**Hyperledger Fabric - Private Channel Data (version2)**

**Proposed design for: https://jira.hyperledger.org/browse/FAB-1151**

**Authors: David Enyeart, Artem Barger, Manish Sethi, Senthil Natarajan, Troy Ronda**

We would like to maintain data such that only its evidence is exposed to the chain, ordering service, and channel peers while the data itself is disseminated to specific peers based on policy. The disseminated data, kept private to the peers, remains validated against the evidence in the chain. In this way, we can achieve finer-grained data confidentiality for transactions while still maintaining ledger consistency and still being able to leverage Fabric for both data evidence and dissemination of the data. As such, there are a class of requirements where data on the ledger should remain *private* from the ordering service and a subset of the peers on the network (channel). Only a hash of the data would be included in the transaction on the chain.

This paper first describes the assumptions on the requirements. Second, we present the proposed solution. Third, we extend the proposed solution to partition and encrypt data between peers on the network (refer to Sections 3 and 4).

# 1. Assumptions

1. Keep certain private data from chain and ordering service, with evidence (hash) on-chain.
2. Private data is accessible to all peers on the channel (assumption for core solution in Section 2, assumption is revised in Section 3 to allow partitioning of private data).
3. Chaincode differentiates between private and normal data such that it can use the proposed APIs for private data management.
4. Private value can be queried/modified/deleted by chaincode APIs as a part of a transaction.
5. Separate set of chaincode APIs for private data, but chaincode normal data APIs and private data APIs are functionally equivalent from chaincode author perspective.
6. Historical values of the private value need to be added to history database on the peers that are authorized to the private data, to support GetHistoryForKey() of private data. FUTURE.
7. Need ability to purge private data based on time-to-live.

# 2. Typical Scenarios

A channel contains 3 orgs: org0, org1 and org2. Each org have 2 peers (e.g., org0-p0, org0-p1) joining the channel.

Scenario 1): In the same channel, only org0 and org1 can access the real data;

Scenario 2): In the same channel and the same org, only subsets of the peer can access the real data;

# 2. Proposed Solution

We propose an approach to guarantee privacy of sensitive data from both Orderer and Chain, by storing private data off-chain in a set of private/side DBs. We present steps involved in our approach for the following scenarios:

(i) Storing private data.

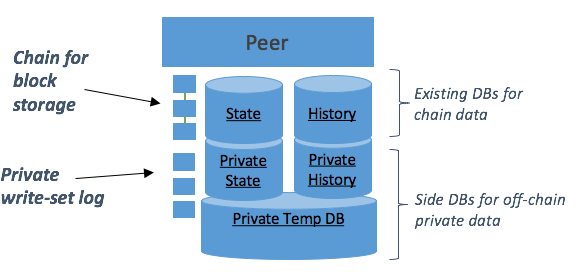
(ii) Querying private data.

(iii) Deleting private data.

(iv) Savepoints and Recovering private data.

(v) Purging private data.

Private data APIs are functionally consistent with regular chaincode data APIs. e.g. stub.privateData.PutState(collection, privateKey,privateValue), where collection is a namespace to partition data (see Section 3 for data partitioning). In addition to the normal data structures, there will be corresponding data structures for private/side DBs:



*Figure 1. Peer with data structure for private data storage*

**(1) Private Temp DB** - stores transient (uncommitted) private read write sets for transactions ‘on the side’, between endorsement time and commit time. Keyed by txid.

**(2) Private write-set log** - Primary storage for committed private write sets, a transaction log of private write sets keyed by blockNum or (blockNum:tranNum) to assist in state transfer alongside blocks (which is the transaction log for public data).

**(3) Private State DB** - Similar to state DB - stores latest version of committed private keys/values. Used by chaincode APIs. Can be rebuilt from private write-set log, just like the normal state db can be rebuilt from normal block storage. Keyed by chaincodeid:key, same as normal state DB.

**(4) Private History DB** - stores history of committed private value updates of a key (pointer to private write-set log blockNum:tranNum). Keyed by chaincodeid:key:blockNum:tranNum, same as normal history DB. Used by GetHistoryForKey chaincode API.

Private state db is structured like normal state db, and private history db is structured like normal history db, in support of the corresponding chaincode APIs.

If state db is configured to be CouchDB, the private state db will also be CouchDB. That is, if rich query from chaincode is enabled against state db, it will also be enabled against private state db. In v1, history db is always on LevelDB (that is, no rich query against historic values). In the diagram, the private state/history database is displayed as a separate logical database from the public state/history database. The implementation may utilize the same physical database with namespacing to keep the public and private data distinct.

Next, we present the proposed handling for each of the scenarios.

## 2.1. Storing private data

***Figure 2: Endorsement flow***

*Client sends Transaction Proposal, Simulation at peer, Private data stored in Temp DB and gossipped, Proposal Response returned to client.*

### Endorsement phase

Private data is passed from client via TransientMap data field in proposal message (check that transient values are not included in signatures, so that signatures persisted into the chain can be validated without having this value). Chaincode may also derive private data keys/values based on the chaincode arguments. Chaincode should explicitly call stub.privateData.PutState(*collection, privateKey*, *privateValue*) to indicate storing the *privateValue* in ledger. During transaction simulation, instead of a single read write set, three read write sets are generated:

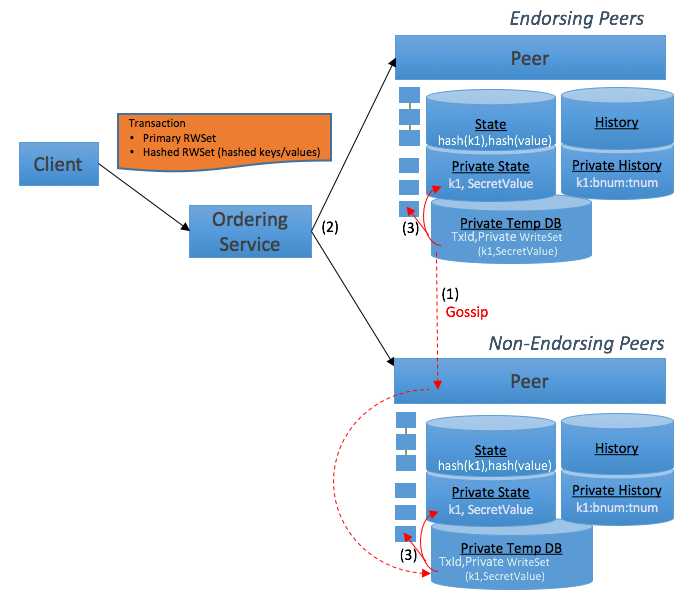
**(1) primary read write set** for public data, will be returned in endorsed proposal response and stored on-chain as normal. **(2) hashed read write set** for private data includes a hash of private keys and a corresponding hash of private values, will be returned in endorsed proposal response and stored on-chain as evidence. Note that there is no need to explicitly pass a hash to chaincode, the private data key and value will automatically be hashed by the fabric. Chaincode authors (or Fabric) need to ensure private keys are long enough (with salt) to ensure hashes of high entropy. The hashed ‘key’ (and ‘version’) will ultimately get persisted into the normal state db to assist state validation check. Note that a separate namespace will be needed in normal state db, so that the hashed keys don’t impact normal key range queries (similar to simple vs composite key segregation). No need to persist hashed ‘value’ in state db, storage in block is sufficient.

**(3) private read write set** that holds the actual private data, will be stored in private temp db until transaction commit, keyed by txid:endorsed\_at\_block:endorser\_id. This set is not returned in endorsed proposal response, is not sent to ordering service, is not stored in block on-chain. Read set is not technically required for simple state validations (since that is done against normal state DB), but read set will still be included because it contains the query-info section that is used for range query phantom validation check, which must be performed against real keys in private state DB (read set query-info holds range query start key, end key, and hash of results so that queries can be re-executed at validation time to check for phantoms, e.g. added keys since simulation time). The private keys/values from the private write set will ultimately get persisted into the private state db upon transaction commit, to enable chaincode private data APIs.

At endorsement time, in addition to getting saved in the local temp DB, the private read write set will also be gossipped to at least K other peers for redundancy (I internal to org and E external), to ensure that private data is available at commit phase. ESCC will ensure that this is complete before endorsement occurs. The K other peers will also store the private read write set in their temp DB and continue gossipping to make the data available across the peers in the network.

Note the tempdb key **txid:endorsed\_at\_block:endorser\_id**:

* **txid** portion is used as lookup key during subsequent validation/commit phase.
* **endorsed\_at\_block** (or saved\_at\_block) will be used for subsequent block-based cleanup (block-to-live BTL policy).
* **endorser\_id** is required in case different endorsers pass different private read write sets (the subsequent validation step will iterate through these to determine the correct one based on match against hashed write set).



***Figure 3. Validation/Commit data flow***

1. *Gossip private data from Endorsers to non-Endorsers (during endorsement).*
2. *Tran ordered into block and distributed to all peers.*
3. *Private data committed to private write set log and private state DB.*

### Validation phase

* Endorsement policy is checked as normal.
* Validate primary ReadSet and hashed ReadSet versions against normal State DB for state validation check, aka MVCC check.
* Validate range queries by doing phantom checks as normal (re-query to ensure query results are the same at simulation time and validation time). Use primary ReadSet against normal state DB and use private ReadSet against private state DB.
* If there are hashed read write sets in transaction, validate that hash of private write set (keys and values) in temp db matches hashed write set keys and values (evidence) from the transaction.
  + There may be multiple entries of the txid in the temp db with different endorser\_ids. Look at the entries matching endorser ids from the endorsed transaction. This validation step will iterate through these to determine the correct one based on match against hashed write set. In most cases only the first one will need to be checked.
  + If the peer does not find any private read write set in temp DB based on txid, the peer can request it from other peers via Gossip, likely trying endorsing peers first. In the very worst case, this may require gossiping with different peers for each transaction in the block.
  + If the private read write set still cannot be found, have a policy option specifying whether to continue with missing private data or halt block processing on this peer.
    - If configured to continue with missing private data: Assuming the tran passed MVCC check (using primary and hashed read set versions), the tran is marked as valid and we can continue on by leaving a missing (or outdated) entry in private state db. This does not break the integrity of the blockchain, but it does mean that this peer cannot endorse future transactions using this private key. Endorsements that do a key-based get from private state will need to hash the key and also get from normal state db. The version in normal state db and private state db must match in order to get a signed endorsement. If not, return an error to endorsement indicating the private state db mismatch.
      * Note that transactions that have non-key queries (range queries or rich queries) and then subsequent updates will not be possible, and support will have to be disabled in this mode. First of all, range query phantom checks against private data will not be possible since not all peers will have the private read write set that is required to perform phantom check validation (we must maintain determinism across all peers in validation phase). Secondly, a query against private state may return fewer results than it should. The missing keys would not be known and therefore it is not possible to validate against the normal State DB key versions. If all the endorsers were missing the data the problem would not be identified. This scenario would put the integrity of the blockchain at risk. Queries without subsequent updates could still be possible, although applications would have to understand the risk of missing items in the result set.
    - If configured to halt: The peer(s) without the private read write set cannot continue and must halt processing until the private read write set arrives. To try to resolve the missing private read write set, the client could resubmit the proposal to another endorser to trigger re-propagation. Alternatively an admin utility could copy the private read write set to the peer that needs it. Note that in this mode range query phantom checks against private data can continue to be enabled (assuming no partitioning of private data). In worst case scenario private data may be lost from client and all peers - this is the fundamental risk with side db design.

### On commit

* The block is committed to chain block storage as normal, and any private read write sets are committed to private write-set log. These are the primary data stores for committed data that can be used for recovery scenarios and state transfer.
* Assuming the transaction is validated:
  + The **primary write set keys/values** (and versions) are stored in normal state db as always (as well as history db).
  + The **hashed write set** **keys** (and versions) are stored in normal state db to support future state validation checks. Note that a separate namespace will be needed in normal state db, so that the hashed keys don’t impact normal key range queries (similar to simple vs composite key segregation). No need to save hashed ‘values’ since they will not be used in chaincode operations or validation, and are available in block. Potentially save hashed write set in normal history db as a future option.
  + The **private write set keys/values** are stored in the private state DB (and private history DB in future).

Irrespective of the transaction state (valid/invalid), we preserve the private write-set in the private temp DB in case other committers still need it. The cleanup will be based a time-to-live policy (implemented as block-to-live policy).

## 2.2. Querying private data

To query on privateValue stored in private state DB, chaincode should call stub.privateData.GetXXX(collection,...) where XXX can be key query, range query, rich query, or history query (in future). To query on normal data, chaincode should call stub.GetXXX(...).

Upon query on private data, <hashedKey, version> will be added to hashed read-set, to enable state version check at validation time.

## 2.3. Deleting private data

To delete a privateValue stored in private state DB, chaincode should call stub.privateData.DeleteState(collection, key).

Deletes are indicated in write sets with null value and IsDelete marker as true. Private data deletes therefore would result in a hashed write set entry and private write set entry like any other private data update. Upon committing the write set, the respective item will be deleted from state database. The original write set and the delete write set will both be maintained in private write set log. See the later purge topics to understand how these can be purged as well.

## 2.4. Savepoints and Recovering private data

### Recovery of private state and history DBs (in future)

Like state db and history db, the private state db and private history db will maintain a savepoint to enable recovery logic. If block height is above any of the database heights upon peer start, the database(s) will be recovered from private write-set log to catch up to block height, before normal processing resumes. Private write-set log is organized by blockNum:tranNum, therefore it is trivial to iterate through the blocks and transactions, applying the private write-sets to the private state and private history DBs. Since private write set log is not signed, also verify each hash with the corresponding hashed write sets as the recovery proceeds.

### Recovery of private write-set log

If private write-set log is itself not up to date (based on savepoint relative to chain block storage), it will need to be recovered first via state transfer from another peer. Retrieve blockNum:TranNum records by gossiping with a peer that has a more up to date private write-set log. New ledger APIs will be made available to gossip for this purpose. Gossip will send the normal block along with the block transaction’s private write-sets in a single structure (assuming the peer has access to all the collections).

## 2.5. Purging private data

### Purge of private temp db records

Recall that private temp db holds uncommitted data that peers can use to obtain the private write-sets during validation/commit phase. This temp data eventually needs to be cleaned up after all peers have committed the data. Additionally, temp data may have been orphaned if endorsements are not submitted to the network as transactions to commit.

Each txid-based key in private temp db will also have a block number concatenated in the key (endorsed\_at\_block from endorser or saved\_at\_block from receiver?). A bulk purge utility will run within peer to purge the temp data. Assuming goleveldb is used for tempdb, the purge utility must run within peer. It could run every X blocks and also be triggered externally. The purge utility will perform a full range scan of all temp data and look at block number embedded in key. Delete temp db record if block number is older than Y blocks. E.g. Every 1000 blocks wake up and purge temp db entries with block numbers less than (current block number - 5000). The block numbers will be specified in a policy at chaincode instantiation time.

### Purge of private read write sets and private state db

A similar purge job can be run within peer to delete old private read write sets and associated private state db entries. Recall that private read write set log is keyed on blocknum:trannum.

* Scan for private read write sets in blocknum purge range
* For each private read write set in the purge range
  + For each key in the private read write set, retrieve it from private state db
  + If the version (block height) is newer than the purge range, leave it in private state db
  + If the version (block height) is in the purge range (or older), purge it from private state db

Likewise the private read write set log is also purged based on policy.

Need to investigate how purge (which is deleting the data) may impact state transfer (which assumes the data is there).

Need to investigate how purge impacts simulations. Likely need to run the purge job in small increments within existing block processing locks, so that transaction simulations do not see inconsistent data. Actually, purging by definition introduces potential inconsistencies, but this would limit the inconsistencies.

## 2.6 Additional Considerations (WIP)

### Salt Considerations

When we protect data confidentiality using hash algorithms, it is important to ensure the input to the hash has sufficient entropy (to prevent dictionary attacks). A mechanism to increase the entropy of a hash is to include a salt / secret (in additional to the original value). When a salt is included into the hash, treating the salt as private data additionally increases the resilience against attackers (as it becomes highly difficult to guess the hash inputs).

### Properties

* Each key has a different salt.
* Salts are kept private (only peers that should have the state can observe the salts).
* Salt must have sufficient entropy (derived from a random source and of sufficient length).

### Handling Salts within Fabric

To assist CC developers, Fabric can (and should) manage the salt included in the hashes of the keys and values.

Confidentiality of Salts (and limitations)

As private state holes can occur (so that the peer doesn’t halt when it’s missing private state, based on policy) and we want to ensure that all channel endorsers know about the existence and validity of a key, the salt for the key needs to be distributed with normal ledger state updates (but needs to be combined with a secret kept private to the appropriate peers). However, the salt for the private values must be better protected by distributing with the private read/write sets (as the value hashes are not involved in determining key existence).

Illustrative Design

* Chain Code developer uses the private data shim API to query or write values. e.g.,
  + <cc-key>=<cc-value>.
* The Fabric ledger only stores hashes of <cc-key> and <cc-value> (as described elsewhere in the document).
* Fabric uses salts to increase confidentiality of the hashes stored in the ledger. Each <cc-key> and <cc-value> hash is computed with different salt values.
  + Salt is randomly generated. To maintain determinism, the random seed comes from the client (as a transient). This seed is then used as the input to a PRNG/KDF on the endorsing peers to produce a random secret from the seed.
  + To further enhance confidentiality (particularly for <cc-keys> as we will see next), these “transactional” salts are also combined with a pre-distributed secret salt for the CC/partition.
* Fabric persists the “transactional” portions of salt.
  + To overcome the hole limitation described above, Fabric stores the salt for <cc-key> in the ledger. As mentioned above (but important to re-emphasis here), this salt will be combined with a pre-distributed secret salt during usage.
  + Fabric stores the salt for <cc-value> in the private state to maximize confidentiality. The salt for <cc-value> must be different than the salt for <cc-key> (and not guessable based on knowing the salt for <cc-key>).
* To summarize, when a private data shim API is used to query or write key/values:
  + The <key-salt> for the <cc-key> is determined based on:
    - Pre-distributed salt (for the CC / partition pair).
    - Salt for <cc-key> stored in the ledger. If this is a key creation then this salt is created based on the random seed from the client.
    - <key-salt>=Pre-distributed salt + salt for <cc-key>.
  + The <key-salt> is combined with <cc-key> prior to hashing.
  + The <value-salt> for the <cc-key> is determined based on:
    - Pre-distributed salt (for the CC / partition pair).
    - Salt for <cc-value> stored in the private state. If this is a write then this salt is created based on the random seed from the client.
    - <value-salt>=Pre-distributed salt + salt for <cc-value>.
  + The <value-salt> is combined with <cc-value> prior to hashing.
  + The hash of the private value and <value-salt> is validated against the hash stored in the ledger.
* The pre-distributed salt must be rotatable, where a new pre-distributed salt applies to new keys and values after a certain block height.

### Transient Data Considerations

When the client sends parameters to an endorsing peer that will be used by the peer as private data, it is treated as transient - meaning that the data is not included into data structures that will become persisted into the ledger.

### Properties

* Transient data is not distributed (directly sent from client to endorsing peer).
* Data automatically persisted by Fabric cannot be reversed into the transient data. By automatic, this means functionality outside of chain code control.
* Persisted data structure signatures (e.g., in the ledgers) remain verifiable without access to the transient data. This property allows signatures within blocks to remain verifiable even when the transient data isn’t available to the examiner.
* Peer is able to validate the transient data was sent by the appropriate client (and not manipulated).

### Illustrative Designs

* The client could place a hash of the transient data under the transaction proposal signature (not the transient data itself). The peer can then validate the transient data against the hash. As usual, the transient data should be augmented with a transient salt to increase entropy; or
* A separate signature for the transient data could be created.

# 3. Private State Collections and Partitioned Storage across Peers

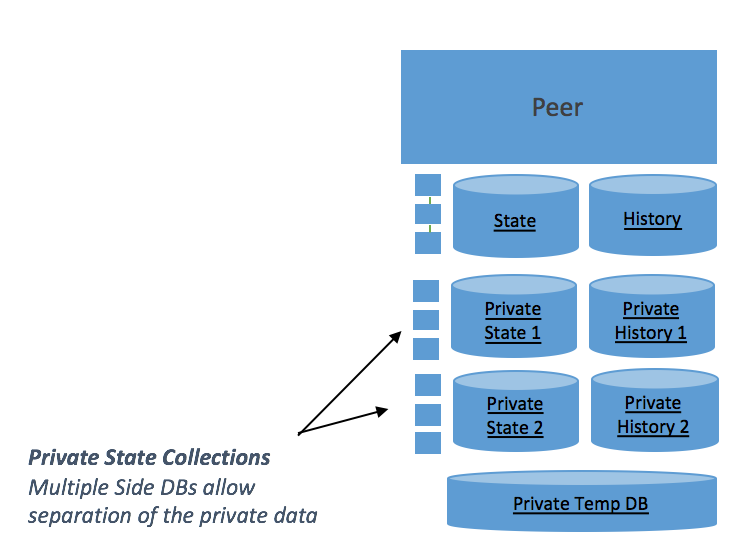
In a Fabric channel, individual participating peers may have differing responsibilities for persisting private state. Rationale for separating responsibilities can result from confidentiality requirements or storage constraints, for example. In some of these situations, the peers still need to transact on the same channel due to requirements for relative ordering, consistency of their transactions across partitions, or performance. Therefore, we will describe a mechanism for achieving these goals on a single channel.

To enable private data storage segregation, we add the concepts of collections and partitioning of storage to the private state mechanism described previously in this document.

3.1. Collection and Storage Partition Structure

Collections: A collection is a named Private/Side DB - each collection has its own state and history as shown in Figure 4. Otherwise, a collection has the same properties as Private/Side DBs that we discussed earlier.

Storage Partitions: Collections can be partitioned such that each peer has different responsibilities for persisting that collection as shown in Figure 5.

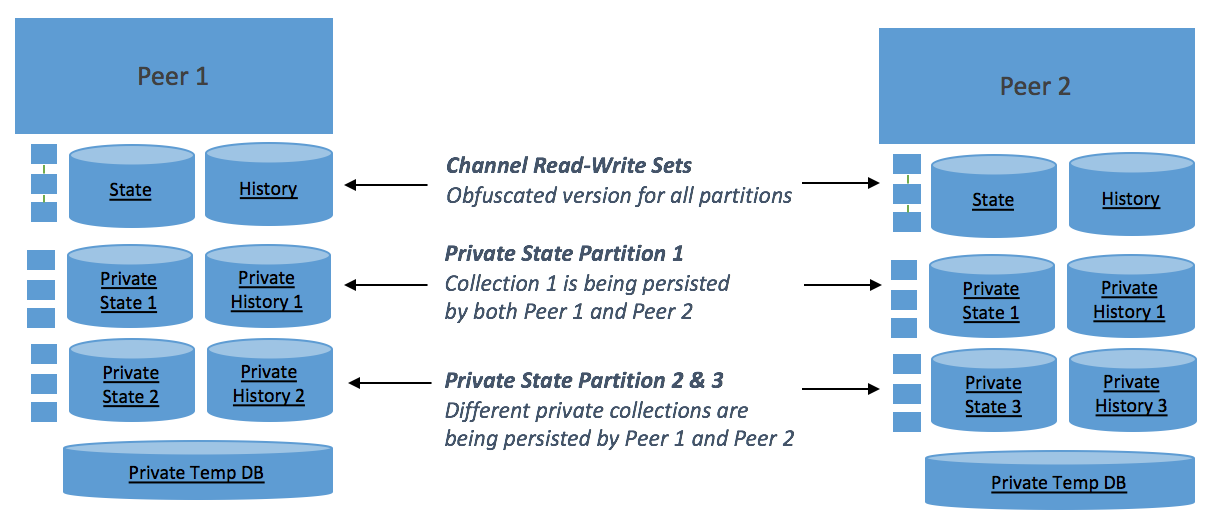


*Figure 4: Private State Collections - Multiple Private DBs exist on a single channel.*

In Figure 4, we additionally show that each collection has its own:

* Private State
* Private History
* Private Write-set log

This structure is analogous to the description provided earlier in the document, but with a set of these components for each collection. *The private data in the collection is therefore segregated from other collections’ data.*

**

*Figure 5: Private State Partitions - Different storage partitions of the private state collections exist between Peer 1 and Peer 2.*

In Figure 5, we show that the storage of each collection can be partitioned to different Peers. This storage partitioning is similar to the channel permission structure - e.g., determined by organization. As the channel's data structures retain knowledge of the read-write set for private data (but not the value), therefore all peers can still perform validation regardless of whether or not they have the storage partition. The private data in collections is therefore “physically” segregated between peers but consistency and ordering is still maintained across the channel.

3.2. Assumptions

This subsection introduces or revises assumptions listed in section 1, as follows:

1. When storage partitions are defined, private data is not *persistently* known to all peers on the channel, but rather a subset of peers. The private data is, however, *temporarily* known across the channel (revised from Section 1, assumption 2).
2. Primary and all private read write sets are committed as an atomic transaction (note: should also be an earlier assumption with Primary and Private Read Write Set).
3. ~~Peers can have restrictions on the operations they perform.~~
   * ~~Peers that are not part of a storage partition are unable to read values associated with keys on that storage partition.~~
   * ~~Peers that are not part of a storage partition can be restricted from performing Update, Delete.~~
4. ~~Peers can create keys (and associated) values even if they are not on a storage partition.~~ Peers can *blindly* create keys (and associated values) even if they are not on a storage partition. Blind create leverages blind writes being possible in this situation. See additional considerations below.
5. Peers are able to globally accept or reject transactions even when there are some private read write sets within a transaction that belong to storage partitions they do not have. I.e., there are no inconsistencies in the channel chain.
   * (The channel's data structures retain knowledge of the read-write set for private data, but not the values).
6. Values can be temporarily held by peers participating in the gossip network transportation for a storage partition.
7. Values held in the temporary DB will be cleared according to their TTL (Note: should be a previous assumption but is even more important with storage partitions).

3.3. Stub API

To support collections, the private data APIs need to support an additional parameter. e.g. stub.PutPrivateData(collection,key,value). The collection parameter indicates which named Private DB should be accessed (or modified).

To support storage partitions, the private data APIs should return an appropriate error to Chain Code when the Chain Code requests data that is on a storage partition the peer is not participating in. It is the client’s responsibility to determine which peers to invoke, if the invocation requires access to a particular storage partition.

3.4. Endorsement - Additional Considerations (WiP)

* *R*ather than simply two write sets being generated, a primary write set is generated along with X private read write sets - one for each affected Collection. i.e., X + 1 write sets are generated. As with the original model, the private read write sets are transmitted via gossip while the primary write set is transmitted via the Ordering Service.
* Peers that are not part of a storage partition cannot read the private values and should return an error to the User Chain Code.
* Transactions involving multiple storage partitions need consistency across the channel. E.g., Read Sets (along with portion of Write Set) may need to disseminated to entire channel to accomplish (as would be done for private data without storage partitions).
* Blind creation is achieved by performing a Get and, if the key doesn’t exist, then performing a blind write (note: if the key does exist, an invalid value is returned from Get due to not being included in the storage partition). During commit, the associated VSCC can then additionally enforce policies where updates are not allowed in these situations. This mechanism leverages the normal state DB containing all hashed keys and their versions. I.e., Peers can call Put and Delete (even when not on a partition) and can call Get to enable version checks (but not to get an actual value).

3.5. Validation - Additional Considerations (WiP)

* The primary and all private read write sets are validated at the same time to allow transactional consistency (all pass or all fail).
* Peers that are not part of a storage partition do not validate the private values against the Write Set (but validate the rest of the Read-Write set).
* ~~Peers are able to distinguish Create from Update or Delete in Write Sets, and are able to reject Updates or Deletes from peers not on the storage partition.~~
* Peers are able to globally accept or reject transactions even when there some private read write sets within a transaction that belong to storage partitions they do not have.
* For normal State DB range queries, a re-query is performed to validate no phantoms exist between simulation time and commit time (e.g., validate that no new keys have been added by other transactions that would influence the range query result set). This phantom check would not be possible with a partitioned private state DB, since not all peers have the private state DB (and since keys are hashed into the normal state DB). All peers must have the same validation logic to ensure determinism across peers. Therefore, the range query phantom check on private state DB queries will result in a transaction invalidation, and perhaps have an option for a less stringent mode where the phantom check is simply disabled. Note that it may be possible to re-enable in the future, if gossip is added at validation time such that peers that do have the data could reach phantom consensus during validation step.

3.6. Commit - Additional Considerations (WiP)

* The primary and all private read write sets are committed at once to allow transactional consistency (all pass or all fail). That is, from a client perspective they appear to be atomic commits, however from an implementation perspective it is handled as separate updates with locking behavior and recovery logic, same approach as is used for keeping block updates and public state db updates in sync.
* Peers that are not part of a storage partition never know the private values, and do not persist the private values to their state databases.

3.7. State Transfer and Gossip - Additional Considerations (WiP)

* *B*ulk state transfer of the private DB will be needed if an organization is joined to a storage partition.

3.8. Configuration Considerations (WiP)

* A configuration transaction should be used to create, re-configure, and drop collections in the channel.
* A configuration transaction should be used to create, re-configure, and drop storage partitions in the channel.
* A configuration transaction should be used to assign collections to a storage partition.
* Note: Potentially combine collection/partition concepts to simplify design.

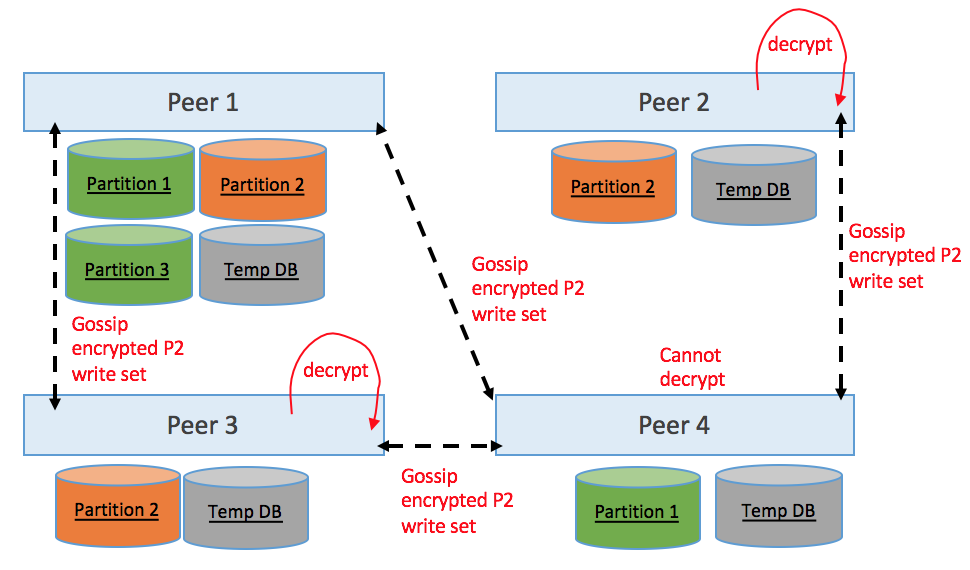
3.9. Archive, Prune, and Purge Considerations (WiP)

* Each value within Temp DB should be associated with a BTL. Within a reasonable delay to the BTL, the value should be purged automatically by the peer.
* It should be possible to archive and/or prune a Private DB by block range.
* It should be possible to archive and/or prune a Private DB logs and history by block range (similar to the goal for the Normal DB’s logs and history).
* It should be possible to drop a Private DB.
* It should be possible to assign a BTL to values within a Private DB. Within a reasonable delay to the BTL, the value should be purged automatically by the peer.
* It should be possible to mark a value as purgeable using an endorsed transaction, such that it becomes purged by peers in the next cycle. It should be possible that this remove / purge value transaction allows pruning from private state, logs, and history.
* BTL can be set to infinite for an item.
* It should be possible for a peer (or organization) to be removed from a storage partition.
  + Note: early cleanup at the peer may be manual in this case (when purge is needed prior to the BTL)?

# 4. Data Privacy - Option 1: Encryption for Private State

Certain parts of sensitive data might belong to the different parties, where none of these parties are willing to disclose the information to others. Therefore this information should be hidden or not viewable by peers which do not belong to organizations allowed to read it. However there is still requirement to provide reliability and resilience of the secret data, hence we would like to be able to leverage encryption in order to prevent peers from different organizations to read the pieces of private data they not allowed to.

4.1. Encryption during Transit

*Figure 6: Private State Transit Encryption - In this illustrative example a write set for Partition 2 is transmitted over Gossip started by Peer 1, and only Peers 2 and 3 can decrypt.*

4.2. Assumptions

1. Encryption public keys can be deterministically determined for each peer participant in a storage partition (or can be determined for an associated anchor peer, in the same organization, that can assist with decryption).
2. Peers are able to create an appropriate message encryption key (MEK) to protect values being sent over gossip (and encrypt it to the appropriate participants).

4.3. Endorsement - Additional Considerations (WiP)

* The private read write sets are encrypted using a message encryption key (MEK), and the MEK will be encrypted using public keys of anchor peers for each organization. See Figure 6 for an illustration.
* The structure should look like: <MEK>pub\_key1,<MEK>pub\_key3 || <data>MEK, where pub\_key1 is the public key of organization 1’s anchor peer and pub\_key3 is the public key of organization 3’s anchor peer. For illustration<MEK>pub\_key2 is not included in the structure as pub\_key2 belongs organization 2’s anchor peer and has been excluded from the partition.

4.4. Validation - Additional Considerations (WiP)

* Peers that are not part of a storage partition do not (and cannot) decrypt messages.

4.5. Commit - Additional Considerations (WiP)

* Encrypted data will be committed into in the plain (decrypted) form into Private State DB only if the peer allowed to read these data
* In order to provide better reliability and data resiliency peers which are not allowed to read the data will not commit it into Private State DB, rather continue to keep it in temp DB in encrypted form. Note: The encrypted data will be purged according to policy. For cases where the encrypted data shouldn’t even be distributed (at all) to some organizations, please see the Subchannels topic below.

4.6. State Transfer and Gossip - Additional Considerations (WiP)

* While gossiping encrypted data inside organization allowed to see it, we should gossip plain text. Once data to pass boundaries of two organization it should be encrypted again.
* Peers decrypt messages prior to validation or persistence, when they are part of the storage partition.

# 5. Data Privacy - Option 2: Subchannels for Private Data *(current choice, although will likely not use term ‘Subchannels’)*

Message routing policies could be customized to ensure that private data only traverses organizations, as determined by the policy for the partition. The primary motivation for decoupling message routing policy is to provide an additional layer of confidentiality protection beyond encryption. Additionally there could be benefits for larger channels to reduce the communication traffic of private data to a subset of peers (assuming there are sufficient peers to maintain resiliency). Similar to above, there can be peers acting as relays (who cannot decrypt) and peers participating in the partition (who can decrypt) -- subchannels simply offers more flexibility on setting policy for message routing.

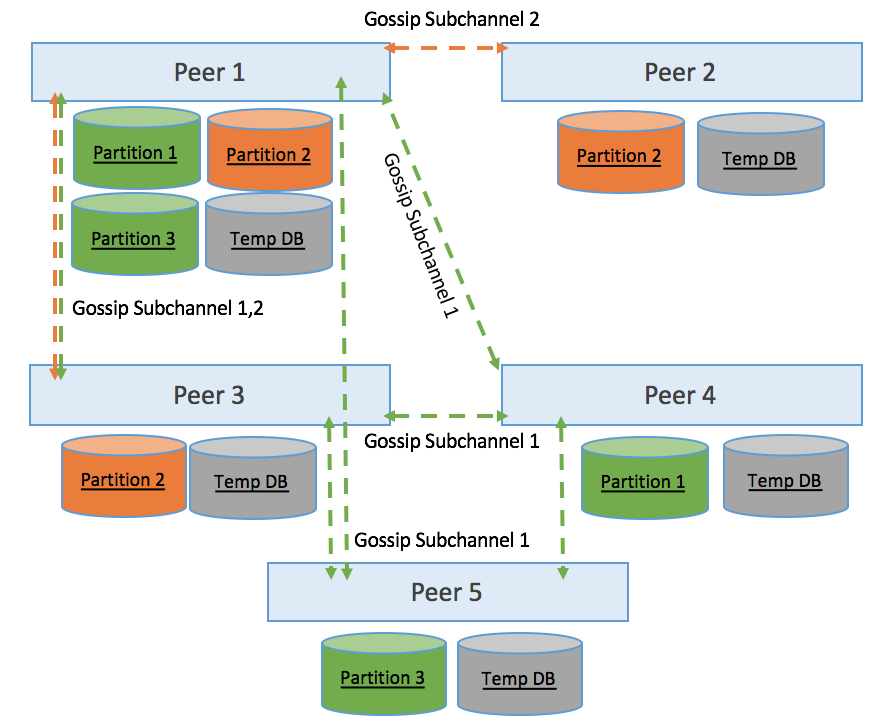
5.1. Partitioned Messaging at the Subchannel Level

P2P communication for managing collections can be limited to a subset of the channel’s gossip network - effectively creating partitioned messaging at the subchannel level.

In Figure 7, we show various configurations of messaging subchannels to illustrate the decoupling of a storage partitions and messaging subchannels.

In particular we illustrate:

* Partitioned messaging subchannels are joined by peers, as permitted by a previous configuration txn.
* Storage partitions are persisted by peers, as permitted by a previous configuration txn.
* Private data within a storage partition is transported by a messaging subchannel, as set by a previous configuration txn.
* The above three parameters are decoupled.



*Figure 7: Private State Messaging Subchannels (Illustrative Example) - A subset of the channel’s gossip network is used to disseminate information for a particular collection.*

From the above illustrative example configuration, you can observe the decoupling of storage partition and gossip configuration:

* Messaging Subchannel 1 is joined by Peers 1,3,4,5.
* Messaging Subchannel 2 is joined by Peers 1,2,3.
* Storage Partition 1 is configured to be persisted by Peers 1, 4 and its private data to be transported by Subchannel 1.
* Storage Partition 2 is configured to be persisted by Peers 1,2,3 and its private data to be transported by Subchannel 2.
* Storage Partition 3 is configured to be persisted by Peers 1,5 and its private data to be transported by Subchannel 1.
* Note that Peer 3 acts as a transport for Storage Partition 1 & 3, but doesn’t persist them.

Note: This subsection could use some additional gossip considerations.

5.2. Configuration Considerations (WiP)

* A configuration transaction should be used to create, re-configure, and drop messaging subchannels in the channel.
* A configuration transaction should be used to assign storage partitions to messaging subchannels.

5.3. Subchannel Gossip - Additional Considerations (WiP)

* Gossip layer need to be able to route messages within different sub channels based sub-channel participation, namely message should be routed only between peers of same organization
* Last configuration update block number should be published within block metadata to allow gossip layer to support scalable dissemination of blocks within sub-channel

# 

# 

# Appendix 1 - Summary of work items

**Ledger (Dave/Manish to lead)**

**User Stories and Tasks below will be encoded in Jira, pending review**

**Ledger Work items (current)**

**Story: Ledger simulation support for private data (using CouchDB state db)**

1. Define proto messages for private rwset and extend existing proto messages for containing hashed rwset (Senthil) - independent
2. Private data transient store, including access APIs (Senthil) - independent
3. Extend KVLedger statedb common structures for maintaining private data and hashed data (Manish) - Depends on (1)
4. Extend couchdb based state mgmt (Manish) - Depends on (3)
5. Extend Transaction simulation for handling private data (Manish)

* private data reads/writes
* populate hashed rwset portion of the TxRWSet
* prepare the private rwset
* add private RWSet to 'Private data transient store’

**Story: Ledger validation for private and hashed data (Manish)**

1. State validation of hashed keys
2. Validate private transient data matches hashed data
3. Support N transient db entries (in case different endorsers pass different transient data)

**Story: Ledger commit for private data**

1. Private data store (equivalent to block storage for regular data), including access APIs that can support state transfer (Senthil) - independent
2. Commit to private data store
3. Commit to private state DB (CouchDB)

**Story: Ledger recovery for private data**

**Story: Partition private data so that peers only have access to data that they are authorized to**

**Story: Handle missing private data at validation/commit time**

1. Request private data from other peers

**Story: Allow block processing to continue with missing private data**

1. Endorsement to check for gaps between hashed and private data, return endorsement failure
2. Disable non-key query/update support when operating under ‘continue with missing private data’ (or when there is data partitioning).

**Story: Salt private keys for additional security (pending)**

**Story: Purge from temp db based on Blocks to Live (BTL) policy**

**Story: Purge from private state db and private read write set storage based on BTL policy (by partition)**

**Peer/Chaincode (Murali to lead)**

* Chaincode APIs for private data (by partition)
* Endorsement - Return error for unsupported partition
* ESCC to ensure dissemination of private data prior to endorsement.
* Policy support at chaincode instantiation time, and associated access APIs
  + Endorsement dissemination of private data to K peers (I internal and E external)
  + Retention policy for temp data (BTL)
  + Retention policy for private state data (BTL) (by partition)
  + Org:Partition mapping
  + Continue with missing private data or halt processing

**Crypto (Elli to lead)**

1. (may already included in other people’s work-items) Define peer partitions using a configuration transaction via special type of “policies”. Peer partition creation should specify in terms of policies the set of peers/organisations that are included in the partition.
   1. would partitions be defined in an identity channel? It may be easier if partitions are to be used cross-channels, and their definition/update via configuration blocks/txs
   2. who would be authorised to update the partition
   3. is the partition a chaincode refers to anonymous? Should it not be anonymous?
   4. do we want to consider the option of having partitions be defined/configured via a system chaincode? May be easier to leverage sideDB features there...
2. enhance peer MSP with encryption key provision capability: peers need to be able using some that resides to the blockchain to generate key-material that would allow to decrypt information associated to the channel. This item refers to the design of the mechanism to allow for such encryption capabilities
3. Enhance the peer partition definition (e.g., via some transaction) to distribute proper key-material through the chain. Key-material could also be part of the genesis of the sideDB.
4. Peers use the key mentioned in step (3) to encrypt data for a peer they need to send updated parts of the state.
5. Peers use their key to decrypt information associated to partitions they are involved in.
6. Design and decide who should be allowed to update a certain partition’s state.
7. If the partition a certain transaction refers to is assumed to be anonymous, i.e., only only peers who are member of partitions know the rest of the members of a partition, then this item refers to the design of techniques that would allow for this. We may also need to explore the relationship between endorsers included in endorsement policies and partitions.

**Gossip (Artem to lead)**

1. Story: **Provide high level design and flow of handling side db**
   1. Endorsement time sequence diagram
   2. Validation/Commit time sequence diagram
   3. Define API’s between communication and endorsement layers
   4. Define API’s between communication and encryption layers
2. Story: **Private data dissemination (endorsement time)**
   1. Add message store to support distribution of private data
   2. Expose gossip broadcast API for endorser
   3. Provide an ability to verify replication guaranties
   4. Routing based private data encryption
3. Story: **Private data request (validation time).**
   1. Add an ability to request missing private data to complete transaction validation.
   2. Do we need private data for validation?
   3. Optimize to request private data from endorsers and only if missing broadcast request to complete missing holes.
4. Story: **State transfer - gather up private read write sets alongside blocks (by partition)**
   1. Change/Extend current state transfer to include transferring of private state
   2. Extend state transfer response to include private data
   3. State transfer of private data should be aligned with partitioning and encryption policies
5. Story: **Maintain organizations partition, keep track on updates of partitioning participants**

# Appendix 2 - Staging

## Features in progress

* Core Side-db components and lifecycle
* Gossip
* Policies
* Purge from temp DB
* Purge from private state DB
* Collections
* Subchannels
* StateDB - CouchDB support
* Continue with missing data = true (no query/update support)

*Note: Need to investigate further splitting Phase 1 into multiple phases to expedite an initial delivery.*

## Future Features

<https://jira.hyperledger.org/browse/FAB-5108>

* StateDB - LevelDB support
* HistoryDB
* Attempt to self-heal missing private state db entries post-commit
* Strong delete (purge a piece of private data without waiting for BTL purge job)
* Partition:org mapping changes
* Full subchannels support (e.g. gossip layer participation in partitioning)
* Continue with missing data = false (query/update support, but no phantom checks possible when there is partitioning).
* Salt handling within Fabric

## Jira breakdown for Side DB - 1151 (supersedes bullets above)

**Phase 1 (mid-August) - Restrict private data from transactions/orderer. Assume all peers have access to all private data. Application must get endorsement from all peers.**

* FAB-5079 P1 - Ledger support for private data collections/partitions (simulate, validate, commit), restrict private data from transaction/orderering
* FAB-5080 P1 - Chaincode API support for private data partitions
* FAB-5081 P1 - State transfer of blocks to include associated private data

**Phase 2 (mid-September) - Map private data partitions to organizations. Gossip dissemination to authorized organization's peers. Application gets endorsement from any subset of peers that have access to the private data.**

* FAB-5082 P2 - Collection ACL - Chaincode policy to map private data collections/partitions to authorized organizations
* FAB-5084 P2 - Gossip (push) dissemination of private data at endorsement time, to organizations/peers authorized to receive the private data
* FAB-5088 P3 - Gossip (pull) request for private data at validation time, to provide private data if it is not yet available on peer upon validation
* FAB-5085 P2 - Endorsement fulfillment of private data policies (return error if chaincode attempts to access private data that peer's organization is not authorized to
* FAB-5086 P2 - Ledger toleration for missing private data (e.g. check for missing data at endorsement time)

**Phase 3 (October) - Additional resiliency; purge of private data**

* FAB-5083 P3 - Chaincode policy to specify minimum number of (internal/external) peers private data must be disseminated to (including enforcement)
* FAB-5089 P3 - Chaincode retention policy for transient private data (block-to-live policy)
* FAB-5090 P3 - Chaincode retention policy for private state data (block-to-live policy)
* FAB-5091 P3 - Ledger purge from private transient db based on block-to-live (BTL) policy
* FAB-5092 P3 - Ledger purge from private state db and write set storage based on block-to-live (BTL) policy
* Coordinator support for missing private data (commit with missing private data after configurable amount of time if private data cannot be gossip pulled)
* Any additional discoveries that are must-do for a MVP delivery of private data.

**Future Items - FAB-5108 (not currently in plan)**

* [FAB-5096](https://jira.hyperledger.org/browse/FAB-5096) Support changes to private data collection:organization mapping (likely move P3)
* [FAB-5093](https://jira.hyperledger.org/browse/FAB-5093) Reconcile missing private data on peers that are authorized to have it
* [FAB-5094](https://jira.hyperledger.org/browse/FAB-5094) Ledger capture history of private data
* [FAB-5097](https://jira.hyperledger.org/browse/FAB-5097) Strong delete of private data prior to block-to-live policy
* [FAB-5098](https://jira.hyperledger.org/browse/FAB-5098) Add chaincode policy to halt rather than proceed with missing private data
* [FAB-5099](https://jira.hyperledger.org/browse/FAB-5099) Add chaincode policy to allow non-key queries in transactions, even if phantom validation cannot be performed due to private data not available (no access or missing)
* [FAB-5101](https://jira.hyperledger.org/browse/FAB-5101) Add salt to private data key/value hashes
* Ad hoc collection creation (e.g. collection per key use case)