validated ledger state transfer (block transfer)

Besides facilitating the pruning of the raw ledger, checkpoints enable state transfer through validated-ledger block transfers. These can partially replace raw-ledger batch transfers.

Conceptually, the block transfer mechanism works similarly to batch transfer. Recall our example in which a peer *p* misses batches 26-53, and is transferring state from a peer *q* that has established a valid checkpoint at block 50. State transfer then proceeds in two steps:

- First, *p* tries to obtain the validated ledger up to the checkpoint of peer *q* (at block 50). To this end, *q* sends to *p* its local (`latestValidCheckpoint,latestValidCheckpointProof`) pair. In our example `latestValidCheckpoint=(hash50,block50)`. If `latestValidCheckpointProof` satisfies the checkpoint validity policy at peer *p*, then the transfer of the blocks 26 to 50 is possible. Otherwise, peer *q* cannot convince *p* that its local checkpoint is valid. Peer *p* might then choose to proceed with raw-ledger batch transfer (Section 5.1).

- If the transfer of blocks 26-50 was successful, peer *p* still needs to complete state transfer by fetching blocks 51-53 of the validated ledger or batches 51-53 of the raw ledger. To this end, *p* can simply follow the Raw-ledger batch transfer protocol (Section 5.1), transferring these from *q* or any other peer. Notice that the validated-ledger blocks contain hashes of respective raw-ledger batches (Section 4.2). Hence the batch transfer of the raw ledger can be done even if peer *p* does not have batch 50 in its Raw Ledger, as hash of batch 50 is included in block 50.

6. Confidentiality

This section explains how this architecture facilitates the deployment of chaincodes that involve processing of sensitive data that must be kept confidential from certain peers.

Fabric-level confidentiality policies. In a nutshell, this architecture offers certain confidentiality features at the *fabric* layer, where:

- endorsers of a confidential chaincode have access to the plaintext of:
    - the chaincode deploy transaction payload;
    - the chaincode invoke transaction payload;
    - the chaincode state and state updates; and
- other peers are prevented from accessing plaintext of this information.

Here we make the assumption that the endorsers included in the endorsement set of a confidential chaincode are trusted by the chaincode creator to access the resources of a chaincode and to maintain their confidentiality.

The degree to which a chaincode employs the fabric-confidentiality features is defined in a **confidentiality policy** specified by the deployer at deployment time. In particular, the fabric offers support for the following confidentiality policies:

Policy ID	Deploy Payload	Invoke Payload	State Updates
Policy `000`	In-the-clear	In-the-clear	In-the-clear
Policy `010`	In-the-clear	Confidential	In-the-clear
Policy `011`	In-the-clear	Confidential	Confidential
Policy `110`	Confidential	Confidential	In-the-clear
Policy `111`	Confidential	Confidential	Confidential
Policy* `0xx`	In-the-clear	any	any
Policy* `1xx`	Confidential	any	any

Table 6.1. Fabric confidentiality policies.

Confidential means that access to the cleartext of the corresponding transaction component (i.e., deploy/invoke payload or state updates) is restricted to the endorsing peers of the transaction. *In-the-clear* means that the corresponding transaction component can be read and accessed by all peers. *Any* denotes either *Confidential* or *In-the-clear*.

In the following we assume that the payload of a deploy transaction includes both code and application data/metadata as a single item. However, in future revisions of this design, we may treat the two independently from a confidentiality policy perspective.

In the rest of this text, a `Confidential` *chaincode* is one with a confidentiality policy different from `000`. Also, in the rest of this text, we make the assumption that every peer is associated with an enrollment (encryption) public key, as described in Hyperledger fabric Protocol specification. In particular, for every endorser `e` a public key `ePubKey` is known to all peers and peer `e` knows the corresponding private key for decryption.

Disclaimers:

Hiding the transaction activity w.r.t. a chaincode. It is important to note that the design does not hide the identifiers of the chaincode for which transactions are executed nor does it hide which parts of chaincode state are updated by a transaction. In other words, transactions and state are encrypted but activity and state changes can be linked by the committing peers. With respect to third parties, the committing peers are permitted by the chaincode creators to notice activity of a chaincode and trusted to not leak this information. However, we aim to remedy this in a revised version of this design.

Granularity of the confidentiality of the state. This design currently treats the a chaincode and its state as one confidentiality domain, not distinguishing different key/value entries. It is possible to assign different confidentiality policies to different parts of the state at the application layer, supporting goals such as some parts of the state need to be visible to a subset of the clients, but other parts not. That is, such guarantees are not prevented by the architecture and can already be implemented at the application level, using adequate cryptographic tools in one or more chaincodes. This would result in *chaincode-level confidentiality*. Other requirements, such as hiding the identities of endorsers and/or the endorsement policy from the committing peers will be tackled at future iteration of this design.

6.1 Confidential chaincode deployment

6.1.1 Creating a deploy transaction

To deploy chaincode `cc` with confidentiality support, the client that deploys `cc` (the *deployer* of `cc`) specifies:

- the chaincode itself, including the source code, and metadata associated to it, subsumed in `chainCode`;
- the policies accompanying the endorsement of transactions associated to that chaincode, i.e.,
  - an endorsement policy, denoted `ccEndorsementPolicy`;
  - a set of endorsers, denoted `ccEndorserSet`;
  - a confidentiality policy `ccConfidentialityPolicy` for the chaincode, specifying confidentiality levels as described in Table 6.1.;
- cryptographic material associated to the chaincode, i.e., an asymmetric encryption key pair `ccPubKey/ccPrivKey`, intended to provide confidentiality of the transactions, state, and state updates of this particular chaincode.

This information is incorporated into a (deploy) transaction `tx` that is subsequently included in a `SUBMIT` message passed to the submitting peer. The client constructs `txPayload` of the deploy transaction of `cc` as follows.

The deployer first sets `txPayload.policies` to <`ccEndorsementPolicy`, `ccEndorserSet`, `ccConfidentialityPolicy`>.

To fill in `txPayload.payload`, it first checks if `ccConfidentialityPolicy` specifies `Deploy Payload = Confidential`. If so, then the deployer encrypts `chainCode` using `ccPubKey`, so that the chaincode and its metadata will only be accessible by peers entitled to see them. That is, `txPayload.chainCode := Enc(ccPubKey, chainCode)`.

Furthermore, the chaincode-specific decryption key `ccPrivKey` is distributed to all endorsing peers of chaincode `cc` (i.e., endorsers in `ccEndorserSet`). This is done by wrapping `ccPrivKey` under the public key of `e` for every endorser `e` in `ccEndorserSet`, `wrappedKey_e := Enc(ePubKey, ccPrivKey)`, where `ePubKey` is the enrollment public key of `e`. Thus, the deployer creates an additional field `txPayload.endorserMessages := ccEndorserMessages`, where `ccEndorserMessages` includes `wrappedKey_e` for all `e` in `ccEndorserSet`.

We emphasize that a deploy transaction may include more fields, that are intentionally omitted here in favor of presentation simplicity. In addition, for sending the wrapped chaincode-key to all endorsers, and for actually encrypting `chainCode` hybrid encryption schemes could be used to achieve better performance. More details can be provided in future versions of this document.

Chaincode `cc` is assigned an identifier, hereafter referred to as `chaincodeID`, as described in Section 2.2, for instance, as a hash of the deploy transaction `tx`. We assume here that it is unique per chaincode and it may become known to all peers.

Endorsement of Deploy Transaction: Recall that every deploy transaction is treated as an invoke transaction of a system chaincode handling chaincode deployments; let this chaincode be denoted by `dsc`, and the respective endorsement policy and set of endorsers are `dscEndorsementPolicy` and `dscEndorserSet`. This means that the deploy transaction of any chaincode, needs to be endorsed according to `dscEndorsementPolicy`.

TODO: Consider whether for confidential chaincodes, the endorsement policy of deploy transaction itself should satisfy the endorsement policy of the chaincode being deployed, that is, whether the deploy transaction of `cc` should also satisfy `ccEndorsementPolicy`. Alternatively, see if it makes sense to split a deploy transaction into two parts, e.g., a `deploy` (that only states the deploy info) and an `install` that sets up the running chaincode.

6.1.2 Peer processes a deploy transaction

When a peer `e` commits a deploy transaction for chaincode `cc`, then it has at least access to `chaincodeID` and to `txPayload.policies`, which contains `ccConfidentialityPolicy`, `ccEndorsementPolicy`, and `ccEndorserSet`.

If `e` is also an endorser for `cc` then in addition `e` obtains the following values:

- the secret decryption key of the chaincode, as `ccPrivKey := Dec(ePrivKey, wrappedKey_e)`;
- the cleartext of the deployment payload, as `chainCode := Dec(ccPrivKey, txPayload.chainCode)`, if `Deploy Payload = Confidential`.

Given the in plaintext deployment payload `chainCode`, and prior to actually installing it, the peer performs a consistency check as described below. The peer then uses cleartext `chainCode` in the remainder of the deployment procedure, as described in Section 2.

Consistency: For deploying a chaincode, the protocol should ensure that every endorser `e` will actually install and run the same code, even if the creator of the chaincode (the peer that submits the deploy transaction) would intend to make endorsers run different code. To this effect, `e` should run a verification step during the execution of the deploy transaction, which ensures, roughly, the following. The deployment transaction either *succeeds* and the peer outputs the chaincode `cc` to be executed or the transaction *fails* and outputs a corresponding error. The deployment ensures the following condition: If two deployment succeeds for two distinct [correct, non-faulty] endorsers, then they both deploy the same chaincode. This condition is equivalent to the *Consistency* property of a Byzantine Consistent Broadcast CGR11; Sec. 3.10(http://distributedprogramming.net).

Since every endorser executes the deployment transaction as a result of obtaining it from the consensus service, all endorsers receive the same `chainCode` blob, which may contain encryptions. But when an endorser decrypts that with its own key, it does not automatically guarantee that the resulting code of `cc` is the same for every other endorser.

This can be addressed in multiple ways: One solution would be to use a specialized multi-receiver encryption scheme that includes randomness in the ciphertext. Alternatively, verifiable encryption can be used with a zero-knowledge proof that the plaintext for all receivers are equal (this seems less efficient than the first option). In any case, this additional data that ensures the consistency condition must be included by the deployer in the deploy transaction `tx`.

TODOs: Many details are not yet addressed, such as:

- Give more information on the implementation of `dsc` (perhaps in a different section). More specifically, information on:
    - the `dsc` code itself, and its `dscEndorsmentPolicy` implementation (ref. section 2.4)
    - details of `tran-proposal` of `dsc`. Perhaps in a separate section?
- Consider alternative implementations for the identifier of the deployed chaincode (`chaincodeID`).
- Describe how to react on detecting a violation of the consistency property above, for example, because the creator provided the wrong wrapped keys to endorsers.

6.2 Confidential chaincode invocation

Invoke transactions for confidential chaincode will comply with the *static* confidentiality policy that has been specified at deployment time (in `ccConfidentialityPolicy`). Future revisions may consider possibilities of *dynamic* confidentiality policy that may evolve during the runtime of the system.

A confidential chaincode is invoked similarly to any other chaincode. The difference is that here a submitting peer of a transaction pertaining to confidential chaincode needs to be an endorser for that chaincode. This means it has access to the keys protecting that chaincode and its state. To help maintain stateless clients, every peer knows which peers are endorsers for a given confidential chaincode (see Section 6.1, `ccEndorserSet`), and can direct clients to an appropriate submitting peer. Hence, in the rest of this section, we assume that a submitting peer is also an endorser.

6.2.1 Creating and submitting a confidential transaction

The client is aware of the chaincode for which it creates a transaction, and of its endorsers. The `SUBMIT` message of a confidential transaction invocation consists of the same fields as non-confidential (see Section 2.1.), i.e., `<SUBMIT, tx, retryFlag>`, where `tx=<clientID,chaincodeID,txPayload,clientSig>`, and where `clientID` is some form of identification of the client at the fabric layer, e.g., a transaction certificate, `clientSig` is signature of a client on other fields of `tx`.

Notice that for security purposes, `tx` may also consist of more fields, that are intentionally omitted here in favor of presentation simplicity.

The difference with respect to non-confidential transactions is as follows. If the confidentiality policy associated to the chaincode (`ccConfidentialityPolicy`) specifies `Invoke Payload = Confidential`, then the client additionally encrypts the invocation arguments, and metadata, say invocation with ccPubKey into txPayload.invocation. That is, txPayload.invocation := Enc(ccPubKey, invocation).

Again, hybrid encryption schemes can be used for better performance. Also keys defined at deployment time can be used to generate other keys, e.g., keys the key state would need to be encrypted with to reduce the overall number of keys that need to be managed/distributed.

TODO: Optionally provide a customized encryption method that also hides the chaincode identifier.

6.2.2 Endorsing a confidential transaction

Upon receiving and verifying <SUBMIT, tx, retryFlag>, for tentatively executing the code associated to the transaction, and for preparing the transaction to be sent to the consensus service, the submitting peer first decrypts a confidential transaction payload. More precisely, if ccConfidentialityPolicy specifies that Invoke Payload is Confidential, the submitting peer first retrieves the corresponding chaincode-specific decryption key ccPrivKey and then decrypts txPayload.invocation. Recall that the submitting peer is assumed to be an endorser for the chaincode. The peer, say e, may retrieve ccPrivKey from the deploy transaction of the chaincode chaincodeID and obtain ccPrivKey through wrapped_e. Then the peer computes invocation := Dec(ccPrivKey, txPayload.invocation).

With the operation and metadata in invocation the submitting peer tentatively executes the transaction using its copy of the state for producing a transaction proposal. If the confidentiality policy of the chaincode specifies that State is also Confidential, then the peer uses ccPrivKey to access the state, decrypting state values while reading.

Moreover, for the state updates when the confidentiality policy specifies that State = Confidential, the submitting peer encrypts new state values in stateUpdates using ccPubKey. With the state in the form of key/value pairs, only the changed values are encrypted. The version dependencies are not encrypted.

The transaction-proposal from the submitting peer now consists of

tran-proposal := (spID, chaincodeID, txContentBlob, stateUpdates, verDep),

where txContentBlob is some form of invoke transaction tx, as submitted by the client.

The submitting peer creates a PROPOSE message to send to the rest of endorsers in the endorser set as before (Section 2.2):

<PROPOSE, tx, tran-proposal>.

Notice that is imperative that endorsers check that the chaincodeID that appears in tx is consistent with the content of tran-proposal. Endorsers follow the same process as before to endorse a transaction.

In summary, this mechanism ensures that in case the state is encrypted, the endorsers for a chaincode have access to the state in the clear but other peers have not. Once the stateUpdates is applied to the state after the consensus service has delivered it to a peer, every peer updates its state. Note that also the endorsers of a chaincode apply the updates and transparently operate on the ciphertext; they only need access to the plaintext again for endorsing a next transaction.

TODO: Revisit section 4 and describe in more detail which parts are added to the ledger.