Hyperledger Fabric Peer Design

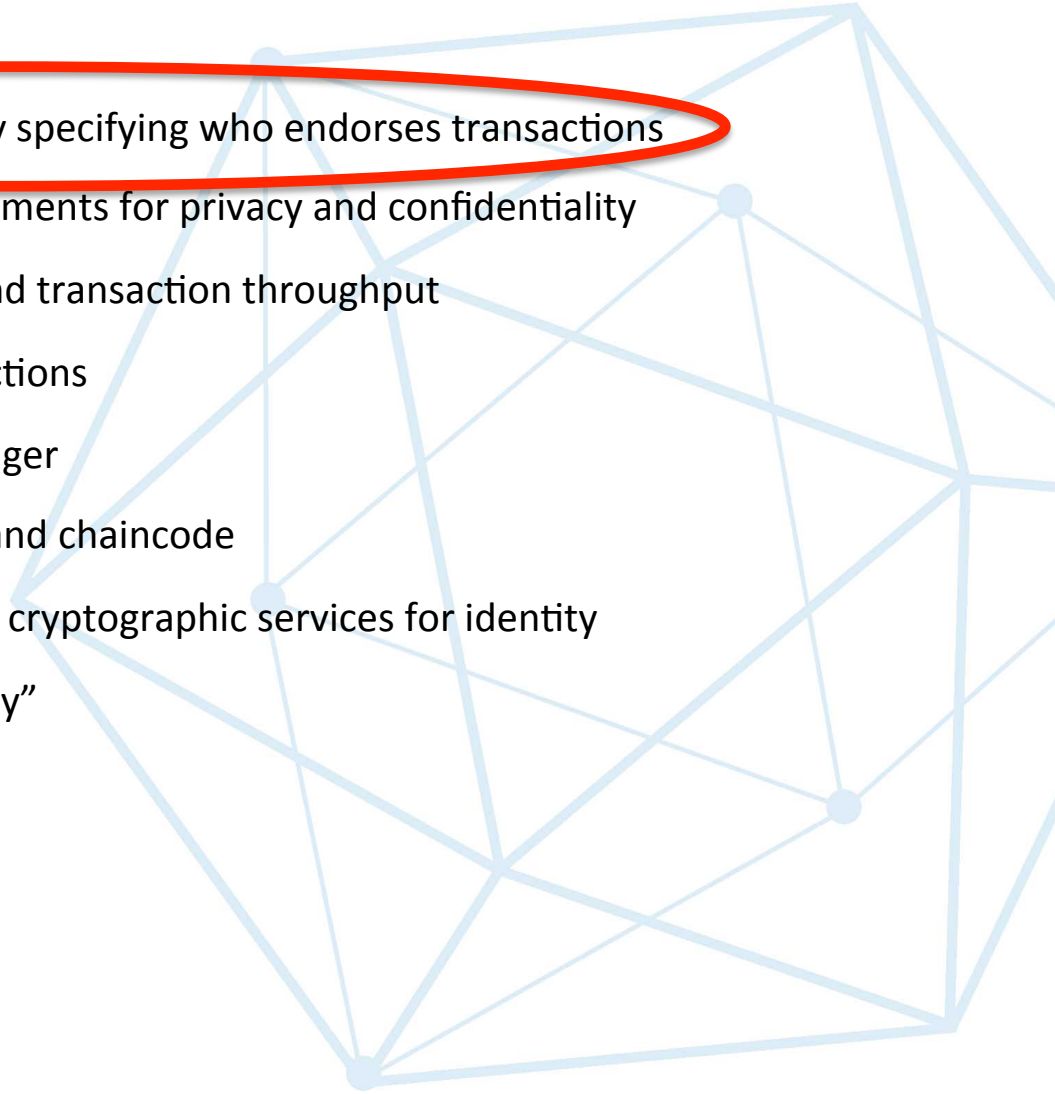
Lance Feagan
IBM China Research Lab

Topics

- Today
  - System Chain Code: ESCC, VSCC
  - Ledger: Design, Interfaces, Implementation
- Future
  - Gossip Protocol
  - Events

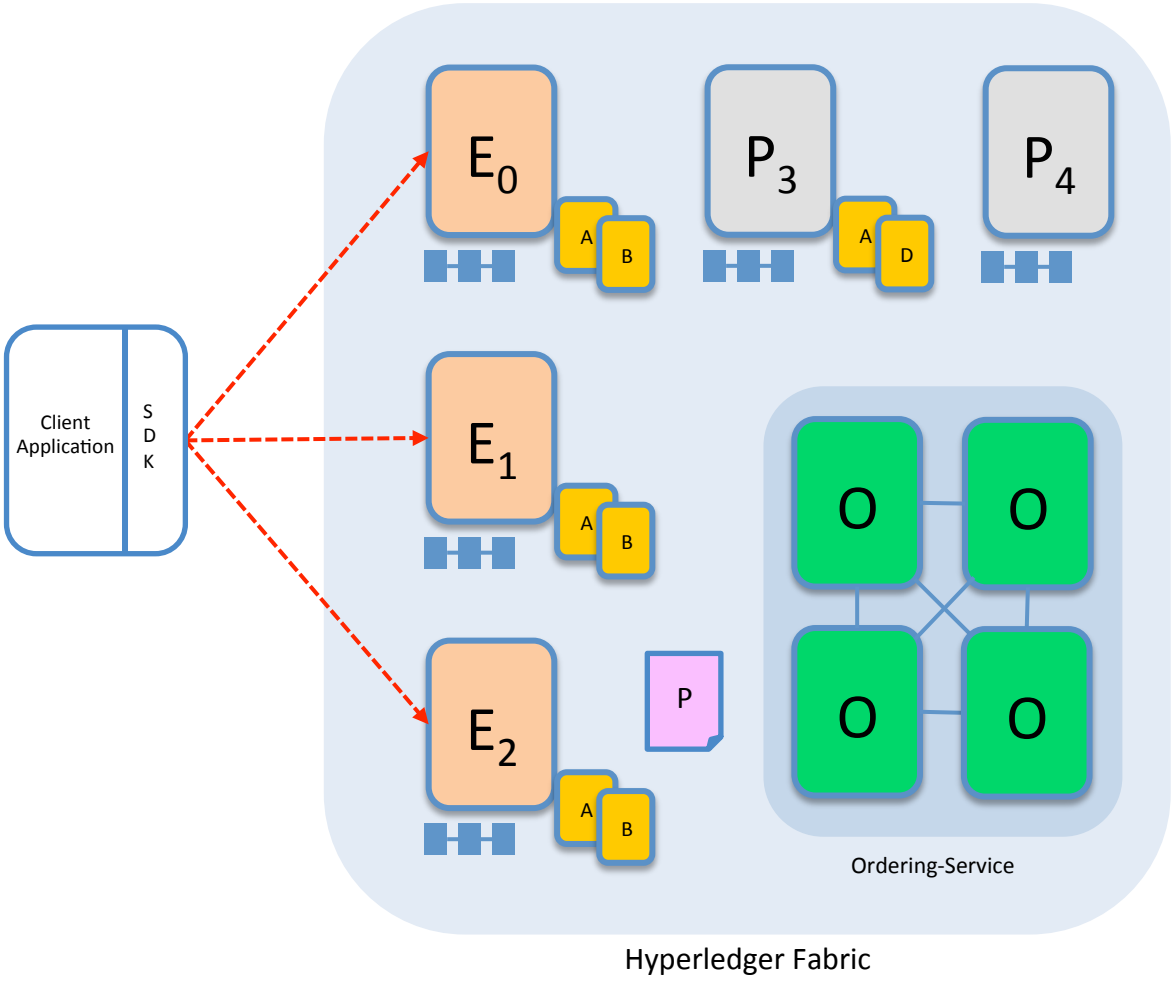
ENDORSEMENT POLICIES

Overview of Hyperledger Fabric v1 – Design Goals



- Better reflect business processes by specifying who endorses transactions
- Support broader regulatory requirements for privacy and confidentiality
- Scale the number of participants and transaction throughput
- Eliminate non deterministic transactions
- Support rich data queries of the ledger
- Dynamically upgrade the network and chaincode
- Support for multiple credential and cryptographic services for identity
- Support for ”bring your own identity”

Sample transaction: Step 1/7 – Propose transaction



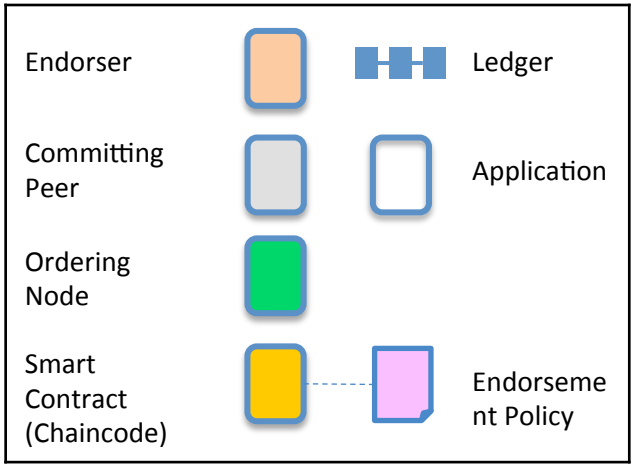
Hyperledger Fabric

Application proposes transaction

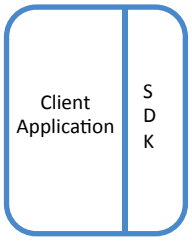
Endorsement policy:
- "E_0, E_1 and E_2 must sign"
- (P_3, P_4 are not part of the policy)

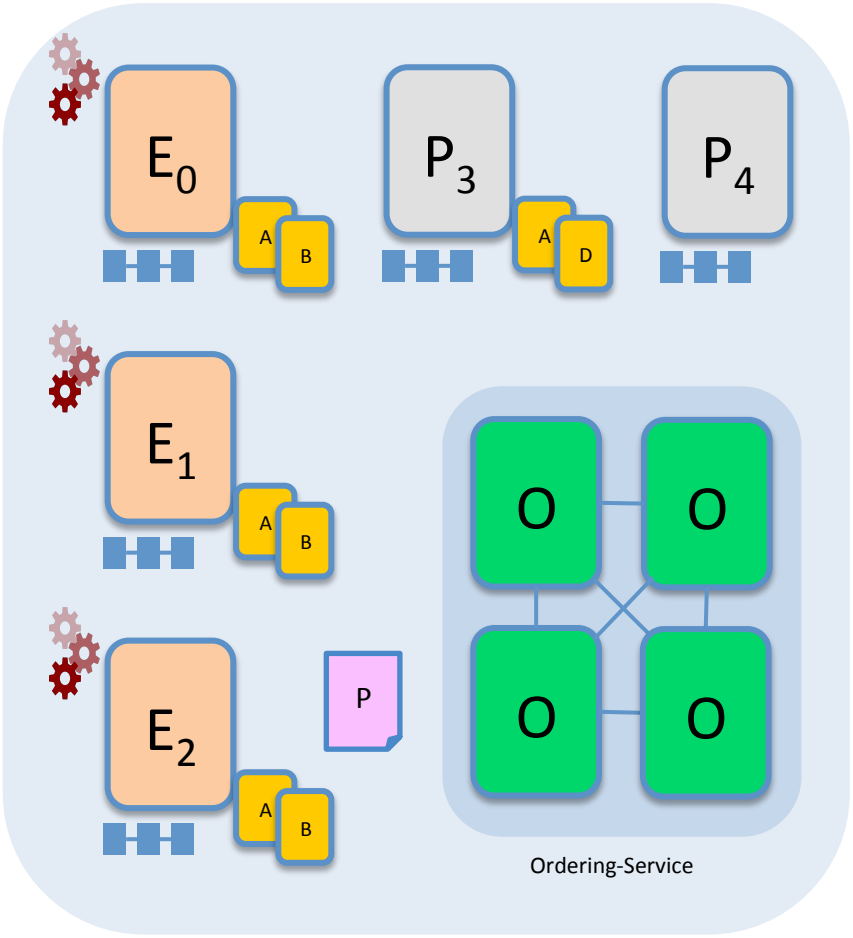
Client application submits a transaction proposal for **Smart Contract A.** It must target the required peers {E_0, E_1, E_2}

Key:



Sample transaction: Step 2/7 – Execute proposal





Hyperledger Fabric

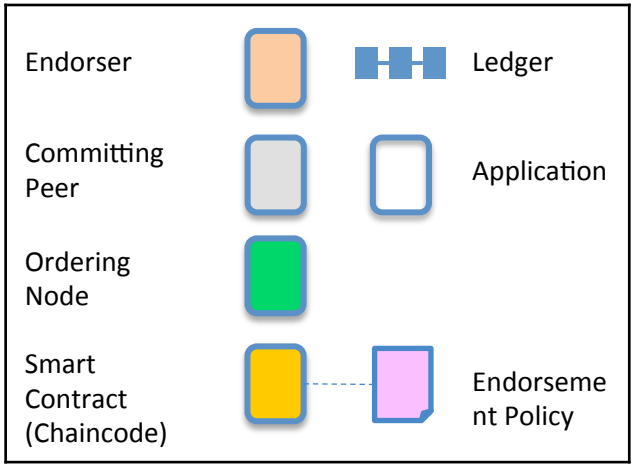
Endorsers Execute Proposals

E_0, E_1 & **E_2** will each execute the *proposed* transaction. None of these executions will update the ledger

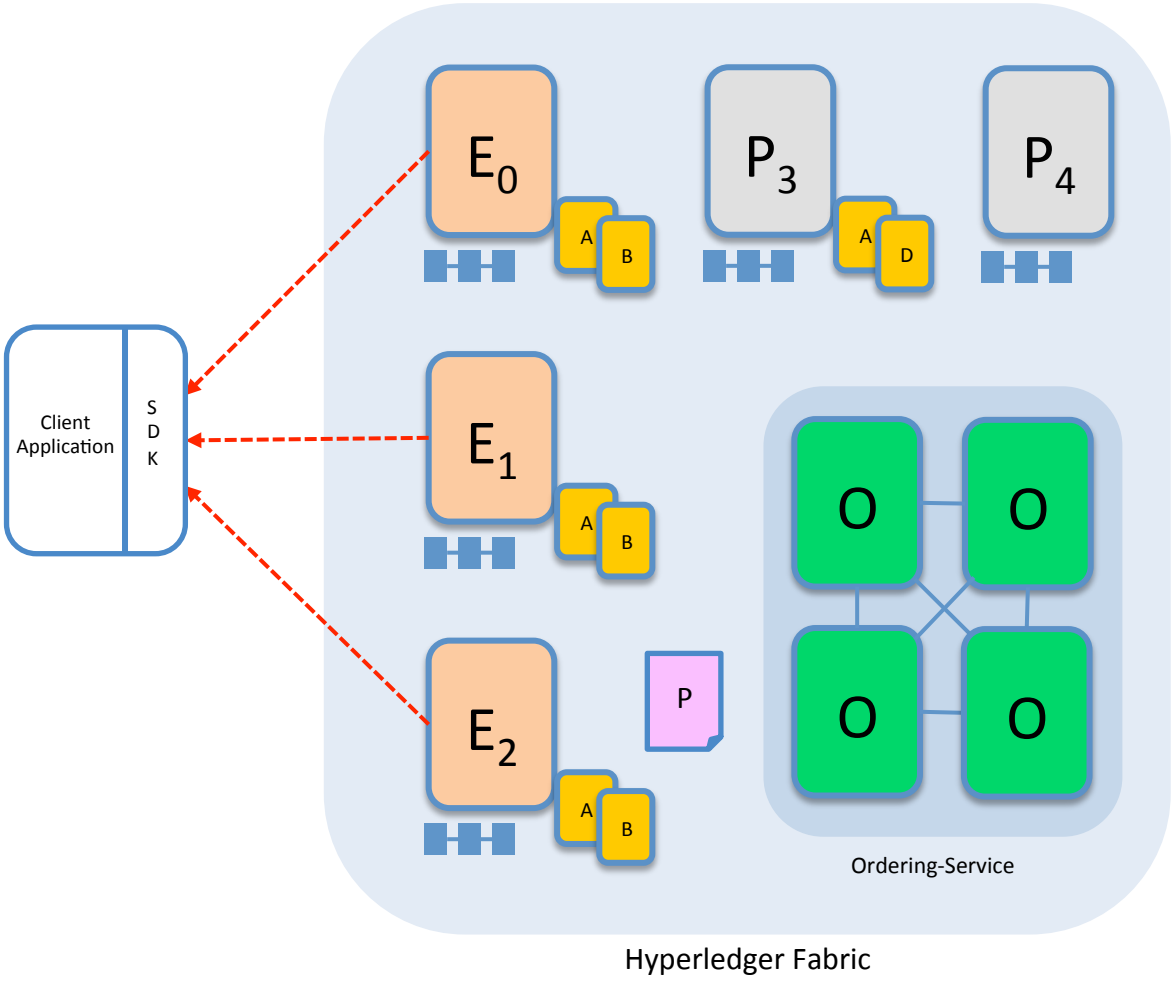
Each execution will capture the set of **R**ead and **W**ritten data, called **RW sets,** which will now flow in the fabric.

Transactions can be signed & encrypted

Key:



Sample transaction: Step 3/7 – Proposal Response



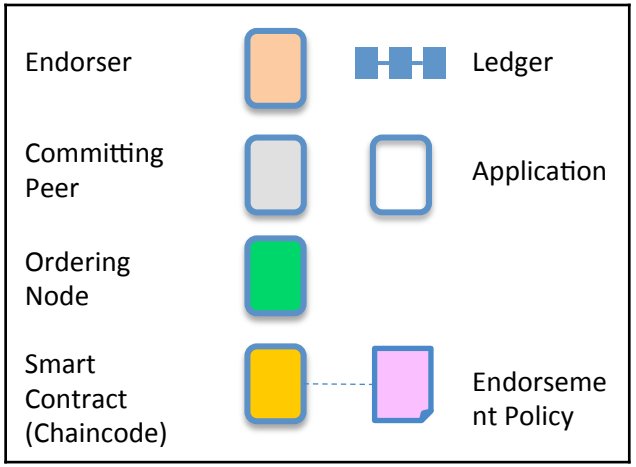
Application receives responses

RW sets are asynchronously returned to application

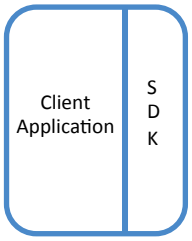
The RW sets are signed by each endorser, and also includes each record version number

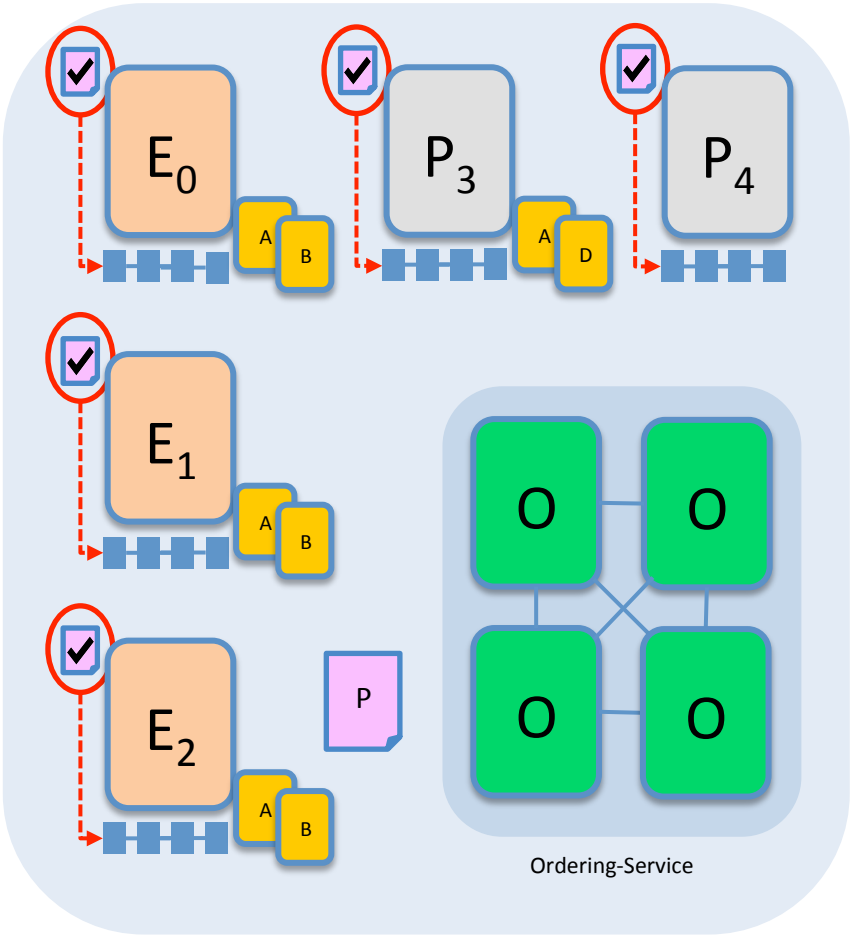
(This information will be checked much later in the consensus process)

Key:



Sample transaction: Step 6/7 – Validate Transaction





Hyperledger Fabric

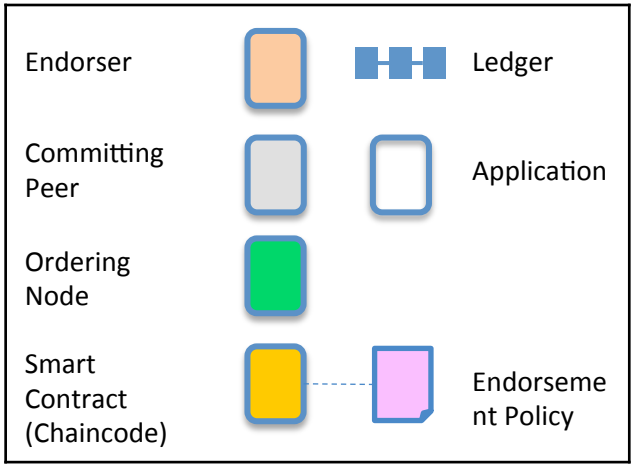
Committing peers validate transactions

Every committing peer validates against the endorsement policy. Also check RW sets are still valid for current world state

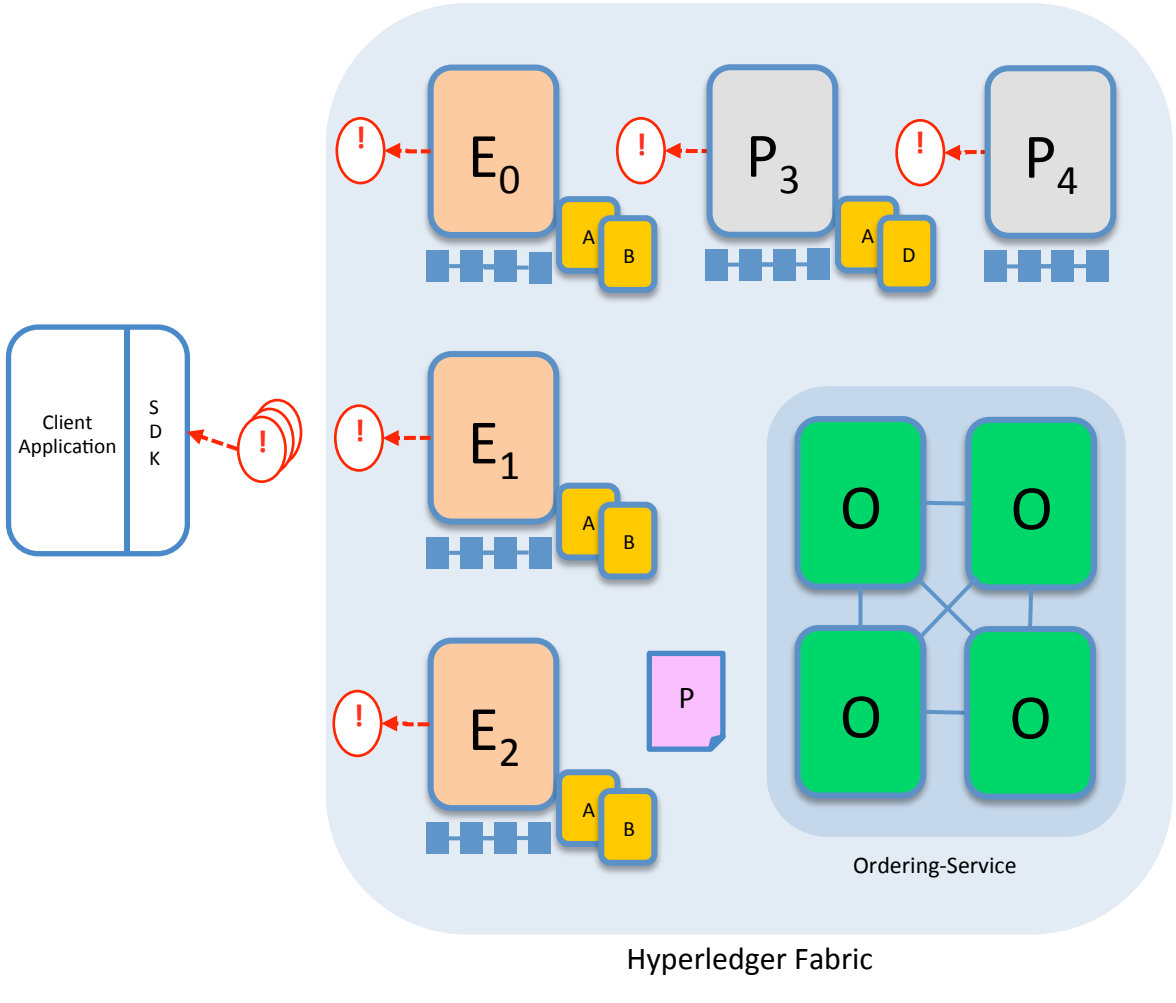
Validated transactions are applied to the world state and retained on the ledger

Invalid transactions are also retained on the ledger but do not update world state

Key:



Sample transaction: Step 7/7 – Notify Transaction



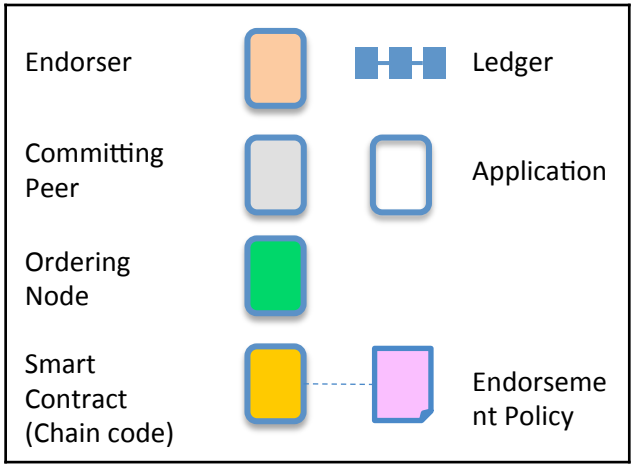
Hyperledger Fabric

Committing peers notify applications

Applications can register to be notified when transactions succeed or fail, and when blocks are added to the ledger

Applications will be notified by each peer to which they are connected

Key:



Transaction Defined

- Transaction is a chaincode function call
  - transaction : [ledger] <proposal> <endorsements>
  - proposal : [channel,] chaincode, <function-spec>
  - function-spec: function name, arguments
  - endorsements : proposal hash, simulation result, signature

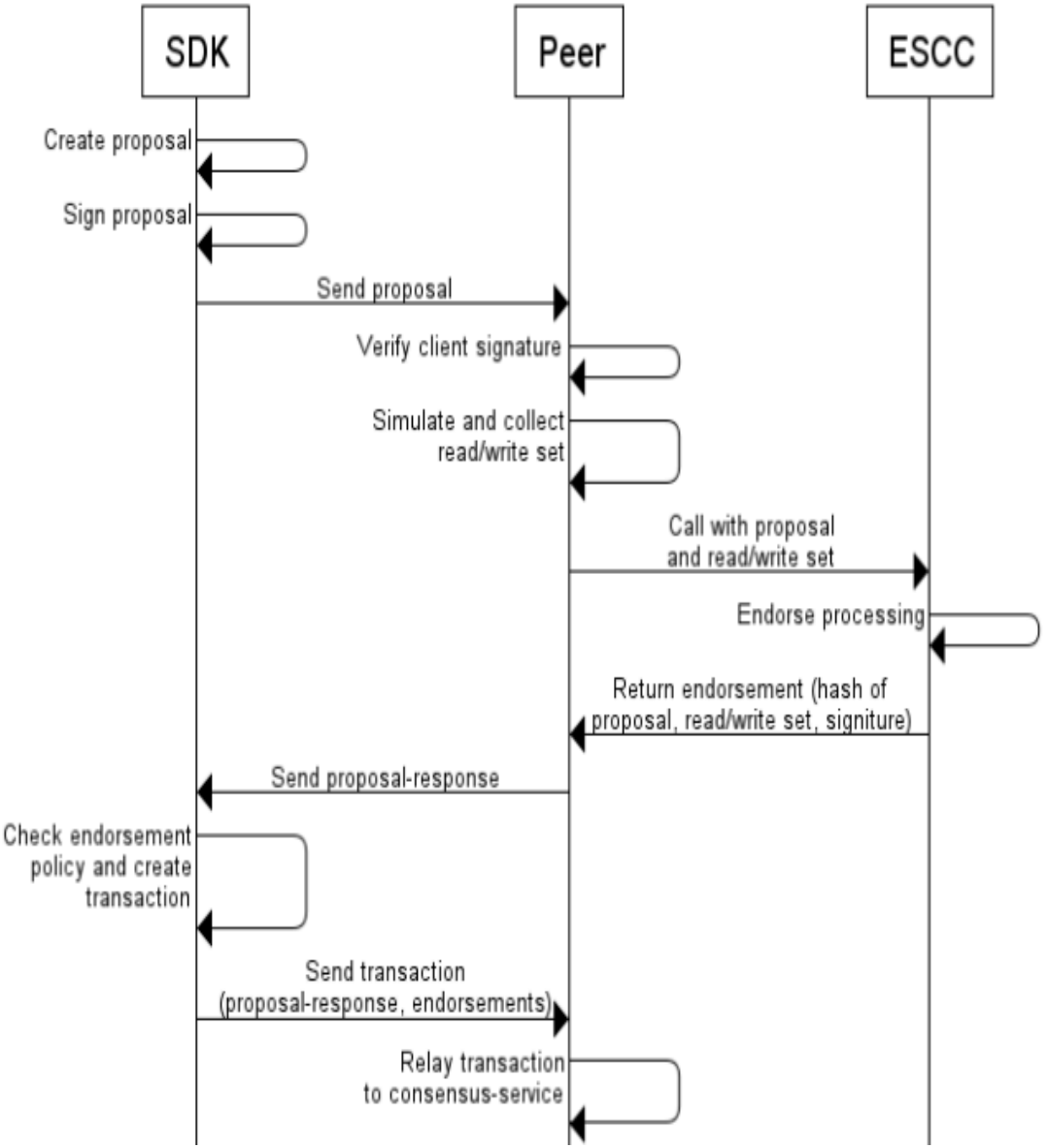
- Each chaincode may be associated with an endorsement and validation system chaincode (ESCC, VSCC)
  - ESCC decides how to endorse a proposal, including simulation and application specifics
  - VSCC decides transaction validity, including correctness of endorsements

Endorsement and Validation Policy

- Each chaincode is associated with an Endorsement and Validation system chaincode, which customers may modify, add more or remove existing ones
  - A chaincode's endorsement and validation should be fixed at chaincode deployment rather than per transaction to avoid inconsistency
  - ESCC endorses a transaction, and VSCC validates the transaction according to the endorsement policy
- We should provide the following implementations of ESCC and VSCC pairs
  - At least one valid signature
  - All endorsers must sign and must be valid
  - Explicit list of signatures; the list is configured as a transaction on the ESCC
  - Percent of valid signatures (ex: 80%); the percentage is specified in a transaction
- To find ESCC and VSCC associated with a chaincode, query the chaincode deployment transaction by sending a query Proposal to the Lifecycle system chaincode

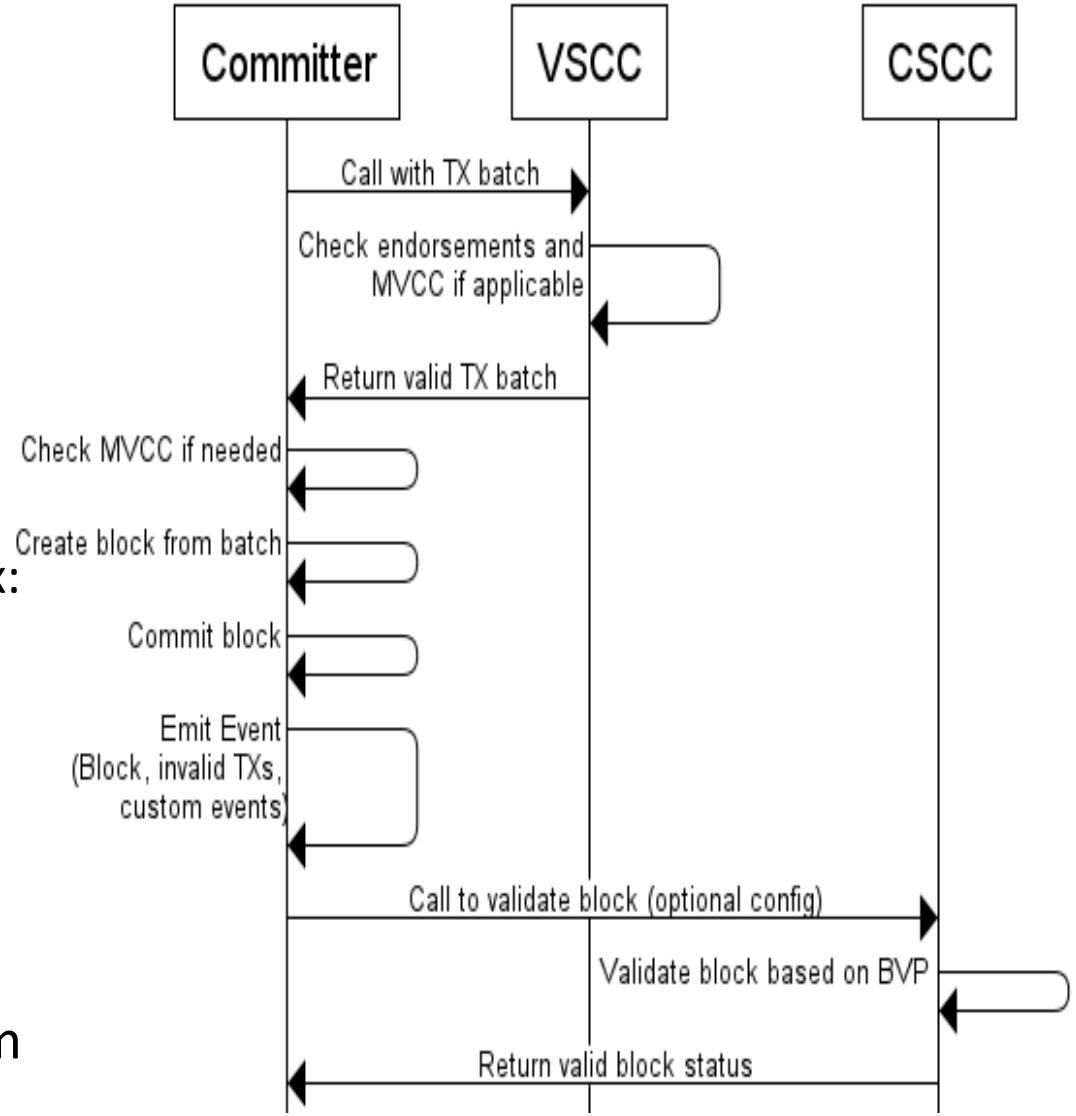
Endorse Transactions

- SDK sends Proposal to Peers based on the chaincode's endorsement policy
  - A peer may relay the proposal on client's behalf
- Endorser system chaincode (ESCC) processes the endorsement
  - ESCC provides ability to customize endorsement
  - Default logic will just sign the Proposal Response
- Client/SDK decides transaction content if endorsement satisfied



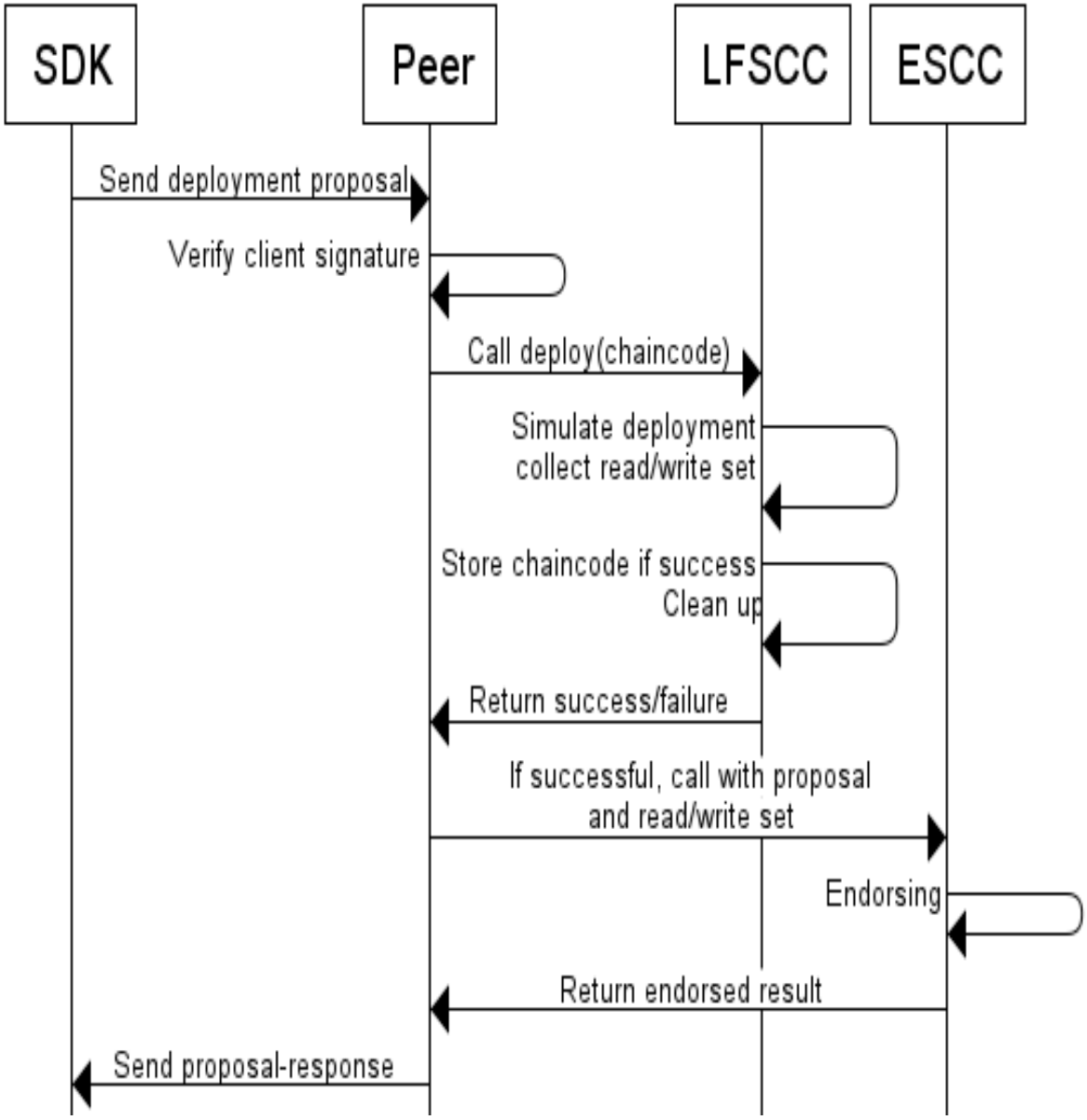
Commit Transactions

- Committing a transaction involves validating each transaction read/write-set and endorsements
- Committer calls Validator system chaincode (VSCC) to validate the batch and commit the block
  - VSCC may perform more sophisticated validation (ex: executing script OP_CHECKSIG in Bitcoin)
- Emit events (block, invalid TXs, custom)
- Block may be optionally validated (ex: checkpoint to detect faults and prune ledger) by Committer system chaincode (CSCC) based on Block Validation Policy



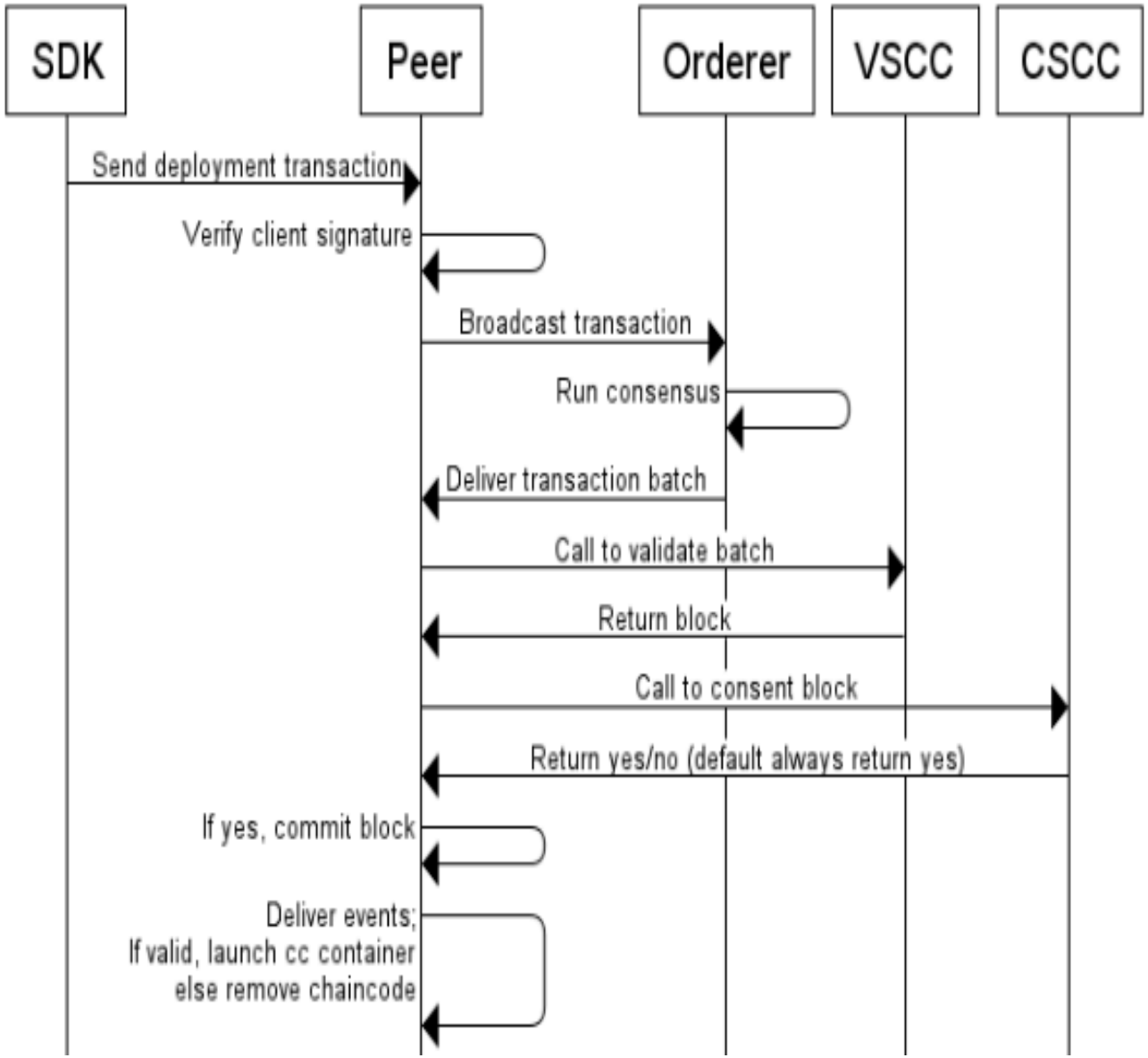
Chaincode Deployment Proposal

- Proposal is a call specification to the Life-Cycle System Chaincode LCSCC to deploy a user chaincode (UCC)
- LCSCC creates version record (version, ucc hash, name) in read-write set
- LCSCC removes container before returning



Chaincode Deployment Transaction

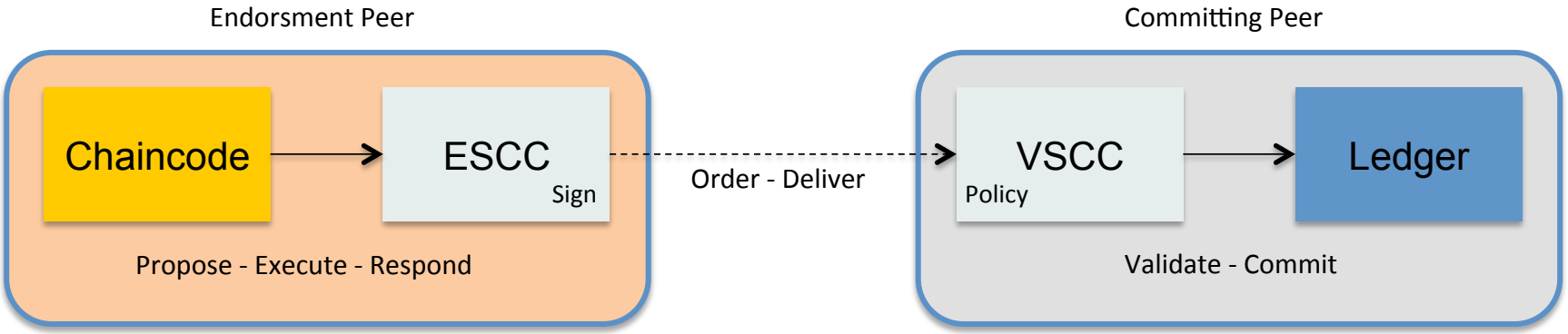
- Deployment transaction records the chaincode address and its initial values onto the ledger
- If the transaction has been successfully committed, launch the container



Endorsement Policies

An endorsement policy describes the conditions by which a transaction can be endorsed. A transaction can only be considered valid if it has been endorsed according to its policy.

- Each chaincode is associated with an Endorsement Policy
- Default implementation: Simple declarative language for the policy
- ESCC (Endorsement System ChainCode) signs the proposal response on the endorsing peer
- VSCC (Validation System ChainCode) validates the endorsements



Endorsement Policy Syntax

```
$ peer chaincode instantiate
-C mychannel
-n mycc
-v 1.0
-p chaincode_example02
-c '{"Args":["init","a", "100",
"b","200"]}'
-P "AND('Org1MSP.member')“
```

This command instantiates the chaincode *mycc* on channel *mychannel* with the policy AND('Org1MSP.member')

Policy Syntax: **EXPR(E[, E...])**

Where **EXPR** is either **AND** or **OR** and **E** is either a principal or nested EXPR.

Principal Syntax: **MSP.ROLE**

Supported roles are: **member** and **admin**.

Where **MSP** is the MSP ID required, and **ROLE** is either “member” or “admin”.

Endorsement Policy Examples

Examples of policies:

- Request 1 signature from all three principals
  - AND('Org1.member', 'Org2.member', 'Org3.member')
- Request 1 signature from either one of the two principals
  - OR('Org1.member', 'Org2.member')
- Request either one signature from a member of the Org1 MSP or (1 signature from a member of the Org2 MSP and 1 signature from a member of the Org3 MSP)
  - OR('Org1.member', AND('Org2.member', 'Org3.member'))

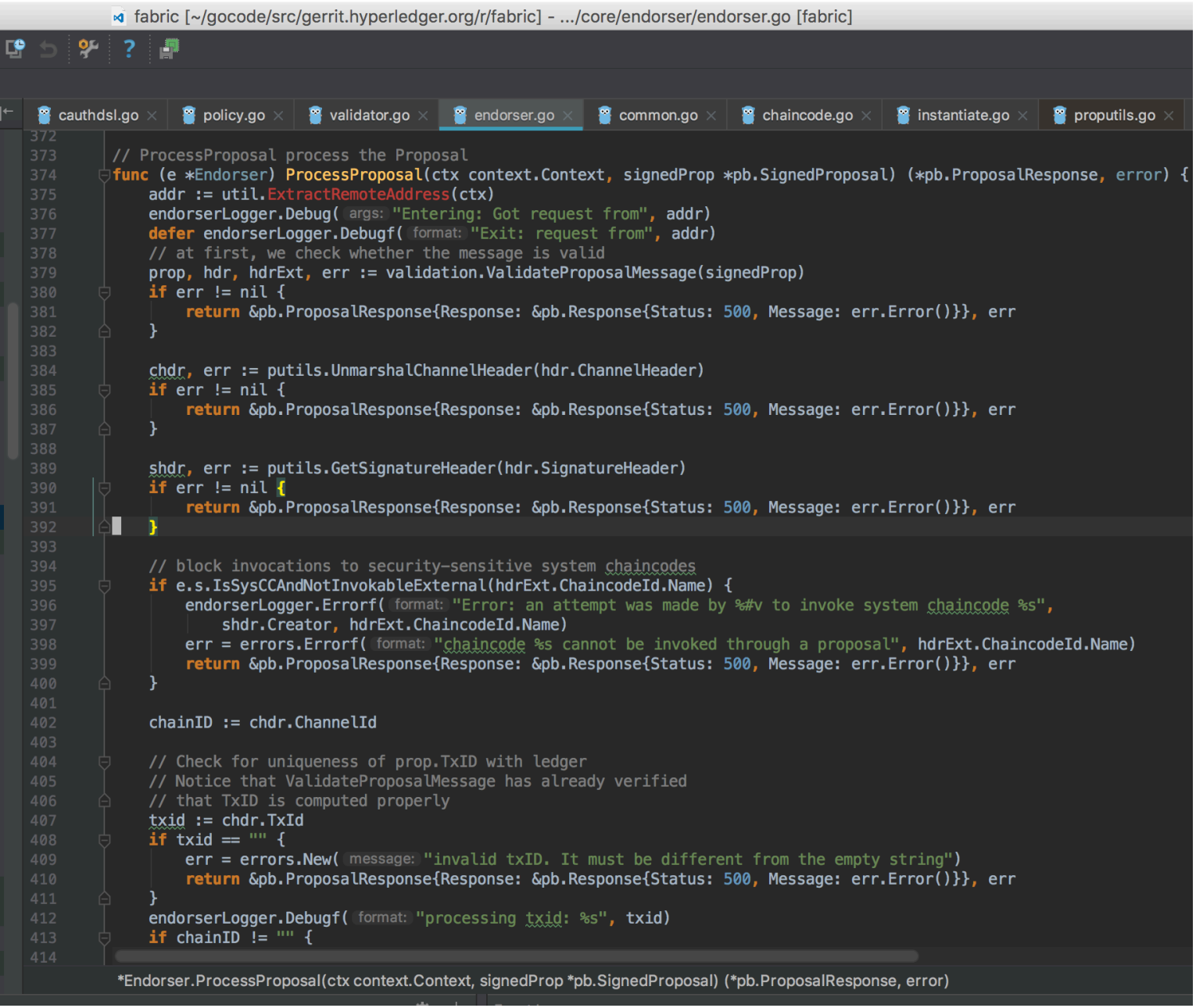
ESCC and VSCC Source Files

- core/endorser/endorser.go:ProcessProposal
- core/committer/txvalidator/validator.go

  Validate→validateTx→

  VSCCValidateTx→VSCCValidateTxForCC
- common/cauthdsl/cauthdsl.go

Endorser Source 1



Endorser Source 2

```
413     if chainID != "" {
414         // here we handle uniqueness check and ACLs for proposals targeting a chain
415         if _, err := e.s.GetTransactionByID(chainID, txid); err == nil {
416             return nil, errors.Errorf( format: "duplicate transaction found [%s]. Creator [%x]", txid, shdr.Creator)
417         }
418
419         // check ACL only for application chaincodes; ACLs
420         // for system chaincodes are checked elsewhere
421         if !e.s.IsSysCC(hdrExt.ChaincodeId.Name) {
422             // check that the proposal complies with the channel's writers
423             if err = e.s.CheckACL(signedProp, chdr, shdr, hdrExt); err != nil {
424                 return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, err
425             }
426         }
427     } else {
428         // chainless proposals do not/cannot affect ledger and cannot be submitted as transactions
429         // ignore uniqueness checks; also, chainless proposals are not validated using the policies
430         // of the chain since by definition there is no chain; they are validated against the local
431         // MSP of the peer instead by the call to ValidateProposalMessage above
432     }
433
434     // obtaining once the tx simulator for this proposal. This will be nil
435     // for chainless proposals
436     // Also obtain a history query executor for history queries, since tx simulator does not cover history
437     var txsim ledger.TxSimulator
438     var historyQueryExecutor ledger.HistoryQueryExecutor
439     if chainID != "" {
440         if txsim, err = e.s.GetTxSimulator(chainID, txid); err != nil {
441             return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, err
442         }
443         if historyQueryExecutor, err = e.s.GetHistoryQueryExecutor(chainID); err != nil {
444             return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, err
445         }
446         // Add the historyQueryExecutor to context
447         // TODO shouldn't we also add txsim to context here as well? Rather than passing txsim parameter
448         // around separately, since eventually it gets added to context anyways
449         ctx = context.WithValue(ctx, chaincode.HistoryQueryExecutorKey, historyQueryExecutor)
450
451         defer txsim.Done()
452     }
```

Endorser Source 3

```
455     // TODO: if the proposal has an extension, it will be of type ChaincodeAction;
456     //       if it's present it means that no simulation is to be performed because
457     //       we're trying to emulate a submitting peer. On the other hand, we need
458     //       to validate the supplied action before endorsing it
459
460     //1 -- simulate
461     cd, res, simulationResult, ccevent, err := e.simulateProposal(ctx, chainID, txid, signedProp, prop, hdrExt.ChaincodeId, txsim)
462     if err != nil {
463         return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, err
464     }
465     if res != nil {
466         if res.Status >= shim.ERROR {
467             endorserLogger.Errorf( format: "simulateProposal() resulted in chaincode response status %d for txid: %s", res.Status, txid)
468             var cceventBytes []byte
469             if ccevent != nil {
470                 cceventBytes, err = putils.GetBytesChaincodeEvent(ccevent)
471                 if err != nil {
472                     return nil, errors.Wrap(err,  message: "failed to marshal event bytes")
473                 }
474             }
475             pResp, err := putils.CreateProposalResponseFailure(prop.Header, prop.Payload, res, simulationResult, cceventBytes, hdrExt.ChaincodeId, hdrExt.PayloadVisibility)
476             if err != nil {
477                 return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, err
478             }
479
480             return pResp, &chaincodeError{ status: res.Status,  msg: res.Message}
481         }
482     }
483
484     //2 -- endorse and get a marshalled ProposalResponse message
485     var pResp *pb.ProposalResponse
486
487     //TODO till we implement global ESCC, CSCC for system chaincodes
488     //chainless proposals (such as CSCC) don't have to be endorsed
489     if chainID == "" {
490         pResp = &pb.ProposalResponse{Response: res}
491     } else {
492         pResp, err = e.endorseProposal(ctx, chainID, txid, signedProp, prop, res, simulationResult, ccevent, hdrExt.PayloadVisibility, hdrExt.ChaincodeId, txsim, cd)
493         if err != nil {
494             return &pb.ProposalResponse{Response: &pb.Response{Status: 500, Message: err.Error()}}, err
495         }
496         if pResp != nil {
497             if res.Status >= shim.ERRORTHRESHOLD {
498                 endorserLogger.Debugf( format: "endorseProposal() resulted in chaincode error for txid: %s", txid)
499                 return pResp, &chaincodeError{ status: res.Status,  msg: res.Message}
500             }
501         }
502     }
503
504     // Set the proposal response payload - it
505     // contains the "return value" from the
506     // chaincode invocation
507     pResp.Response.Payload = res.Payload
508
509     return pResp, nil
510 }
511
```

Endorser Source 4

```
283  //endorse the proposal by calling the ESCC
284  func (e *Endorser) endorseProposal(ctx context.Context, chainID string, txid string, signedProp *pb.SignedProposal,
285      proposal *pb.Proposal, response *pb.Response, simRes []byte, event *pb.ChaincodeEvent, visibility []byte,
286          ccid *pb.ChaincodeID, txsim ledger.TxSimulator, cd resourcesconfig.ChaincodeDefinition) (*pb.ProposalResponse, error) {
287      endorserLogger.Debugf( format: "Entry - txid: %s channel id: %s chaincode id: %s", txid, chainID, ccid)
288      defer endorserLogger.Debugf( format: "Exit")
289
290      isSysCC := cd == nil
291      // 1) extract the name of the escc that is requested to endorse this chaincode
292      var escc string
293      //ie, not "lscc" or system chaincodes
294      if isSysCC {
295          escc = "escc"
296      } else {
297          escc = cd.Endorsement()
298          if escc == "" { // this should never happen, LSCC always fills this field
299              panic( v: "No ESCC specified in ChaincodeData")
300          }
301      }
302
303      endorserLogger.Debugf( format: "info: escc for chaincode id %s is %s", ccid, escc)
304
305      // marshalling event bytes
306      var err error
307      var eventBytes []byte
308      if event != nil {
309          eventBytes, err = putils.GetBytesChaincodeEvent(event)
310          if err != nil {
311              return nil, errors.Wrap(err, message: "failed to marshal event bytes")
312          }
313      }
314
315      resBytes, err := putils.GetBytesResponse(response)
316      if err != nil {
317          return nil, errors.Wrap(err, message: "failed to marshal response bytes")
318      }
319
320      // set version of executing chaincode
321      if isSysCC {
322          // if we want to allow mixed fabric levels we should
323          // set syscc version to ""
324          ccid.Version = util.GetSysCCVersion()
325      } else {
326          ccid.Version = cd.CCVersion()
327      }
328
329      ccidBytes, err := putils.Marshal(ccid)
330      if err != nil {
331          return nil, errors.Wrap(err, message: "failed to marshal ChaincodeID")
332      }
```

Endorser Source 5

```
334     // 3) call the ESCC we've identified
335     // arguments:
336     // args[0] - function name (not used now)
337     // args[1] - serialized Header object
338     // args[2] - serialized ChaincodeProposalPayload object
339     // args[3] - ChaincodeID of executing chaincode
340     // args[4] - result of executing chaincode
341     // args[5] - binary blob of simulation results
342     // args[6] - serialized events
343     // args[7] - payloadVisibility
344     args := [][]byte{[]byte(""), proposal.Header, proposal.Payload, ccidBytes, resBytes, simRes, eventBytes, visibility}
345     version := util.GetSysCCVersion()
346     ecccis := &pb.ChaincodeInvocationSpec{ChaincodeSpec: &pb.ChaincodeSpec{Type: pb.ChaincodeSpec_GOLANG, ChaincodeId: &pb.ChaincodeID{Name: escc}, Input: &pb.ChaincodeInput{Args: args}}}
347     res, _, err := e.callChaincode(ctx, chainID, version, txid, signedProp, proposal, ecccis, &pb.ChaincodeID{Name: escc}, txsim)
348     if err != nil {
349         return nil, err
350     }
351
352     if res.Status >= shim.ERRORTHRESHOLD {
353         return &pb.ProposalResponse{Response: res}, nil
354     }
355
356     prBytes := res.Payload
357     // Note that we do not extract any simulation results from
358     // the call to ESCC. This is intentional becuse ESCC is meant
359     // to endorse (i.e. sign) the simulation results of a chaincode,
360     // but it can't obviously sign its own. Furthermore, ESCC runs
361     // on private input (its own signing key) and so if it were to
362     // produce simulationr results, they are likely to be different
363     // from other ESCCs, which would stand in the way of the
364     // endorsement process.
365
366     //3 -- respond
367     pResp, err := putils.GetProposalResponse(prBytes)
368     if err != nil {
369         return nil, err
370     }
371
372     return pResp, nil
373 }
```

Validation Source 1

```go
// Validate performs the validation of a block. The validation
// of each transaction in the block is performed in parallel.
// The approach is as follows: the committer thread starts the
// tx validation function in a goroutine (using a semaphore to cap
// the number of concurrent validating goroutines). The committer
// thread then reads results of validation (in orderer of completion
// of the goroutines) from the results channel. The goroutines
// perform the validation of the txs in the block and enqueue the
// validation result in the results channel. A few note-worthy facts:
// 1) to keep the approach simple, the committer thread enqueues
//    all transactions in the block and then moves on to reading the
//    results.
// 2) for parallel validation to work, it is important that the
//    validation function does not change the state of the system.
//    Otherwise the order in which validation is perform matters
//    and we have to resort to sequential validation (or some locking).
//    This is currently true, because the only function that affects
//    state is when a config transaction is received, but they are
//    guaranteed to be alone in the block. If/when this assumption
//    is violated, this code must be changed.
func (v *txValidator) Validate(block *common.Block) error {
    var err error
    var errPos int

    logger.Debug( args: "START Block Validation")
    defer logger.Debug( args: "END Block Validation")
    // Initialize trans as valid here, then set invalidation reason code upon invalidation below
    txsfltr := ledgerUtil.NewTxValidationFlags(len(block.Data.Data))
    // txsChaincodeNames records all the invoked chaincodes by tx in a block
    txsChaincodeNames := make(map[int]*sysccprovider.ChaincodeInstance)
    // upgradedChaincodes records all the chaincodes that are upgraded in a block
    txsUpgradedChaincodes := make(map[int]*sysccprovider.ChaincodeInstance)
    // array of txids
    txidArray := make([]string, len(block.Data.Data))

    results := make(chan *blockValidationResult)
    go func() {
        for tIdx, d := range block.Data.Data {
            tIdxLcl := tIdx
            dLcl := d

            // ensure that we don't have too many concurrent validation workers
            v.support.Acquire(context.Background(), n: 1)

            go func() {
                defer v.support.Release( n: 1)

                validateTx(&blockValidationRequest{
                    d:     dLcl,
                    block: block,
                    tIdx:  tIdxLcl,
                    v:     v,
                }, results)
            }()
        }
    }()

    logger.Debugf( format: "expecting %d block validation responses", len(block.Data.Data))

    // now we read responses in the order in which they come back
    for i := 0; i < len(block.Data.Data); i++ {
        res := <-results

        if res.err != nil {
            // if there is an error, we buffer its value, wait for
            // all workers to complete validation and then return
            // the error from the first tx in this block that returned an error
            logger.Debugf( format: "got terminal error %s for idx %d", res.err, res.tIdx)

            if err == nil || res.tIdx < errPos {
                err = res.err
                errPos = res.tIdx
            }
        } else {
            // if there was no error, we set the txsfltr and we set the
            // txsChaincodeNames and txsUpgradedChaincodes maps
            logger.Debugf( format: "got result for idx %d, code %d", res.tIdx, res.validationCode)

            txsfltr.SetFlag(res.tIdx, res.validationCode)

            if res.validationCode == peer.TxValidationCode_VALID {
                if res.txsChaincodeName != nil {
                    txsChaincodeNames[res.tIdx] = res.txsChaincodeName
                }
                if res.txsUpgradedChaincode != nil {
                    txsUpgradedChaincodes[res.tIdx] = res.txsUpgradedChaincode
                }
                txidArray[res.tIdx] = res.txid
            }
        }
    }

    // if we're here, all workers have completed the validation.
    // If there was an error we return the error from the first
    // tx in this block that returned an error
    if err != nil {
        return err
    }

    // if we operate with this capability, we mark invalid any transaction that has a txid
    // which is equal to that of a previous tx in this block
    if v.support.Capabilities().ForbidDuplicateTXIdInBlock() {
        markTXIdDuplicates(txidArray, txsfltr)
    }

    // if we're here, all workers have completed validation and
    // no error was reported; we set the tx filter and return
    // success

    txsfltr = v.invalidTXsForUpgradeCC(txsChaincodeNames, txsUpgradedChaincodes, txsfltr)

    // Initialize metadata structure
    utils.InitBlockMetadata(block)

    block.Metadata.Metadata[common.BlockMetadataIndex_TRANSACTIONS_FILTER] = txsfltr

    return nil
}
```

Validation Source 2

```
302  func validateTx(req *blockValidationRequest, results chan<- *blockValidationResult) {
303      block := req.block
304      d := req.d
305      tIdx := req.tIdx
306      v := req.v
307      txID := ""
308
309      if d == nil {
310          results <- &blockValidationResult{
311              tIdx: tIdx,
312          }
313          return
314      }
315
316      if env, err := utils.GetEnvelopeFromBlock(d); err != nil {
317          logger.Warningf( format: "Error getting tx from block(%s)", err)
318          results <- &blockValidationResult{
319              tIdx:           tIdx,
320              validationCode: peer.TxValidationCode_INVALID_OTHER_REASON,
321          }
322          return
323      } else if env != nil {
324          // validate the transaction: here we check that the transaction
325          // is properly formed, properly signed and that the security
326          // chain binding proposal to endorsements to tx holds. We do
327          // NOT check the validity of endorsements, though. That's a
328          // job for VSCC below
329          logger.Debugf( format: "validateTx starts for block %p env %p txn %d", block, env, tIdx)
330          defer logger.Debugf( format: "validateTx completes for block %p env %p txn %d", block, env, tIdx)
331          var payload *common.Payload
332          var err error
333          var txResult peer.TxValidationCode
334          var txsChaincodeName *sysccprovider.ChaincodeInstance
335          var txsUpgradedChaincode *sysccprovider.ChaincodeInstance
336
337          if payload, txResult = validation.ValidateTransaction(env, v.support.Capabilities()); txResult != peer.TxValidationCode_VALID {
338              logger.Errorf( format: "Invalid transaction with index %d", tIdx)
339              results <- &blockValidationResult{
340                  tIdx:           tIdx,
341                  validationCode: txResult,
342              }
343              return
344          }
345
346          chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)
347          if err != nil {
348              logger.Warningf( format: "Could not unmarshal channel header, err %s, skipping", err)
349              results <- &blockValidationResult{
350                  tIdx:           tIdx,
351                  validationCode: peer.TxValidationCode_INVALID_OTHER_REASON,
352              }
353              return
354          }
```

Validation Source 3

```
356         channel := chdr.ChannelId
357         logger.Debugf( format: "Transaction is for chain %s", channel)
358
359         if !v.chainExists(channel) {
360             logger.Errorf( format: "Dropping transaction for non-existent chain %s", channel)
361             results <- &blockValidationResult{
362                 tIdx:           tIdx,
363                 validationCode: peer.TxValidationCode_TARGET_CHAIN_NOT_FOUND,
364             }
365             return
366         }
367
368         if common.HeaderType(chdr.Type) == common.HeaderType_ENDORSER_TRANSACTION {
369             // Check duplicate transactions
370             txID = chdr.TxId
371             if _, err := v.support.Ledger().GetTransactionByID(txID); err == nil {
372                 logger.Error( args: "Duplicate transaction found, ", txID, ", skipping")
373                 results <- &blockValidationResult{
374                     tIdx:           tIdx,
375                     validationCode: peer.TxValidationCode_DUPLICATE_TXID,
376                 }
377                 return
378             }
379
380             // Validate tx with vscc and policy
381             logger.Debug( args: "Validating transaction vscc tx validate")
382             err, cde := v.vscc.VSCCValidateTx(payload, d, env)
383             if err != nil {
384                 logger.Errorf( format: "VSCCValidateTx for transaction txId = %s returned error %s", txID, err)
385                 switch err.(type) {
386                 case *VSCCExecutionFailureError:
387                     results <- &blockValidationResult{
388                         tIdx: tIdx,
389                         err:  err,
390                     }
391                     return
392                 case *VSCCInfoLookupFailureError:
393                     results <- &blockValidationResult{
394                         tIdx: tIdx,
395                         err:  err,
396                     }
397                     return
398                 default:
399                     results <- &blockValidationResult{
400                         tIdx:           tIdx,
401                         validationCode: cde,
402                     }
403                     return
404                 }
405             }
406
407             invokeCC, upgradeCC, err := v.getTxCCInstance(payload)
408             if err != nil {
409                 logger.Errorf( format: "Get chaincode instance from transaction txId = %s returned error %s", txID, err)
410                 results <- &blockValidationResult{
411                     tIdx:           tIdx,
412                     validationCode: peer.TxValidationCode_INVALID_OTHER_REASON,
413                 }
414                 return
415             }
```

Validation Source 4

```
416             txsChaincodeName = invokeCC
417             if upgradeCC != nil {
418                 logger.Infof( format: "Find chaincode upgrade transaction for chaincode %s on chain %s with new version %s",
419                 txsUpgradedChaincode = upgradeCC
420             }
421         } else if common.HeaderType(chdr.Type) == common.HeaderType_CONFIG {
422             configEnvelope, err := configtx.UnmarshalConfigEnvelope(payload.Data)
423             if err != nil {
424                 err := fmt.Errorf( format: "Error unmarshaling config which passed initial validity checks: %s", err)
425                 logger.Critical(err)
426                 results <- &blockValidationResult{
427                     tIdx: tIdx,
428                     err:  err,
429                 }
430                 return
431             }
432
433             if err := v.support.Apply(configEnvelope); err != nil {
434                 err := fmt.Errorf( format: "Error validating config which passed initial validity checks: %s", err)
435                 logger.Critical(err)
436                 results <- &blockValidationResult{
437                     tIdx: tIdx,
438                     err:  err,
439                 }
440                 return
441             }
442             logger.Debugf( format: "config transaction received for chain %s", channel)
443         } else if common.HeaderType(chdr.Type) == common.HeaderType_PEER_RESOURCE_UPDATE {
444             // FIXME: in the context of FAB-7341, we should introduce validation
445             //        for this kind of transaction here. For now we just ignore this
446             //        type of transaction and delegate its validation to other components
447
448             results <- &blockValidationResult{
449                 tIdx: tIdx,
450                 err:  nil,
451             }
452             return
453         } else {
454             logger.Warningf( format: "Unknown transaction type [%s] in block number [%d] transaction index [%d]",
455                 common.HeaderType(chdr.Type), block.Header.Number, tIdx)
456             results <- &blockValidationResult{
457                 tIdx:           tIdx,
458                 validationCode: peer.TxValidationCode_UNKNOWN_TX_TYPE,
459             }
460             return
461         }
462
463         if _, err := proto.Marshal(env); err != nil {
464             logger.Warningf( format: "Cannot marshal transaction due to %s", err)
465             results <- &blockValidationResult{
466                 tIdx:           tIdx,
467                 validationCode: peer.TxValidationCode_MARSHAL_TX_ERROR,
468             }
469             return
470         }
```

Validation Source 5

```
471             // Succeeded to pass down here, transaction is valid
472             results <- &blockValidationResult{
473                 tIdx:                 tIdx,
474                 txsChaincodeName:     txsChaincodeName,
475                 txsUpgradedChaincode: txsUpgradedChaincode,
476                 validationCode:       peer.TxValidationCode_VALID,
477                 txid:                 txID,
478             }
479             return
480         } else {
481             logger.Warning( args: "Nil tx from block")
482             results <- &blockValidationResult{
483                 tIdx:           tIdx,
484                 validationCode: peer.TxValidationCode_NIL_ENVELOPE,
485             }
486             return
487         }
488     }
```

Validation Source 6

```
680  func (v *vsccValidatorImpl) VSCCValidateTx(payload *common.Payload, envBytes []byte, env *common.Envelope) (error, peer.TxValidationCode) {
681      logger.Debugf( format: "VSCCValidateTx starts for env %p envbytes %p", env, envBytes)
682      defer logger.Debugf( format: "VSCCValidateTx completes for env %p envbytes %p", env, envBytes)
683
684      // get header extensions so we have the chaincode ID
685      hdrExt, err := utils.GetChaincodeHeaderExtension(payload.Header)
686      if err != nil {
687          return err, peer.TxValidationCode_BAD_HEADER_EXTENSION
688      }
689
690      // get channel header
691      chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)
692      if err != nil {
693          return err, peer.TxValidationCode_BAD_CHANNEL_HEADER
694      }
695
696      /* obtain the list of namespaces we're writing stuff to;
697         at first, we establish a few facts about this invocation:
698         1) which namespaces does it write to?
699         2) does it write to LSCC's namespace?
700         3) does it write to any cc that cannot be invoked? */
701      wrNamespace := []string{}
702      writesToLSCC := false
703      writesToNonInvokableSCC := false
704      respPayload, err := utils.GetActionFromEnvelope(envBytes)
705      if err != nil {
706          return fmt.Errorf( format: "GetActionFromEnvelope failed, error %s", err), peer.TxValidationCode_BAD_RESPONSE_PAYLOAD
707      }
708      txRWSet := &rwsetutil.TxRwSet{}
709      if err = txRWSet.FromProtoBytes(respPayload.Results); err != nil {
710          return fmt.Errorf( format: "txRWSet.FromProtoBytes failed, error %s", err), peer.TxValidationCode_BAD_RWSET
711      }
712      for _, ns := range txRWSet.NsRwSets {
713          if v.txWritesToNamespace(ns) {
714              wrNamespace = append(wrNamespace, ns.NameSpace)
715
716              if !writesToLSCC && ns.NameSpace == "lscc" {
717                  writesToLSCC = true
718              }
719
720              if !writesToNonInvokableSCC && v.sccprovider.IsSysCCAndNotInvokableCC2CC(ns.NameSpace) {
721                  writesToNonInvokableSCC = true
722              }
723
724              if !writesToNonInvokableSCC && v.sccprovider.IsSysCCAndNotInvokableExternal(ns.NameSpace) {
725                  writesToNonInvokableSCC = true
726              }
727          }
728      }
```

Validation Source 7

```
730         // get name and version of the cc we invoked
731         ccID := hdrExt.ChaincodeId.Name
732         ccVer := respPayload.ChaincodeId.Version
733
734         // sanity check on ccID
735         if ccID == "" {
736             err := fmt.Errorf( format: "invalid chaincode ID")
737             logger.Errorf( format: "%s", err)
738             return err, peer.TxValidationCode_INVALID_OTHER_REASON
739         }
740         if ccID != respPayload.ChaincodeId.Name {
741             err := fmt.Errorf( format: "inconsistent ccid info (%s/%s)", ccID, respPayload.ChaincodeId.Name)
742             logger.Errorf( format: "%s", err)
743             return err, peer.TxValidationCode_INVALID_OTHER_REASON
744         }
745         // sanity check on ccver
746         if ccVer == "" {
747             err := fmt.Errorf( format: "invalid chaincode version")
748             logger.Errorf( format: "%s", err)
749             return err, peer.TxValidationCode_INVALID_OTHER_REASON
750         }
751
752         // we've gathered all the info required to proceed to validation;
753         // validation will behave differently depending on the type of
754         // chaincode (system vs. application)
755
756         if !v.sccprovider.IsSysCC(ccID) {
757             // if we're here, we know this is an invocation of an application chaincode;
758             // first of all, we make sure that:
759             // 1) we don't write to LSCC - an application chaincode is free to invoke LSCC
760             //    for instance to get information about itself or another chaincode; however
761             //    these legitimate invocations only ready from LSCC's namespace; currently
762             //    only two functions of LSCC write to its namespace: deploy and upgrade and
763             //    neither should be used by an application chaincode
764             if writesToLSCC {
765                 return fmt.Errorf( format: "Chaincode %s attempted to write to the namespace of LSCC", ccID),
766                     peer.TxValidationCode_ILLEGAL_WRITESET
767             }
768             // 2) we don't write to the namespace of a chaincode that we cannot invoke - if
769             //    the chaincode cannot be invoked in the first place, there's no legitimate
770             //    way in which a transaction has a write set that writes to it; additionally
771             //    we don't have any means of verifying whether the transaction had the rights
772             //    to perform that write operation because in v1, system chaincodes do not have
773             //    any endorsement policies to speak of. So if the chaincode can't be invoked
774             //    it can't be written to by an invocation of an application chaincode
775             if writesToNonInvokableSCC {
776                 return fmt.Errorf( format: "Chaincode %s attempted to write to the namespace of a system chaincode that cannot be invoked", ccID),
777                     peer.TxValidationCode_ILLEGAL_WRITESET
778             }
779
780             // validate *EACH* read write set according to its chaincode's endorsement policy
781             for _, ns := range wrNamespace {
782                 // Get latest chaincode version, vscc and validate policy
783                 txcc, vscc, policy, err := v.GetInfoForValidate(chdr.TxId, chdr.ChannelId, ns)
784                 if err != nil {
785                     logger.Errorf( format: "GetInfoForValidate for txId = %s returned error %s", chdr.TxId, err)
786                     return err, peer.TxValidationCode_INVALID_OTHER_REASON
787                 }
```

Validation Source 8

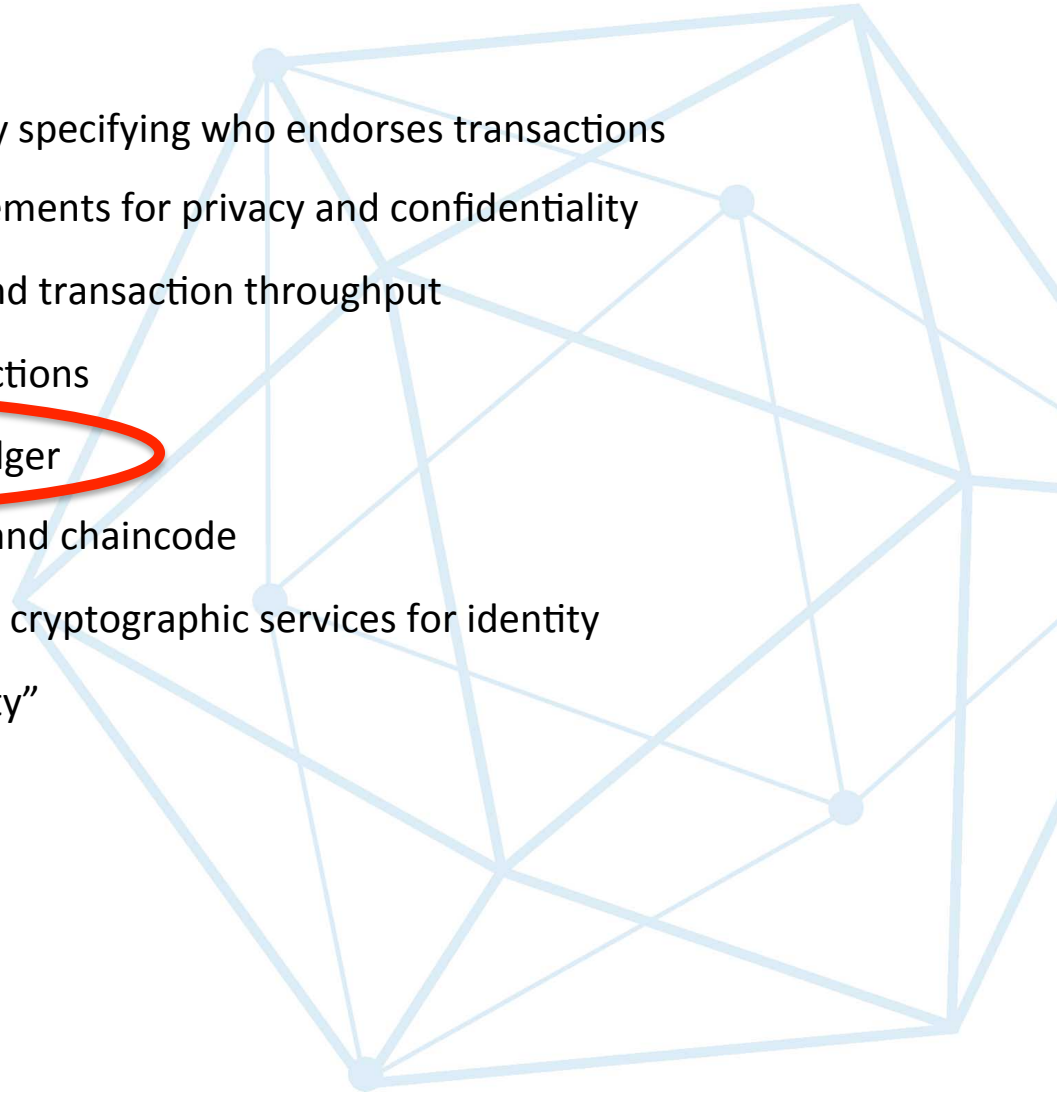
```
789             // if the namespace corresponds to the cc that was originally
790             // invoked, we check that the version of the cc that was
791             // invoked corresponds to the version that lscc has returned
792             if ns == ccID && txcc.ChaincodeVersion != ccVer {
793                 err := fmt.Errorf( format: "Chaincode %s:%s/%s didn't match %s:%s/%s in lscc", ccID, ccVer, chdr.ChannelId, txcc.ChaincodeName, txcc.ChaincodeVersion, chdr.ChannelId)
794                 logger.Errorf(err.Error())
795                 return err, peer.TxValidationCode_EXPIRED_CHAINCODE
796             }
797
798             // do VSCC validation
799             if err = v.VSCCValidateTxForCC(envBytes, chdr.TxId, chdr.ChannelId, vscc.ChaincodeName, vscc.ChaincodeVersion, policy); err != nil {
800                 switch err.(type) {
801                 case *VSCCEndorsementPolicyError:
802                     return err, peer.TxValidationCode_ENDORSEMENT_POLICY_FAILURE
803                 default:
804                     return err, peer.TxValidationCode_INVALID_OTHER_REASON
805                 }
806             }
807         }
808     } else {
809         // make sure that we can invoke this system chaincode - if the chaincode
810         // cannot be invoked through a proposal to this peer, we have to drop the
811         // transaction; if we didn't, we wouldn't know how to decide whether it's
812         // valid or not because in v1, system chaincodes have no endorsement policy
813         if v.sccprovider.IsSysCCAndNotInvokableExternal(ccID) {
814             return fmt.Errorf( format: "Committing an invocation of cc %s is illegal", ccID),
815                 peer.TxValidationCode_ILLEGAL_WRITESET
816         }
817
818         // Get latest chaincode version, vscc and validate policy
819         _, vscc, policy, err := v.GetInfoForValidate(chdr.TxId, chdr.ChannelId, ccID)
820         if err != nil {
821             logger.Errorf( format: "GetInfoForValidate for txId = %s returned error %s", chdr.TxId, err)
822             return err, peer.TxValidationCode_INVALID_OTHER_REASON
823         }
824
825         // validate the transaction as an invocation of this system chaincode;
826         // vscc will have to do custom validation for this system chaincode
827         // currently, VSCC does custom validation for LSCC only; if an hlf
828         // user creates a new system chaincode which is invokable from the outside
829         // they have to modify VSCC to provide appropriate validation
830         if err = v.VSCCValidateTxForCC(envBytes, chdr.TxId, vscc.ChainID, vscc.ChaincodeName, vscc.ChaincodeVersion, policy); err != nil {
831             switch err.(type) {
832             case *VSCCEndorsementPolicyError:
833                 return err, peer.TxValidationCode_ENDORSEMENT_POLICY_FAILURE
834             default:
835                 return err, peer.TxValidationCode_INVALID_OTHER_REASON
836             }
837         }
838     }
839
840     return nil, peer.TxValidationCode_VALID
841 }
```

Validation Source 9

```
843  func (v *vsccValidatorImpl) VSCCValidateTxForCC(envBytes []byte, txid, chid, vsccName, vsccVer string, policy []byte) error {
844      logger.Debugf( format: "VSCCValidateTxForCC starts for envbytes %p", envBytes)
845      defer logger.Debugf( format: "VSCCValidateTxForCC completes for envbytes %p", envBytes)
846      ctxt, txsim, err := v.ccprovider.GetContext(v.support.Ledger(), txid)
847      if err != nil {
848          msg := fmt.Sprintf( format: "Cannot obtain context for txid=%s, err %s", txid, err)
849          logger.Errorf(msg)
850          return &VSCCExecutionFailureError{ reason: msg}
851      }
852      defer txsim.Done()
853
854      // build arguments for VSCC invocation
855      // args[0] - function name (not used now)
856      // args[1] - serialized Envelope
857      // args[2] - serialized policy
858      args := [][]byte{[]byte(""), envBytes, policy}
859
860      // get context to invoke VSCC
861      vscctxid := coreUtil.GenerateUUID()
862      cccid := v.ccprovider.GetCCContext(chid, vsccName, vsccVer, vscctxid,  syscc: true,  signedProp: nil,  prop: nil)
863
864      // invoke VSCC
865      logger.Debug( args: "Invoking VSCC txid", txid, "chaindID", chid)
866      res, _, err := v.ccprovider.ExecuteChaincode(ctxt, cccid, args)
867      if err != nil {
868          msg := fmt.Sprintf( format: "Invoke VSCC failed for transaction txid=%s, error %s", txid, err)
869          logger.Errorf(msg)
870          return &VSCCExecutionFailureError{ reason: msg}
871      }
872      if res.Status != shim.OK {
873          logger.Errorf( format: "VSCC check failed for transaction txid=%s, error %s", txid, res.Message)
874          return &VSCCEndorsementPolicyError{ reason: fmt.Sprintf( format: "%s", res.Message)}
875      }
876
877      return nil
878  }
```

Pluggable, Queryable World State

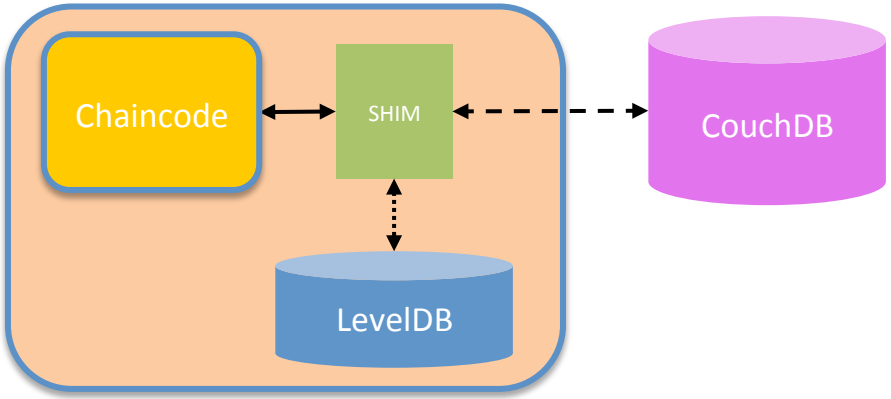
Overview of Hyperledger Fabric v1 – Design Goals



- Better reflect business processes by specifying who endorses transactions
- Support broader regulatory requirements for privacy and confidentiality
- Scale the number of participants and transaction throughput
- Eliminate non deterministic transactions
- Support rich data queries of the ledger
- Dynamically upgrade the network and chaincode
- Support for multiple credential and cryptographic services for identity
- Support for ”bring your own identity”

World State Database

- Pluggable worldstate database
- Default embedded key/value implementation using LevelDB
  - Support for keyed queries, but cannot query on value
- Support for Apache CouchDB
  - Full query support on key and value (JSON documents)
  - Meets a large range of chaincode, auditing, and reporting requirements
  - Will support reporting and analytics via data replication to an analytics engine such as Spark (future)
  - Id/document data model compatible with existing chaincode key/value programming model



Ledger v1 Objectives

Support v1 endorsement/consensus model - separation of simulation (chaincode execution) and block validation/commit

- Chaincode execution simulated on 'endorsing' peers (subset of 'committing' peers)
- Transaction validated and committed on all 'committing' peers
- Parallel simulation enabled on endorsers for improved concurrency and scalability

Persist transaction read/write sets on the blockchain

- Immutability, Auditing, Provenance

Remove dependence on RocksDB

- Utilize LevelDB instead

Optimize data storage for blockchain use patterns

- New file-based blockchain ledger for immutable transaction log
- LevelDB indexes against file-based ledger for efficient lookups
- LevelDB key/value state database for transaction execution (by default)

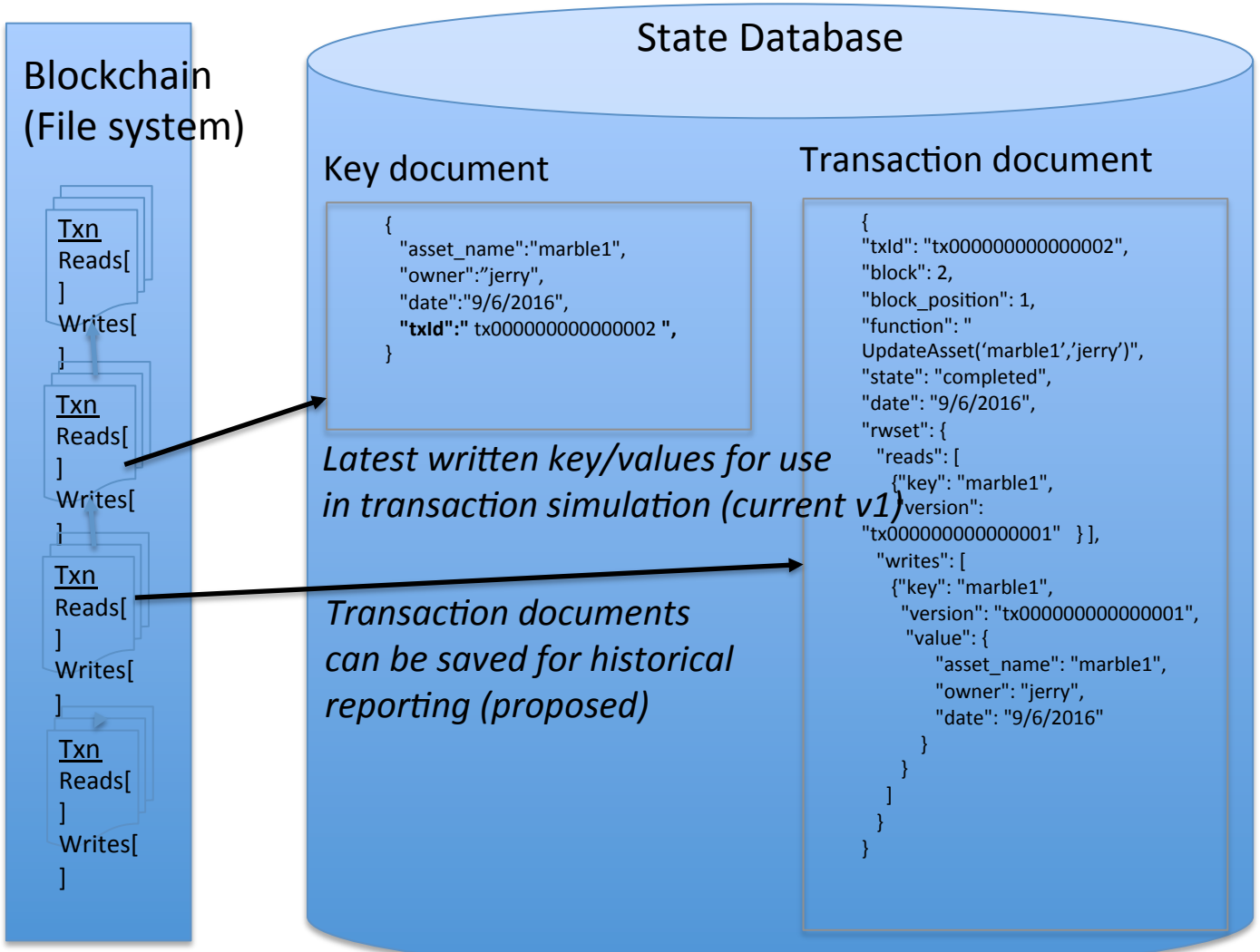
Enrich query capability of data in the blockchain

- Efficient non-key queries
- Historical queries (simple provenance scenarios)

Support for plugging in external state databases

- First external database is CouchDB – supports rich data query when modeling chaincode data as JSON

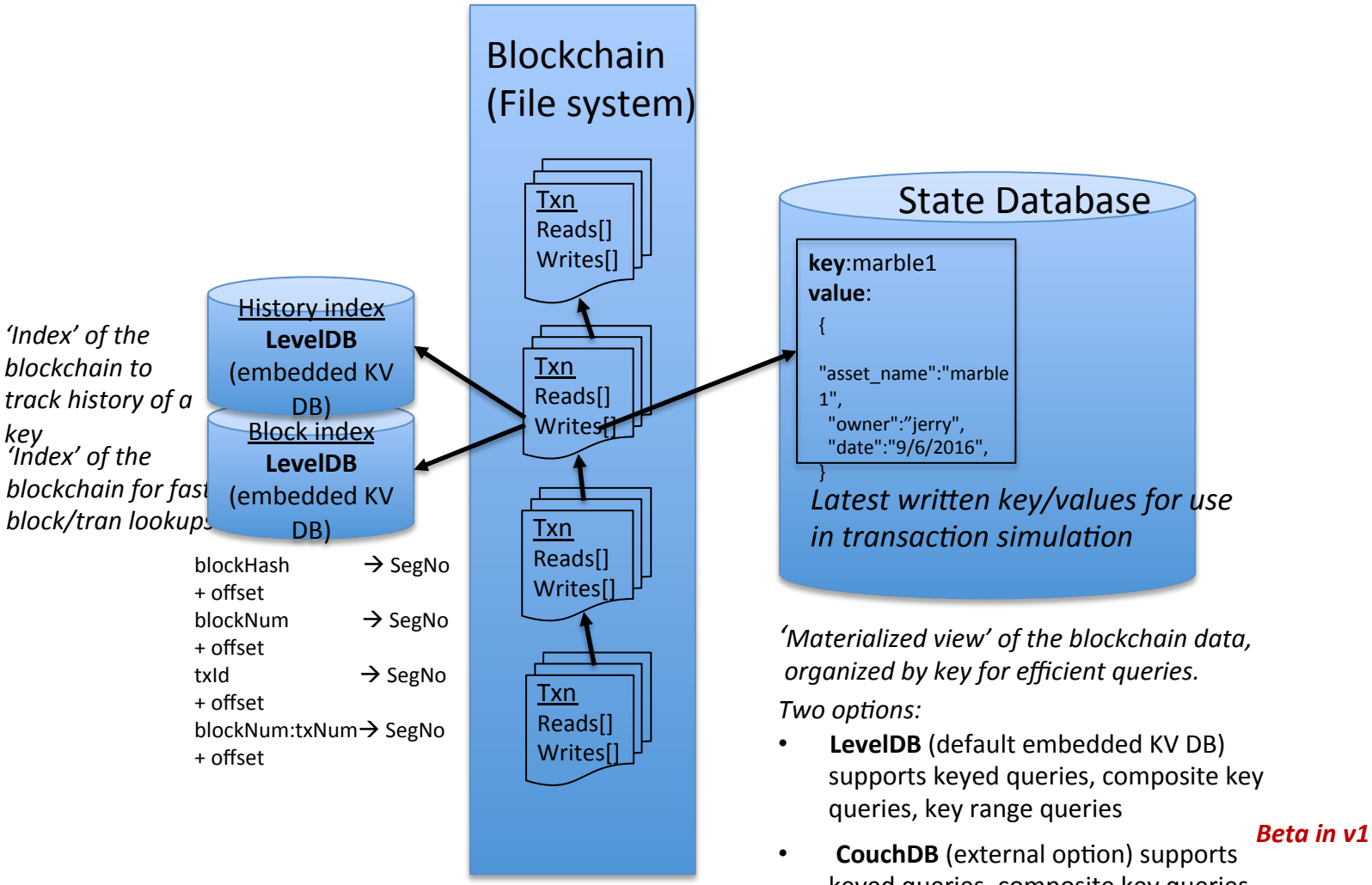
Ledger v1



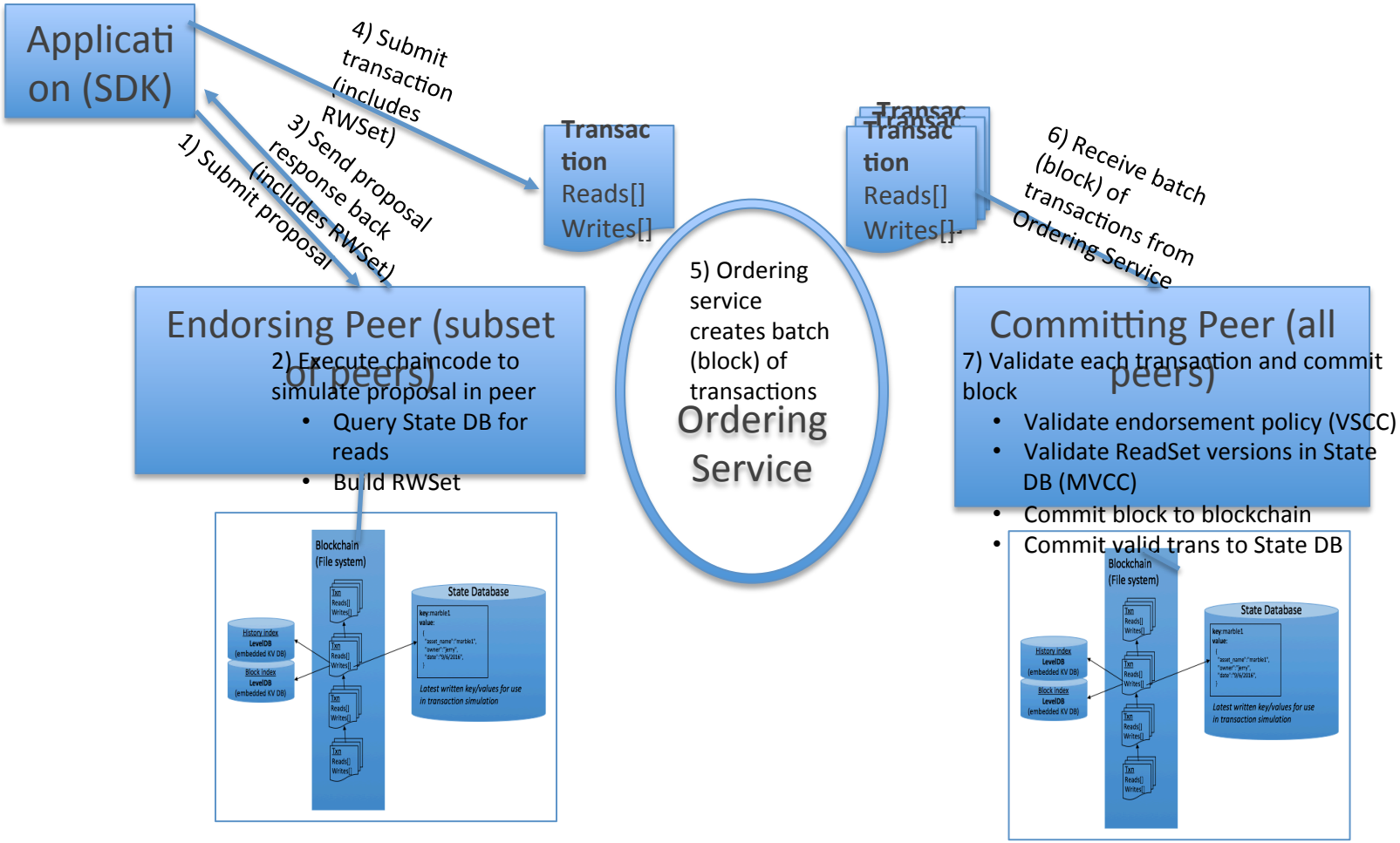
Immutable source of truth

'Index' of the blockchain for runtime queries

Ledger v1


Blockchain
(File system)
Txn
Reads[]
Writes[]
Txn
Reads[]
Writes[]
Txn
Reads[]
Writes[]
Txn
Reads[]
Writes[]
State Database
key:marble1
value:
{
"asset_name":"marble1",
"owner":"jerry",
"date":"9/6/2016",
}
Latest written key/values for use in transaction simulation
History index
LevelDB
(embedded KV DB)
Block index
LevelDB
(embedded KV DB)
'Index' of the blockchain to track history of a key
'Index' of the blockchain for fast block/tran lookups
blockHash → SegNo + offset
blockNum → SegNo + offset
txId → SegNo + offset
blockNum:txNum → SegNo + offset
'Materialized view' of the blockchain data, organized by key for efficient queries.
Two options:
• LevelDB (default embedded KV DB) supports keyed queries, composite key queries, key range queries
• CouchDB (external option) supports keyed queries, composite key queries, key range queries, plus full data rich queries
Beta in v1
Immutable source of truth

v1 Transaction Lifecycle



Logical structure of a ReadWriteSet

```
Block{
   Transactions [
      {
         "Id" : txUUID2
         "Invoke" : “Method(arg1, arg2,..,argN)"
         “TxRWSet" : [
              { ”Chaincode” : “ccId”
                 “Reads”:[{"key" : “key1", "version” : “v1” }]
                 “Writes”:[{"key" : “key1", ”value" : bytes1}]
             } // end chaincode RWSet
          ] // end TxRWSet
       },  // end transaction with "Id" txUUID2

      { // another transaction },
   ] // end Transactions
}// end Block
```

Endorsing Peer (Simulation):

– Simulates transaction and generates ReadWriteSet

Committing Peer (Validation/Commit):

– Read set is utilized by MVCC validation check to ensure values read during simulation have not changed (ensures serializable isolation).

– Write set is applied to ledger (blockchain and state database) for validated transactions

Chaincode Data Patterns

Single key operations

GetState()/PutState() - Read and Write a single key/value.

Key range queries

Can be used in chaincode transaction logic (e.g. identify keys to update).

Fabric guarantees result set is stable between endorsement time and commit time, ensuring the integrity of the transaction.

GetStateByRange()

- Read keys between a startKey and endKey.

GetStateByPartialCompositeKey()

- Read keys that start with a common prefix. For example, for a chaincode key that is composed of K1-K2-K3 (composite key), ability to query on K1 or K1-K2 (performs range query under the covers). Replacement for v0.6 GetRows() table api.

Non-Key queries on data content **beta in v1**

Available when using a state database that supports content query (e.g. CouchDB)

Read-only queries against current state, not appropriate for use in chaincode transaction logic, unless application can guarantee result set is stable between endorsement time and commit time.

GetQueryResult()

– Pass a query string in the syntax of the state database

Pluggable state database - Objectives

Rich Query API for Blockchain (non-key query on data content)

- Leverage state-of-the-art database engines to extend query capabilities against blockchain data.
- Ensure interface supports plugging in different state database, for example by a vendor building on top of fabric
- To the degree possible, embed database and maintain within fabric, rather than requiring DBA skills

State Database options - Queryability

- In a **key/value database** such as **RocksDB** or **LevelDB**, the content is a blob and only queryable by key
  - Does not meet chaincode, auditing, reporting requirements for many use cases

- In a **document database** such as **CouchDB**, the content is JSON and fully queryable
  - Meets a large percentage of chaincode, auditing, and simple reporting requirements
  - For deeper reporting and analytics, replicate to an analytics engine such as Spark
  - Id/document data model compatible with existing chaincode key/value programming model, therefore no application changes are required when modeling chaincode data as JSON

- **SQL data stores** also possible, but requires more complicated relational transformation layer, as well as schema management.

State Database options - Queryability

- In a **key/value database** such as **RocksDB** or **LevelDB**, the content is a blob and only queryable by key
  - Does not meet chaincode, auditing, reporting requirements for many use cases

Default in v1

- In a **document database** such as **CouchDB**, the content is JSON and fully queryable
  - Meets a large percentage of chaincode, auditing, and simple reporting requirements
  - For deeper reporting and analytics, replicate to an analytics engine such as Spark
  - Id/document data model compatible with existing chaincode key/value programming model, therefore no application changes are required when modeling chaincode data as JSON

Beta in v1

- **SQL data stores** also possible, but requires more complicated relational transformation layer, as well as schema management.

Future

CouchDB State Database Details

- Single CouchDB instance under each peer
  - Redundancy provided by peer 'replicas' rather than database replicas
    - Normal blockchain model – peer with dedicated local data store
  - Since no database replicas, writes to database are guaranteed consistent and durable (not 'eventually consistent', as would be the case if there were database replicas)
  - Same basic model as when using local LevelDB state database
    - Except that CouchDB runs in separate local process, rather than embedded in peer process
- Configure for only local peer connections to CouchDB
  - Disable remote connections
  - Committing peer is only process that writes to state database
  - All queries go through peer authentication and authorization
  - Again, same interaction patterns as when using local LevelDB state database, but with more powerful query capability
  - CouchDB will be made available as Docker image. Docker compose will configure communication between peer container and couchdb containers (*coming Feb 2017*)

Pluggable State Database - The Challenges

How to support v1 endorsement/simulation model, when most databases do not support simulation result sets?

- That is, how to make uncommitted updates against an arbitrary database and determine the ReadWriteSet that is required for endorsement and commit validation? Not possible with most databases…

Solution:

- Query database for key values during endorser simulation, using database's rich query language
- Perform simulation updates in private workspace (peer memory) using normal chaincode APIs, e.g. PutState()
- Get ReadWriteSet from endorser simulation (Reads come from DB queries, Writes come from simulation in private workspace)
- Endorsement, Consensus, Validation use transaction ReadWriteSet Simulation Results as normal
- Apply Writes to database during Commit phase
- Peer maintains data integrity across blockchain file storage and state database

Transaction simulation is a proposal only. Updates are not yet applied to database. Implications:

- Transaction simulation does not support Read Your own Writes. GetState() always retrieves from state database.
- At commit time, validation is required to ensure conditions at simulation time (ReadSet) are still valid.
  - For key-based queries (GetState), validation step does a simple MVCC check on the ReadSet versions
  - For key-based range queries (GetStateByRange, GetStateByPartialCompositeKey) validation step re-queries to ensure result set is same – e.g. no added (phantom) items since simulation time
  - Non-key queries – Read-only queries against current state, not appropriate for use in chaincode transaction logic, unless application can guarantee result set is stable between endorsement time and commit time.

GetQueryResult() API

- Enabled when using a queryable state database such as CouchDB
  - Accepts a query string that gets passed to CouchDB state database /_find API, returns a set of key/values (documents)
  - Utilizes CouchDB /_find API query syntax

- Query API available in application chaincode
  - Read-only queries - *e.g. find all assets owned by Alice*
    - Client SDK invokes chaincode on an endorsing peer
    - Endorser returns response with ReadSet
    - Client does NOT submit transaction to ordering service (technically they could submit to ordering service to log the read, but this is not typical)
  - Queries as part of write transactions - *e.g. find all assets owned by Alice and transfer them to Bob*
    - Client SDK invokes chaincode on N endorsing peers
    - Endorser(s) return response with ReadWriteSet
    - Client SDK submits endorsed transaction to ordering service
    - Application must guarantee that result set will be stable between transaction simulation and commit time. If application cannot guarantee, structure data to use range queries or partial composite key queries instead
  - Access control enforced at either application or chaincode level

GetQueryResult() API- Indexes

- Indexes can be created in CouchDB to accelerate queries
- Initially, create indexes by calling CouchDB APIs from peer machine
- Evaluating options to create indexes upon chaincode deployment or via peer APIs

Query System Chaincode (QSCC)

- New system chaincode deployed by default in v1 to query blockchain
- Client can invoke against any peer, using same endorser request/response model that is used for application chaincode calls
- QSCC includes the following APIs (chaincode functions):
  - GetChainInfo
  - GetBlockByNumber
  - GetBlockByHash
  - GetTransactionByID

GetHistoryForKey

- GetHistoryForKey() API uses LevelDB history index to return history of values (states) for a key.
- Used for simple lineage/provenance scenarios.
- Available in application chaincode (similar to GetQueryResult)

Ledger Source Files

- core/ledger/ledger_interface.go
- core/ledger/kvledger/kv_ledger.go
- core/ledger/kvledger/txmgmt/statedb/statedb.go
- core/ledger/kvledger/txmgmt/statedb/statecouchdb/statecouchdb.go
- core/ledger/kvledger/history/historydb/historydb.go
- core/ledger/kvledger/history/historydb/historyleveldb/historyleveldb_query_executer.go

ledger_interface.go 1

```go
// QueryExecutor executes the queries
// Get* methods are for supporting KV-based data model. ExecuteQuery method is for supporting a rich datamodel and query support
//
// ExecuteQuery method in the case of a rich data model is expected to support queries on
// latest state, historical state and on the intersection of state and transactions
type QueryExecutor interface {
    // GetState gets the value for given namespace and key. For a chaincode, the namespace corresponds to the chaincodeId
    GetState(namespace string, key string) ([]byte, error)
    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([][]byte, error)
    // GetStateRangeScanIterator returns an iterator that contains all the key-values between given key ranges.
    // startKey is included in the results and endKey is excluded. An empty startKey refers to the first available key
    // and an empty endKey refers to the last available key. For scanning all the keys, both the startKey and the endKey
    // can be supplied as empty strings. However, a full scan should be used judiciously for performance reasons.
    // The returned ResultsIterator contains results of type *KV which is defined in protos/ledger/queryresult.
    GetStateRangeScanIterator(namespace string, startKey string, endKey string) (commonledger.ResultsIterator, error)
    // ExecuteQuery executes the given query and returns an iterator that contains results of type specific to the underlying data store.
    // Only used for state databases that support query
    // For a chaincode, the namespace corresponds to the chaincodeId
    // The returned ResultsIterator contains results of type *KV which is defined in protos/ledger/queryresult.
    ExecuteQuery(namespace, query string) (commonledger.ResultsIterator, error)
    // GetPrivateData gets the value of a private data item identified by a tuple <namespace, collection, key>
    GetPrivateData(namespace, collection, key string) ([]byte, error)
    // GetPrivateDataMultipleKeys gets the values for the multiple private data items in a single call
    GetPrivateDataMultipleKeys(namespace, collection string, keys []string) ([][]byte, error)
    // GetPrivateDataRangeScanIterator returns an iterator that contains all the key-values between given key ranges.
    // startKey is included in the results and endKey is excluded. An empty startKey refers to the first available key
    // and an empty endKey refers to the last available key. For scanning all the keys, both the startKey and the endKey
    // can be supplied as empty strings. However, a full scan shuold be used judiciously for performance reasons.
    // The returned ResultsIterator contains results of type *KV which is defined in protos/ledger/queryresult.
    GetPrivateDataRangeScanIterator(namespace, collection, startKey, endKey string) (commonledger.ResultsIterator, error)
    // ExecuteQuery executes the given query and returns an iterator that contains results of type specific to the underlying data store.
    // Only used for state databases that support query
    // For a chaincode, the namespace corresponds to the chaincodeId
    // The returned ResultsIterator contains results of type *KV which is defined in protos/ledger/queryresult.
    ExecuteQueryOnPrivateData(namespace, collection, query string) (commonledger.ResultsIterator, error)
    // Done releases resources occupied by the QueryExecutor
    Done()
}
```

ledger_interface.go 2

```
133     // HistoryQueryExecutor executes the history queries
134     type HistoryQueryExecutor interface {
135         // GetHistoryForKey retrieves the history of values for a key.
136         // The returned ResultsIterator contains results of type *KeyModification which is defined in protos/ledger/queryresult.
137         GetHistoryForKey(namespace string, key string) (commonledger.ResultsIterator, error)
138     }
139
140     // TxSimulator simulates a transaction on a consistent snapshot of the 'as recent state as possible'
141     // Set* methods are for supporting KV-based data model. ExecuteUpdate method is for supporting a rich datamodel and query support
142     type TxSimulator interface {
143         QueryExecutor
144         // SetState sets the given value for the given namespace and key. For a chaincode, the namespace corresponds to the chaincodeId
145         SetState(namespace string, key string, value []byte) error
146         // DeleteState deletes the given namespace and key
147         DeleteState(namespace string, key string) error
148         // SetMultipleKeys sets the values for multiple keys in a single call
149         SetStateMultipleKeys(namespace string, kvs map[string][]byte) error
150         // ExecuteUpdate for supporting rich data model (see comments on QueryExecutor above)
151         ExecuteUpdate(query string) error
152         // SetPrivateData sets the given value to a key in the private data state represented by the tuple <namespace, collection, key>
153         SetPrivateData(namespace, collection, key string, value []byte) error
154         // SetPrivateDataMultipleKeys sets the values for multiple keys in the private data space in a single call
155         SetPrivateDataMultipleKeys(namespace, collection string, kvs map[string][]byte) error
156         // DeletePrivateData deletes the given tuple <namespace, collection, key> from private data
157         DeletePrivateData(namespace, collection, key string) error
158         // GetTxSimulationResults encapsulates the results of the transaction simulation.
159         // This should contain enough detail for
160         // - The update in the state that would be caused if the transaction is to be committed
161         // - The environment in which the transaction is executed so as to be able to decide the validity of the environment
162         //   (at a later time on a different peer) during committing the transactions
163         // Different ledger implementation (or configurations of a single implementation) may want to represent the above two pieces
164         // of information in different way in order to support different data-models or optimize the information representations.
165         // Returned type 'TxSimulationResults' contains the simulation results for both the public data and the private data.
166         // The public data simulation results are expected to be used as in V1 while the private data simulation results are expected
167         // to be used by the gossip to disseminate this to the other endorsers (in phase-2 of sidedb)
168         GetTxSimulationResults() (*TxSimulationResults, error)
169     }
```

kv_ledger.go

```
177     // NewTxSimulator returns new `ledger.TxSimulator`
178     func (l *kvLedger) NewTxSimulator(txid string) (ledger.TxSimulator, error) {
179         return l.txtmgmt.NewTxSimulator(txid)
180     }
181
182     // NewQueryExecutor gives handle to a query executor.
183     // A client can obtain more than one 'QueryExecutor's for parallel execution.
184     // Any synchronization should be performed at the implementation level if required
185     func (l *kvLedger) NewQueryExecutor() (ledger.QueryExecutor, error) {
186         return l.txtmgmt.NewQueryExecutor(util.GenerateUUID())
187     }
188
189     // NewHistoryQueryExecutor gives handle to a history query executor.
190     // A client can obtain more than one 'HistoryQueryExecutor's for parallel execution.
191     // Any synchronization should be performed at the implementation level if required
192     // Pass the ledger blockstore so that historical values can be looked up from the chain
193     func (l *kvLedger) NewHistoryQueryExecutor() (ledger.HistoryQueryExecutor, error) {
194         return l.historyDB.NewHistoryQueryExecutor(l.blockStore)
195     }
196
```

statedb.go

```go
// VersionedDB lists methods that a db is supposed to implement
type VersionedDB interface {
    // GetState gets the value for given namespace and key. For a chaincode, the namespace corresponds to the chaincodeId
    GetState(namespace string, key string) (*VersionedValue, error)
    // GetVersion gets the version for given namespace and key. For a chaincode, the namespace corresponds to the chaincodeId
    GetVersion(namespace string, key string) (*version.Height, error)
    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([]*VersionedValue, error)
    // GetStateRangeScanIterator returns an iterator that contains all the key-values between given key ranges.
    // startKey is inclusive
    // endKey is exclusive
    // The returned ResultsIterator contains results of type *VersionedKV
    GetStateRangeScanIterator(namespace string, startKey string, endKey string) (ResultsIterator, error)
    // ExecuteQuery executes the given query and returns an iterator that contains results of type *VersionedKV.
    ExecuteQuery(namespace, query string) (ResultsIterator, error)
    // ApplyUpdates applies the batch to the underlying db.
    // height is the height of the highest transaction in the Batch that
    // a state db implementation is expected to ues as a save point
    ApplyUpdates(batch *UpdateBatch, height *version.Height) error
    // GetLatestSavePoint returns the height of the highest transaction upto which
    // the state db is consistent
    GetLatestSavePoint() (*version.Height, error)
    // ValidateKey tests whether the key is supported by the db implementation.
    // For instance, leveldb supports any bytes for the key while the couchdb supports only valid utf-8 string
    ValidateKey(key string) error
    // BytesKeySuppoted returns true if the implementation (underlying db) supports the any bytes to be used as key.
    // For instance, leveldb supports any bytes for the key while the couchdb supports only valid utf-8 string
    BytesKeySuppoted() bool
    // Open opens the db
    Open() error
    // Close closes the db
    Close()
}
```

statecouchdb.go 1

```
141     // GetState implements method in VersionedDB interface
142     func (vdb *VersionedDB) GetState(namespace string, key string) (*statedb.VersionedValue, error) {
143         logger.Debugf( format: "GetState(). ns=%s, key=%s", namespace, key)
144
145         compositeKey := constructCompositeKey(namespace, key)
146
147         couchDoc, _, err := vdb.db.ReadDoc(string(compositeKey))
148         if err != nil {
149             return nil, err
150         }
151         if couchDoc == nil {
152             return nil, nil
153         }
154
155         // remove the data wrapper and return the value and version
156         returnValue, returnVersion := removeDataWrapper(couchDoc.JSONValue, couchDoc.Attachments)
157
158         return &statedb.VersionedValue{Value: returnValue, Version: returnVersion}, nil
159     }
```

```
844     func constructCompositeKey(ns string, key string) []byte {
845         compositeKey := []byte(ns)
846         compositeKey = append(compositeKey, compositeKeySep...)
847         compositeKey = append(compositeKey, []byte(key)...)
848         return compositeKey
849     }
850
851     func splitCompositeKey(compositeKey []byte) (string, string) {
852         split := bytes.SplitN(compositeKey, compositeKeySep, n: 2)
853         return string(split[0]), string(split[1])
854     }
```

statecouchdb.go 2

```
285
286     // ExecuteQuery implements method in VersionedDB interface
287     func (vdb *VersionedDB) ExecuteQuery(namespace, query string) (statedb.ResultsIterator, error) {
288
289         // Get the querylimit from core.yaml
290         queryLimit := ledgerconfig.GetQueryLimit()
291
292         queryString, err := ApplyQueryWrapper(namespace, query, queryLimit,  querySkip: 0)
293         if err != nil {
294             logger.Debugf( format: "Error calling ApplyQueryWrapper(): %s\n", err.Error())
295             return nil, err
296         }
297
298         queryResult, err := vdb.db.QueryDocuments(queryString)
299         if err != nil {
300             logger.Debugf( format: "Error calling QueryDocuments(): %s\n", err.Error())
301             return nil, err
302         }
303         logger.Debugf( format: "Exiting ExecuteQuery")
304         return newQueryScanner(*queryResult), nil
305     }
```

query_wrapper.go#ApplyQueryWrapper 1

```
/*
ApplyQueryWrapper parses the query string passed to CouchDB
the wrapper prepends the wrapper "data." to all fields specified in the query
All fields in the selector must have "data." prepended to the field names
Fields listed in fields key will have "data." prepended
Fields in the sort key will have "data." prepended

- The query will be scoped to the chaincodeid

- limit be added to the query and is based on config
- skip is defaulted to 0 and is currently not used, this is for future paging implementation

In the example a contextID of "marble" is assumed.

Example:

Source Query:
{"selector":{"owner": {"$eq": "tom"}},
"fields": ["owner", "asset_name", "color", "size"],
"sort": ["size", "color"]}

Result Wrapped Query:
{"selector":{"$and":[{"chaincodeid":"marble"},{"data.owner":{"$eq":"tom"}}]},
"fields": ["data.owner","data.asset_name","data.color","data.size","_id","version"],
"sort":["data.size","data.color"],"limit":10,"skip":0}

*/
```

query_wrapper.go#ApplyQueryWrapper 2

```go
func ApplyQueryWrapper(namespace, queryString string, queryLimit, querySkip int) (string, error) {

    //create a generic map for the query json
    jsonQueryMap := make(map[string]interface{})

    //unmarshal the selected json into the generic map
    decoder := json.NewDecoder(bytes.NewBuffer([]byte(queryString)))
    decoder.UseNumber()
    err := decoder.Decode(&jsonQueryMap)
    if err != nil {
        return "", err
    }

    //traverse through the json query and wrap any field names
    processAndWrapQuery(jsonQueryMap)

    //if "fields" are specified in the query, then add the "_id", "version" and "chaincodeid" fields
    if jsonValue, ok := jsonQueryMap[jsonQueryFields]; ok {
        //check to see if this is an interface map
        if reflect.TypeOf(jsonValue).String() == "[]interface {}" {

            //Add the "_id", "version" and "chaincodeid" fields,  these are needed by default
            jsonQueryMap[jsonQueryFields] = append(jsonValue.([]interface{}),
                elems: "_id", "version", "chaincodeid")
        }
    }

    //Check to see if the "selector" is specified in the query
    if jsonValue, ok := jsonQueryMap[jsonQuerySelector]; ok {
        //if the "selector" is found, then add the "$and" clause and the namespace filter
        setNamespaceInSelector(namespace, jsonValue, jsonQueryMap)
    } else {
        //if the "selector" is not found, then add a default namespace filter
        setDefaultNamespaceInSelector(namespace, jsonQueryMap)
    }

    //Add limit
    jsonQueryMap[jsonQueryLimit] = queryLimit

    //Add skip
    jsonQueryMap[jsonQuerySkip] = querySkip

    //Marshal the updated json query
    editedQuery, _ := json.Marshal(jsonQueryMap)

    logger.Debugf( format: "Rewritten query with data wrapper: %s", editedQuery)

    return string(editedQuery), nil

}
```

statecouchdb.go#ApplyUpdates 1

```
307  // ApplyUpdates implements method in VersionedDB interface
308  func (vdb *VersionedDB) ApplyUpdates(batch *statedb.UpdateBatch, height *version.Height) error {
309
310      // STEP 1: GATHER DOCUMENT REVISION NUMBERS REQUIRED FOR THE COUCHDB BULK UPDATE
311      //         ALSO BUILD PROCESS BATCHES OF UPDATE DOCUMENTS BASED ON THE MAX BATCH SIZE
312
313      // initialize a missing key list
314      var missingKeys []*statedb.CompositeKey
315
316      //initialize a processBatch for updating bulk documents
317      processBatch := statedb.NewUpdateBatch()
318
319      //get the max size of the batch from core.yaml
320      maxBatchSize := ledgerconfig.GetMaxBatchUpdateSize()
321
322      //initialize a counter to track the batch size
323      batchSizeCounter := 0
324
325      // Iterate through the batch passed in and create process batches
326      // using the max batch size
327      namespaces := batch.GetUpdatedNamespaces()
328      for _, ns := range namespaces {
329          nsUpdates := batch.GetUpdates(ns)
330          for k, vv := range nsUpdates {
331
332              //increment the batch size counter
333              batchSizeCounter++
334
335              compositeKey := statedb.CompositeKey{Namespace: ns, Key: k}
336
337              // Revision numbers are needed for couchdb updates.
338              // vdb.committedDataCache.revisionNumbers is a cache of revision numbers based on ID
339              // Document IDs and revision numbers will already be in the cache for read/writes,
340              // but will be missing in the case of blind writes.
341              // If the key is missing in the cache, then add the key to missingKeys
342              _, keyFound := vdb.committedDataCache.revisionNumbers[compositeKey]
343
344              if !keyFound {
345                  // Add the key to the missing key list
346                  // As there can be no duplicates in UpdateBatch, no need check for duplicates.
347                  missingKeys = append(missingKeys, &compositeKey)
348              }
349
350              //add the record to the process batch
351              if vv.Value == nil {
352                  processBatch.Delete(ns, k, vv.Version)
353              } else {
354                  processBatch.Put(ns, k, vv.Value, vv.Version)
355              }
356
```

statecouchdb.go#ApplyUpdates 2

```
355             }
356
357             //Check to see if the process batch exceeds the max batch size
358             if batchSizeCounter >= maxBatchSize {
359
360                 //STEP 2:  PROCESS EACH BATCH OF UPDATE DOCUMENTS
361
362                 err := vdb.processUpdateBatch(processBatch, missingKeys)
363                 if err != nil {
364                     return err
365                 }
366
367                 //reset the batch size counter
368                 batchSizeCounter = 0
369
370                 //create a new process batch
371                 processBatch = statedb.NewUpdateBatch()
372
373                 // reset the missing key list
374                 missingKeys = []*statedb.CompositeKey{}
375
376             }
377
378         }
379     }
380
381     //STEP 3:  PROCESS ANY REMAINING DOCUMENTS
382     err := vdb.processUpdateBatch(processBatch, missingKeys)
383     if err != nil {
384         return err
385     }
386
387     // STEP 4: IF THERE WAS SUCCESS UPDATING COUCHDB, THEN RECORD A SAVEPOINT FOR THIS BLOCK HEIGHT
388
389     // Record a savepoint at a given height
390     err = vdb.recordSavepoint(height)
391     if err != nil {
392         logger.Errorf( format: "Error during recordSavepoint: %s\n", err.Error())
393         return err
394     }
395
396     return nil
397 }
```

statecouchdb.go#processUpdateBatch 1

```
399  //ProcessUpdateBatch updates a batch
400  func (vdb *VersionedDB) processUpdateBatch(updateBatch *statedb.UpdateBatch, missingKeys []*statedb.CompositeKey) error {
401
402      // An array of missing keys is passed in to the batch processing
403      // A bulk read will then add the missing revisions to the cache
404      if len(missingKeys) > 0 {
405
406          if logger.IsEnabledFor(logging.DEBUG) {
407              logger.Debugf( format: "Retrieving keys with unknown revision numbers, keys= %s", printCompositeKeys(missingKeys))
408          }
409
410          err := vdb.LoadCommittedVersions(missingKeys)
411          if err != nil {
412              return err
413          }
414      }
415
416      // STEP 1: CREATE COUCHDB DOCS FROM UPDATE SET THEN DO A BULK UPDATE IN COUCHDB
417
418      // Use the batchUpdateMap for tracking couchdb updates by ID
419      // this will be used in case there are retries required
420      batchUpdateMap := make(map[string]*BatchableDocument)
421
422      namespaces := updateBatch.GetUpdatedNamespaces()
423      for _, ns := range namespaces {
424          nsUpdates := updateBatch.GetUpdates(ns)
425          for k, vv := range nsUpdates {
426              compositeKey := constructCompositeKey(ns, k)
427
428              // Create a document structure
429              couchDoc := &couchdb.CouchDoc{}
430
431              // retrieve the couchdb revision from the cache
432              // Documents that do not exist in couchdb will not have revision numbers and will
433              // exist in the cache with a revision value of nil
434              revision := vdb.committedDataCache.revisionNumbers[statedb.CompositeKey{Namespace: ns, Key: k}]
435
436              var isDelete bool // initialized to false
437              if vv.Value == nil {
438                  isDelete = true
439              }
440
441              logger.Debugf( format: "Channel [%s]: key(string)=[%s] key(bytes)=[%#v], prior revision=[%s], isDelete=[%t]",
442                  vdb.dbName, string(compositeKey), compositeKey, revision, isDelete)
443
444              if isDelete {
445                  // this is a deleted record.  Set the _deleted property to true
446                  couchDoc.JSONValue = createCouchdbDocJSON(string(compositeKey), revision,  value: nil, ns, vv.Version,  deleted: true)
447
448              } else {
```

statecouchdb.go#processUpdateBatch 2

```
448             } else {
449
450                 if couchdb.IsJSON(string(vv.Value)) {
451                     // Handle as json
452                     couchDoc.JSONValue = createCouchdbDocJSON(string(compositeKey), revision, vv.Value, ns, vv.Version,  deleted: false)
453
454                 } else { // if value is not json, handle as a couchdb attachment
455
456                     attachment := &couchdb.AttachmentInfo{}
457                     attachment.AttachmentBytes = vv.Value
458                     attachment.ContentType = "application/octet-stream"
459                     attachment.Name = binaryWrapper
460                     attachments := append([]*couchdb.AttachmentInfo{}, attachment)
461
462                     couchDoc.Attachments = attachments
463                     couchDoc.JSONValue = createCouchdbDocJSON(string(compositeKey), revision,  value: nil, ns, vv.Version,  deleted: false)
464
465                 }
466             }
467
468             // Add the current docment, revision and delete flag to the update map
469             batchUpdateMap[string(compositeKey)] = &BatchableDocument{CouchDoc: *couchDoc, Deleted: isDelete}
470
471         }
472     }
473
474     if len(batchUpdateMap) > 0 {
475
476         //Add the documents to the batch update array
477         batchUpdateDocs := []*couchdb.CouchDoc{}
478         for _, updateDocument := range batchUpdateMap {
479             batchUpdateDocument := updateDocument
480             batchUpdateDocs = append(batchUpdateDocs, &batchUpdateDocument.CouchDoc)
481         }
482
483         // Do the bulk update into couchdb
484         // Note that this will do retries if the entire bulk update fails or times out
485         batchUpdateResp, err := vdb.db.BatchUpdateDocuments(batchUpdateDocs)
486         if err != nil {
487             return err
488         }
489
490         // STEP 2: IF INDIVIDUAL DOCUMENTS IN THE BULK UPDATE DID NOT SUCCEED, TRY THEM INDIVIDUALLY
491
492         // iterate through the response from CouchDB by document
493         for _, respDoc := range batchUpdateResp {
494
495             // If the document returned an error, retry the individual document
496             if respDoc.Ok != true {
497
498                 batchUpdateDocument := batchUpdateMap[respDoc.ID]
499
500                 var err error
```

statecouchdb.go#processUpdateBatch 3

```
500                 var err error
501
502                 //Remove the "_rev" from the JSON before saving
503                 //this will allow the CouchDB retry logic to retry revisions without encountering
504                 //a mismatch between the "If-Match" and the "_rev" tag in the JSON
505                 if batchUpdateDocument.CouchDoc.JSONValue != nil {
506                     err = removeJSONRevision( jsonValue: &batchUpdateDocument.CouchDoc.JSONValue)
507                     if err != nil {
508                         return err
509                     }
510                 }
511
512                 // Check to see if the document was added to the batch as a delete type document
513                 if batchUpdateDocument.Deleted {
514
515                     //Log the warning message that a retry is being attempted for batch delete issue
516                     logger.Warningf( format: "CouchDB batch document delete encountered an problem. Retrying delete for document ID:%s", respDoc.ID)
517
518                     // If this is a deleted document, then retry the delete
519                     // If the delete fails due to a document not being found (404 error),
520                     // the document has already been deleted and the DeleteDoc will not return an error
521                     err = vdb.db.DeleteDoc(respDoc.ID,  rev: "")
522                 } else {
523
524                     //Log the warning message that a retry is being attempted for batch update issue
525                     logger.Warningf( format: "CouchDB batch document update encountered an problem. Retrying update for document ID:%s", respDoc.ID)
526
527                     // Save the individual document to couchdb
528                     // Note that this will do retries as needed
529                     _, err = vdb.db.SaveDoc(respDoc.ID,  rev: "", &batchUpdateDocument.CouchDoc)
530                 }
531
532                 // If the single document update or delete returns an error, then throw the error
533                 if err != nil {
534
535                     errorString := fmt.Sprintf( format: "Error occurred while saving document ID = %v  Error: %s  Reason: %s\n",
536                         respDoc.ID, respDoc.Error, respDoc.Reason)
537
538                     logger.Errorf(errorString)
539                     return fmt.Errorf(errorString)
540
541                 }
542             }
543         }
544
545     }
546
547     return nil
548 }
```

historydb.go

```
26  // HistoryDBProvider provides an instance of a history DB
27  type HistoryDBProvider interface {
28      // GetDBHandle returns a handle to a HistoryDB
29      GetDBHandle(id string) (HistoryDB, error)
30      // Close closes all the HistoryDB instances and releases any resources held by HistoryDBProvider
31      Close()
32  }
33
34  // HistoryDB - an interface that a history database should implement
35  type HistoryDB interface {
36      NewHistoryQueryExecutor(blockStore blkstorage.BlockStore) (ledger.HistoryQueryExecutor, error)
37      Commit(block *common.Block) error
38      GetLastSavepoint() (*version.Height, error)
39      ShouldRecover(lastAvailableBlock uint64) (bool, uint64, error)
40      CommitLostBlock(blockAndPvtdata *ledger.BlockAndPvtData) error
41  }
42
```

historyleveldb_query_executer.go 1

```go
// GetHistoryForKey implements method in interface `ledger.HistoryQueryExecutor`
func (q *LevelHistoryDBQueryExecutor) GetHistoryForKey(namespace string, key string) (commonledger.ResultsIterator, error) {

    if ledgerconfig.IsHistoryDBEnabled() == false {
        return nil, errors.New( text: "History tracking not enabled - historyDatabase is false")
    }

    var compositeStartKey []byte
    var compositeEndKey []byte
    compositeStartKey = historydb.ConstructPartialCompositeHistoryKey(namespace, key, endkey: false)
    compositeEndKey = historydb.ConstructPartialCompositeHistoryKey(namespace, key, endkey: true)

    // range scan to find any history records starting with namespace~key
    dbItr := q.historyDB.db.GetIterator(compositeStartKey, compositeEndKey)
    return newHistoryScanner(compositeStartKey, namespace, key, dbItr, q.blockStore), nil
}

//historyScanner implements ResultsIterator for iterating through history results
type historyScanner struct {
    compositePartialKey []byte //compositePartialKey includes namespace~key
    namespace           string
    key                 string
    dbItr               iterator.Iterator
    blockStore          blkstorage.BlockStore
}

func newHistoryScanner(compositePartialKey []byte, namespace string, key string,
    dbItr iterator.Iterator, blockStore blkstorage.BlockStore) *historyScanner {
    return &historyScanner{ compositePartialKey: compositePartialKey,  namespace: namespace,  key: key,  dbItr: dbItr,  blockStore: blockStore}
}
```

historyleveldb_query_executer.go 2

historyleveldb_query_executer.go | kv_ledger.go | config.go | couchdbutil.go | historydb.go | historyleveldb.go | util.go | val

```
71  func (scanner *historyScanner) Next() (commonledger.QueryResult, error) {
72      if !scanner.dbItr.Next() {
73          return nil, nil
74      }
75      historyKey := scanner.dbItr.Key() // history key is in the form namespace~key~blocknum~trannum
76
77      // SplitCompositeKey(namespace~key~blocknum~trannum, namespace~key~) will return the blocknum~trannum in second position
78      _, blockNumTranNumBytes := historydb.SplitCompositeHistoryKey(historyKey, scanner.compositePartialKey)
79      blockNum, bytesConsumed := util.DecodeOrderPreservingVarUint64(blockNumTranNumBytes[0:])
80      tranNum, _ := util.DecodeOrderPreservingVarUint64(blockNumTranNumBytes[bytesConsumed:])
81      logger.Debugf( format: "Found history record for namespace:%s key:%s at blockNumTranNum %v:%v\n",
82          scanner.namespace, scanner.key, blockNum, tranNum)
83
84      // Get the transaction from block storage that is associated with this history record
85      tranEnvelope, err := scanner.blockStore.RetrieveTxByBlockNumTranNum(blockNum, tranNum)
86      if err != nil {
87          return nil, err
88      }
89
90      // Get the txid, key write value, timestamp, and delete indicator associated with this transaction
91      queryResult, err := getKeyModificationFromTran(tranEnvelope, scanner.namespace, scanner.key)
92      if err != nil {
93          return nil, err
94      }
95      logger.Debugf( format: "Found historic key value for namespace:%s key:%s from transaction %s\n",
96          scanner.namespace, scanner.key, queryResult.(*queryresult.KeyModification).TxId)
97      return queryResult, nil
98  }
99
100 func (scanner *historyScanner) Close() {
101     scanner.dbItr.Release()
102 }
```

historyleveldb_query_executer.go 3

historyleveldb_query_executer.go | kv_ledger.go | config.go | couchdbutil.go | historydb.go | historyleveldb.go | util.go | validator

```
104  // getTxIDandKeyWriteValueFromTran inspects a transaction for writes to a given key
105  func getKeyModificationFromTran(tranEnvelope *common.Envelope, namespace string, key string) (commonledger.QueryResult, error) {
106      logger.Debugf( format: "Entering getKeyModificationFromTran()\n", namespace, key)
107
108      // extract action from the envelope
109      payload, err := putils.GetPayload(tranEnvelope)
110      if err != nil {
111          return nil, err
112      }
113
114      tx, err := putils.GetTransaction(payload.Data)
115      if err != nil {
116          return nil, err
117      }
118
119      _, respPayload, err := putils.GetPayloads(tx.Actions[0])
120      if err != nil {
121          return nil, err
122      }
123
124      chdr, err := putils.UnmarshalChannelHeader(payload.Header.ChannelHeader)
125      if err != nil {
126          return nil, err
127      }
128
129      txID := chdr.TxId
130      timestamp := chdr.Timestamp
131
132      txRWSet := &rwsetutil.TxRwSet{}
```

historyleveldb_query_executer.go 4

```
133
134     // Get the Result from the Action and then Unmarshal
135     // it into a TxReadWriteSet using custom unmarshalling
136     if err = txRWSet.FromProtoBytes(respPayload.Results); err != nil {
137         return nil, err
138     }
139
140     // look for the namespace and key by looping through the transaction's ReadWriteSets
141     for _, nsRWSet := range txRWSet.NsRwSets {
142         if nsRWSet.NameSpace == namespace {
143             // got the correct namespace, now find the key write
144             for _, kvWrite := range nsRWSet.KvRwSet.Writes {
145                 if kvWrite.Key == key {
146                     return &queryresult.KeyModification{TxId: txID, Value: kvWrite.Value,
147                         Timestamp: timestamp, IsDelete: kvWrite.IsDelete}, nil
148                 }
149             } // end keys loop
150             return nil, errors.New( text: "Key not found in namespace's writeset")
151         } // end if
152     } //end namespaces loop
153     return nil, errors.New( text: "Namespace not found in transaction's ReadWriteSets")
154
155 }
156
```

Final Thoughts + Q&A

- I hope you have enjoyed today's talk and learned how a Hyperledger Peer endorses, validates, simulates and queries transactions.