UBIN PHASE 2

Technical Documentation

Corda

This report describes the technical design and observations of the Corda prototype in Ubin Phase 2.

Table of Contents

1 Platform and System Design

Corda is a distributed ledger technology platform suitable to be used by regulated financial institutions. It's inspired by blockchain systems and is known for recording, processing and synchronising financial agreements between know and identified parties. Corda uses UTXO (unspent transaction output) model, where new data records, representing asset units or deals, need to reference previous versions of that data by transaction hash, creating an immutable chain of provenance and lineage. When a data record is involved in a transaction, the Corda node will 'soft lock' that record so that it cannot be used in another transaction at the same time. A separate Notary, or consensus service, preserves ordering and uniqueness, and prevents double-spend of assets.

In Corda, update of the UTXO models is applied through transactions, which takes the current state as an input and produces a new state as an output. To be committed to the ledger, a proposed transaction must achieve both validity and uniqueness consensus.

Validity consensus can be reached if a proposed transaction defines output states that it is valid according to the contract that is tied to the respective states and has all the required signatures to which the proposed transaction needs. To verify the validity of the input states of the transaction, it is necessary to verify every transaction in the chain led up to the creation of the input states of the proposed transaction *(walking the chain).*

To prevent double spent, a valid transaction must also reach uniqueness consensus. Uniqueness consensus ensures that all the input states in the proposed transaction are not already consumed in another transaction.

Notary services in Corda provide uniqueness consensus. The transaction achieves uniqueness consensus only if the Notary has signed or notarised. The Notary would only sign the transaction if it has not signed another transaction with any of the proposed transaction input states. In doing so, the Notary finalises the transaction: until the Notary signature is obtained, a transaction would not be committed to the ledger. In Ubin Phase 2 Corda prototype, a Simple Notary is used to achieve uniqueness consensus.

To maintain privacy, each message or transaction is addressed and sent to the specific counterparty (peer to peer) in the network, the message/transaction would not be visible to other node in the network that is not involved in the transaction. To improve user privacy in the transaction, Ubin Phase 2 Corda prototype employs Confidential Identities which involves the generation of new random keys per transaction that are linked with a corresponding linkage certificate, obscuring the identity of the network participant. The certificate is only sent to the parties involved in a transaction.

A transaction involves multiple steps between the sender, receiver and the involvement of a third party such as the Notary. To facilitate peer-to-peer communication between the nodes involved in the transaction, Corda uses multi-party sub-protocols call flow. Apart from managing point to point communication between nodes, flow framework is also used to encapsulate business process.

For example, in the illustration below, transactions 1 and 2 are only visible to node A and B, transaction 3 is only visible to node B and C, transaction 4 is only visible to node C and D, and transaction 5 is only visible to node B and D.

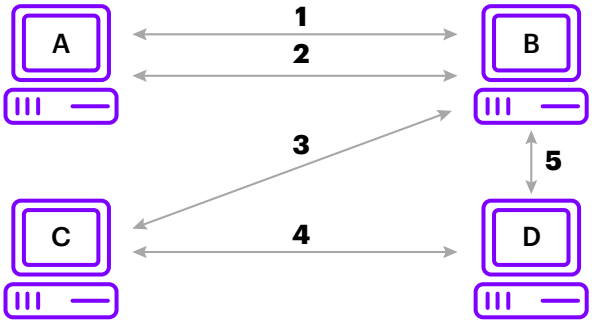


Figure 1: Illustration of five transactions between four banks in a Corda network

In this project, the prototype on Corda was developed using ***Corda version 1.0***, leveraging Kotlin as the main programming language. Some of the design concepts employed in Ubin Phase 2 are as follows:

- A representation of Cash for fund transfer whenever a sender has sufficient funds
- A representation of an Obligation from a sender to a receiver to be paid in cash in the future. This is used to represent pending or queued payment instructions when a sender has insufficient funds to settle at the point of initiating the fund transfer
- A vault for each node to hold all its Cash and Obligations states
- Both Cash and Obligation use the UTXO state model to represent the disposition of the object

1.1 Logical Architecture

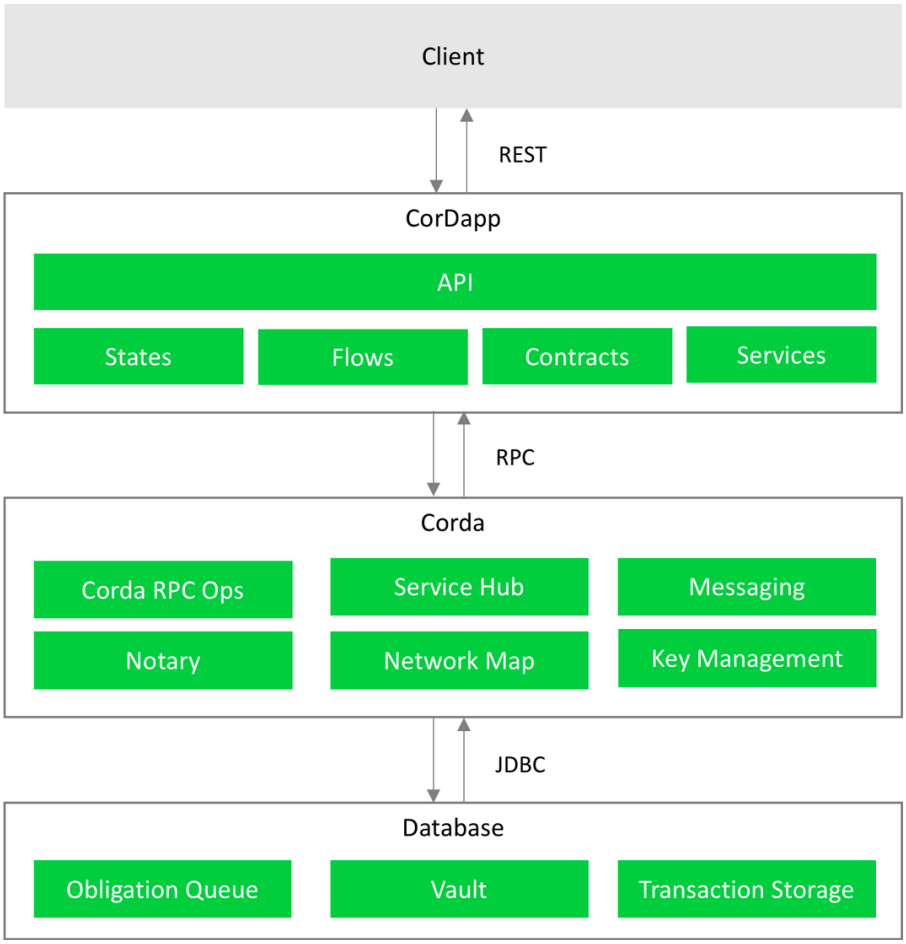


Figure 2: Logical Architecture

CorDapp is an implementation that extends the Corda default capabilities. CorDapps comprises of data structure definitions, state definitions, contract definitions, and flow definitions. In Ubin prototype on Corda, CorDapp also exposes RESTful APIs leveraging on JAX-RS framework which is consumed by the client. The client of this prototype is a bank web application where each bank will have its own application. Corda nodes communicate asynchronously using AMQP/1.0 with the other nodes in the network, this messaging between nodes is handled by the underlying Corda framework.

1.1.1 Flow

Corda automates orchestration of the transaction using flow. A flow is a sequence of steps that tells a node how to achieve a specific ledger update, such as issuing an asset or settling a trade. Nodes communicate by passing messages between flows. Each node has zero or more flow classes that are registered to respond to messages from a single flow.

Flows can be composed by starting a flow as a sub process in the context of another flow. The flow that is started as a sub process is known as a *subflow*. The parent flow will wait until the subflow returns.

Ubin Phase 2 leverages the confidential identity capability provided by Corda to shield the participants in the transaction. When preparing the transaction, flow would request to generate new public and private key which is used only in that transaction. To store the state/data in the vault, this prototype uses H2 database and each Corda node has its own instance of the database. Corda leverages on JDBC to retrieve and manipulate data in the database. Corda

provides a library of flow to handle common tasks, meaning that developers do not have to redefine the logic behind common processes such as:

- Notarizing and recording a transaction
- Gathering signatures from counterparty nodes
- Verifying a chain of transactions
- Swap identities with other party

1.1.2 State

State is an immutable object to represent data **(facts)** in the ledger. As states are immutable, it cannot be changed directly to reflect a change in the state of the world. When there is a need to manipulate the states, a transaction is proposed to create new version of the state and mark existing state as historic. There are few states is implemented/used in Ubin Phase 2 implementation as following:

1. Cash
2. Obligation
3. Redeem
4. Pledge

1.1.3 Contract

Every state contains a reference to a contract that governs the changes of the state over time. Contracts is an executable code which validates changes to state objects in transaction. Contract execution is deterministic and it validates the validity of the transaction based on the content of transaction itself. A contract is tied to a state, it takes transaction as an input and verify the states based on the rules defined to determine whether the transaction is valid.

1.1.4 Queue

When an obligation is issued, the obligation is added to the queue for the purpose of maintaining FIFO ordering and prioritisation. Since there is no concept of queuing within Corda, in Ubin Phase 2 Prorotype, queue is developed using custom table persisted in each node H2 database. Obligation is only added to the Queue of the payment instruction sender, it would not be added to the receiver queue. Obligation that is stored in the queue has additional attributes such as status and priority. Status and priority of the obligation are not shared to the counterparties that is involved in the transaction and only known to the issuer of the obligation itself hence if there are any changes to the priority and status, only issuer would know the status and priority of the obligation itself. In Ubin Phase 2 protoype, CorDapp would access the queue with SQL query through JDBC.

1.2 Physical Architecture

Ubin Phase 2 Corda prototype has 15 nodes, deployed across 15 Azure virtual machines (VMs). Each virtual machine has:

- 1 core
- 3.5 GB RAM running
- Ubuntu OS 16.04 LTS

The VMs are organised as:

- 1 VM for Network Map
- 1 VM for Notary

- 1 VM for Regulator Node
- 1 VM for Central Bank
- 11 VM for 11 Bank Nodes (1 VM per participating bank)

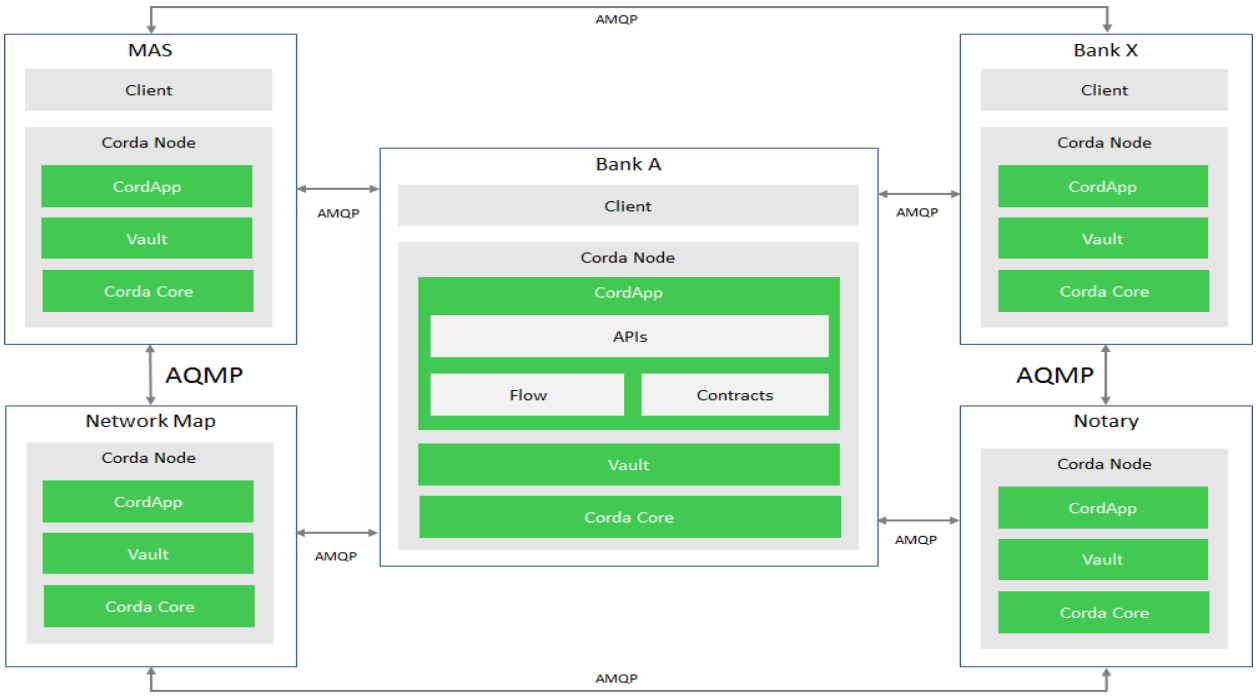


Figure 3: Physical Architecture

Corda network in Ubin Phase 2 consists of the following components:

1. Network Map
    - Network Map service keeps track of all participants on the Corda network and emits the new Network Map to the registered nodes. This is only used to publish the newest Network Map to the participants.
    - Network Map service stores a list of all the participants in the network.
    - Network Map is cached on every Corda node by default when a new node is added. If the Network Map Service is temporarily unavailable, existing nodes will still be able to communicate with each other, but new nodes cannot join the network.

2. Notary
    - Notary service signs transactions before they are committed as states to each of the Corda Node vaults privy to the transaction. Ubin Phase 2 Corda network uses Simple Notary type,
    - Notary services provides uniqueness consensus by attesting that a given transaction does not contain any states which has been consumed in another transaction to prevent double-spend (a consumed state is being used in another transaction).
    - Every state holds a reference to the uniqueness service which governs it (but it is possible to switch the uniqueness service for the state if agreed upon by all parties). The uniqueness service must sign any transaction where states governed by it are changed by the Corda Bank Nodes. A non-validating uniqueness service will store the Stateref which is made up of the hash of the transaction and its index.

This will ultimately provide the pointer to the previous transactions and states for verification. A validating uniqueness service will store the hash of the transaction graph, which will include all of the previous transactions and their contents. This is being solved with a secure enclave hardware solution to provide data leakage.

- If the notary service is not available at a point in time when transaction is in progress, then the transaction will be held in a persistent queue on the sender's side awaiting the uniqueness service signature, and the flows between parties will not be finalized. Any states governed by that uniqueness service will not be editable until the service returns to a running state.

3. Bank Nodes
    - A participant of a distributed ledger network, e.g. a bank, will run a regular node.
    - This allows them to transact with other participants, sending and receiving digital assets, or running shared business logic (contracts).
    - Nodes communicate with services and other participant nodes using AMQP/1.0 over TLS.
    - Messages that are successfully processed by a node generate a signed acknowledgement message called a receipt, which may be generated some time after the message is processed in the case where acknowledgements are being batched to amortize signing overhead, and the receipt identifies the message by the hash of its content.

      The purpose of the receipts is to give a node undeniable evidence that a counterparty received a notification that would stand up later in a dispute mediation process.

2 Functional Specifications

2.1 Fund Transfer

In the Corda design, a 'Transfer' of funds is executed through a peer-to-peer approach where only the sender and receiver banks will process, validate and record the transaction.

With Confidential Identities, sender would request a new, unique pair of public key and certificate from the receiver of the payment instruction. This anonymous identity is only known to both sender and receiver. This helps to shield the parties that are involved in payment instruction so that future owners of the asset are not able to identify the previous owners. This is important to preserve privacy in the UTXO model where the chain of custody of the asset needs to be validated all the way back to issuance of the asset by MAS. When a party in the network verifying input states of a transaction, they will be able to see the previous transactions that leading up to creation of the states in the current transaction, without shielding the participants, they would be able to see who were involved in that historical transaction.

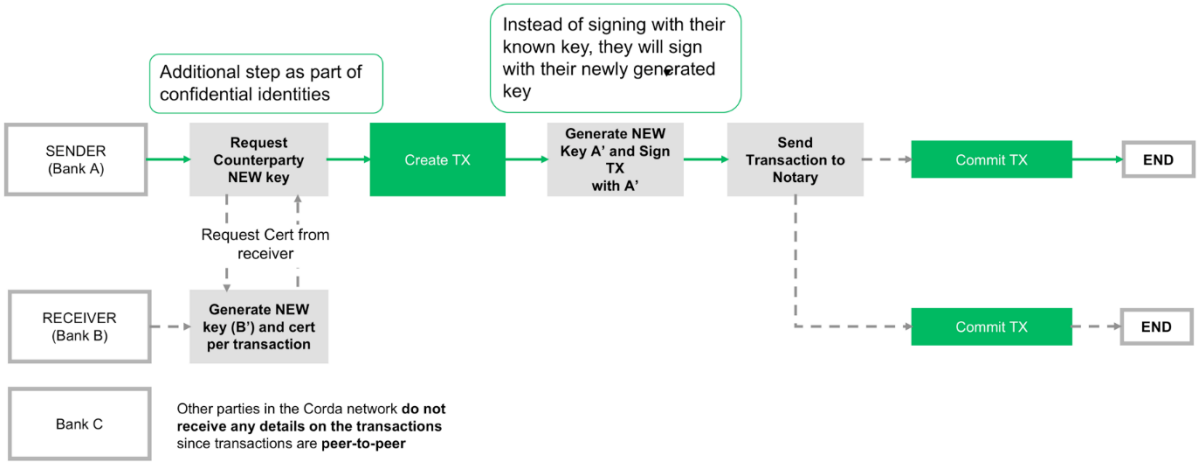


Figure 4: Transaction flow of Fund Transfer in Corda

As illustrated in diagram above, a sender initiating the transfer of funds would first request the new key from the counterparty to be used in the transaction. The public key of the counterparty is used in the transaction when generating the cash output states, setting the receiver as the new owner giving them control over signing future transactions. When signing the transaction, sender also generates a new key and signs the transaction with the new key. After the contracts verify and accept input and output states in the transaction to ensure that it is contractually valid, transaction would be sent to the respective counterparty for them to sign.

When the signing is completed, the transaction is sent to the notary to verify the uniqueness of the states and sign with its signature as well. Upon notarisation, both the sender and receiver would commit the final transaction with its output states to its respective ledgers. This approach allows the privacy of the transaction to be maintained in real time and in future transactions where the lineage of states used as input states do not reveal the identities of the past participants.

In this process, Confidential Identities assures that only the sender and receiver parties can identify the identities of the participants in the payment instruction, however the transaction amount remains visible. To identify the identities, a party must have the correspondent certificates which are stored off ledger and the certificate is not sent along with the transactions. Certificates are only shared between participants during transaction orchestration in the flow. When another party is resolving the dependency for transaction verification, it will only see anonymous party because it does not have the certificate correspondent to the key in the transaction.

Alternatively, in the 'Transfer' flow, if a sender party does not have sufficient funds for an immediate settlement of fund transfer, an Obligation state will be created in an alternative flow. Obligations will be registered into a persistent queue (refer to Section 4.2.1) maintained by the sender party which can be cancelled, reprioritised, or otherwise settled when funds are available or processed through gridlock resolution (refer to Section 4.3.1).

The Notary functions as a service that accepts transactions submitted to them for uniqueness consensus. The Notary would either return an accepted signed transaction or a rejection error that a double spend has occurred due to the any of the input states has been consumed in another transaction.

The sequence diagram below illustrates fund transfer in Corda in detail:

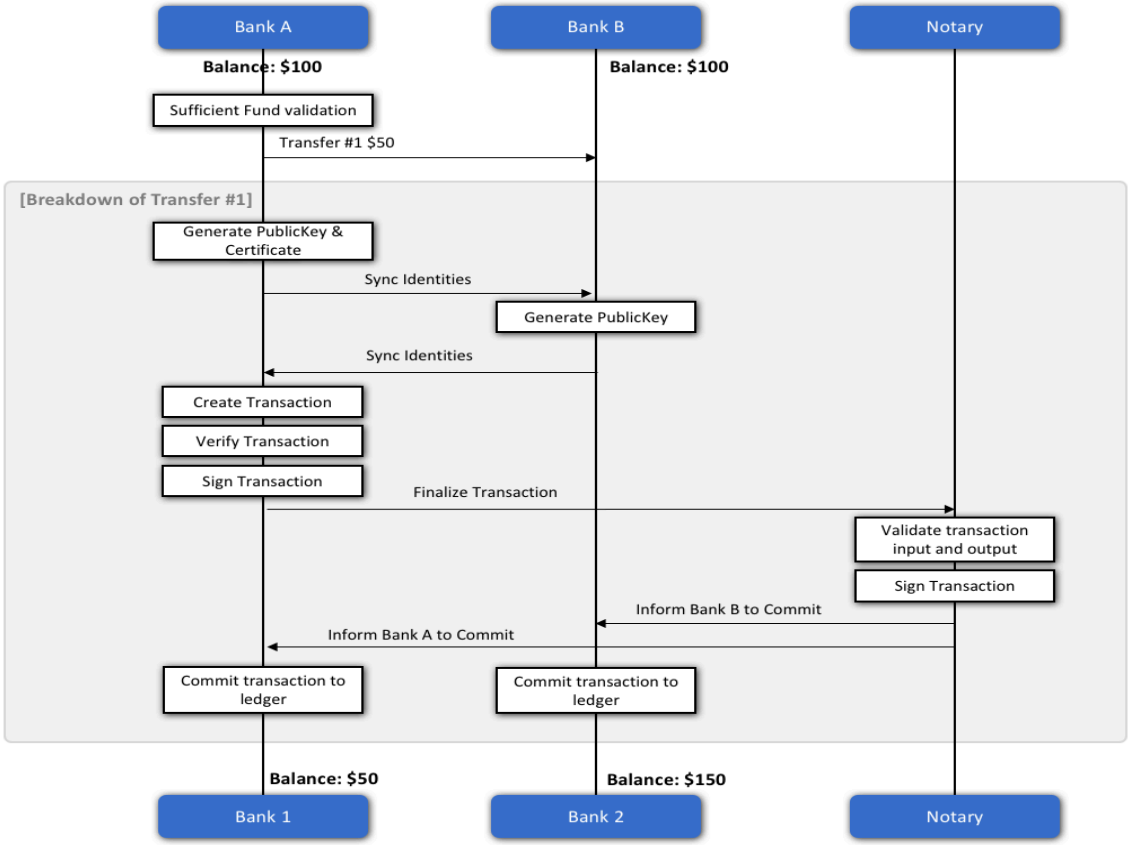


Figure 5: Sequence Diagram for Fund Transfer in Corda

In above example, Bank A issues the fund transfer of $50 to Bank B, where it has $100 in the balance hence there is sufficient liquidity to complete this transfer. The process involved in this fund transfer are as followed:

- Bank A will check if it has sufficient fund balance for a straight-through transfer
- Bank A will check if there is any queued payment with a higher priority than the current transfer instruction
- Bank A will generate fresh public key and certificate during transfer
- Bank A will synchronise the identities
- Bank B will generate fresh public key and certificate and synchronise the identities
- Bank A will prepare the transaction and sign the transaction
- Notary will provide **uniqueness consensus** for this transaction to prevent double-spending
- Upon Notary's signature, the relevant participants of the transaction are notified to commit the transaction to the ledger

2.2 Queue Mechanism

During a 'Transfer' flow, a payment instruction initiated when there is insufficient funds in the sender's balance will lead to the issuance of an Obligation state in the ledgers of both the sender and receiver. Confidential identities are used by generating a new key and certificate for the transaction. This key and certificate would then be exchanged with the counterparty. The newly generated keys would be used to identify the participants (lender and borrower) of

the Obligation state. The Obligation state would be issued as output of the transaction, with the transaction signed and sent to the counterparty. If the counterparty responds with the verified and signed transaction, the Obligation state details (i.e. linearid, status, etc.) would be put to the queue of payment instruction sender. If not, the transaction would be cancelled and the Obligation states would not be issued.

Each Obligation states represents a 'Pending' funds transfer or an outgoing unsettled payment instruction that will be settled in the future. These Obligation states details are also replicated and maintained by the sender in its persistent priority-queue to ensure that the node will be able to recover the information in the event of any node restart. Each of the Obligation states in the queue is tagged with a priority level that can only be changed by the sender. The priority level is visible only to the sender as the queue is local to the sender party, and therefore not shared with other nodes to ensure privacy.

Queue implementation in Corda is using a custom table persisted in H2 database in each node. When obligation is added to the sender queue, there is additional information, such as status and priority stored along with the obligation state detail in the sender's H2 database. This ensures status and priority are only visible to the sender of the payment instruction since status and priority are not part of the obligation state which propagated to the receiver ledger.

- Priority in Corda represented in number where high priority payment would have '1' as priority, while normal priority payment would have '0'.
- Status in Corda queue represented in number with following possible values:

Table 1: Status in Corda queue

Status	Value
0	ACTIVE
1	HOLD
2	CANCELLED
3	SETTLED

2.2.1 Queue Settlement

The queue maintains a FIFO (first-in-first-out) order of sequence. Each virtual machine has a scheduled processing function that periodically triggers a settlement API to attempt to settle obligation in the queue if possible based on the current funds available. The queue settlement logic will iterate through the queue of obligation to settle in sequence by the next active, highest priority and oldest obligation first. By changing the priority level, the order of which the obligations are settled changes according to the settlement logic. The queue also maintains the status of each of the obligation. An obligation can be put on 'hold' status to exclude them from participating in any settlement or gridlock resolution. They could be reactivated by changing their status back to 'active' in the queue, allowing them to participate in any future settlements and gridlock resolution.

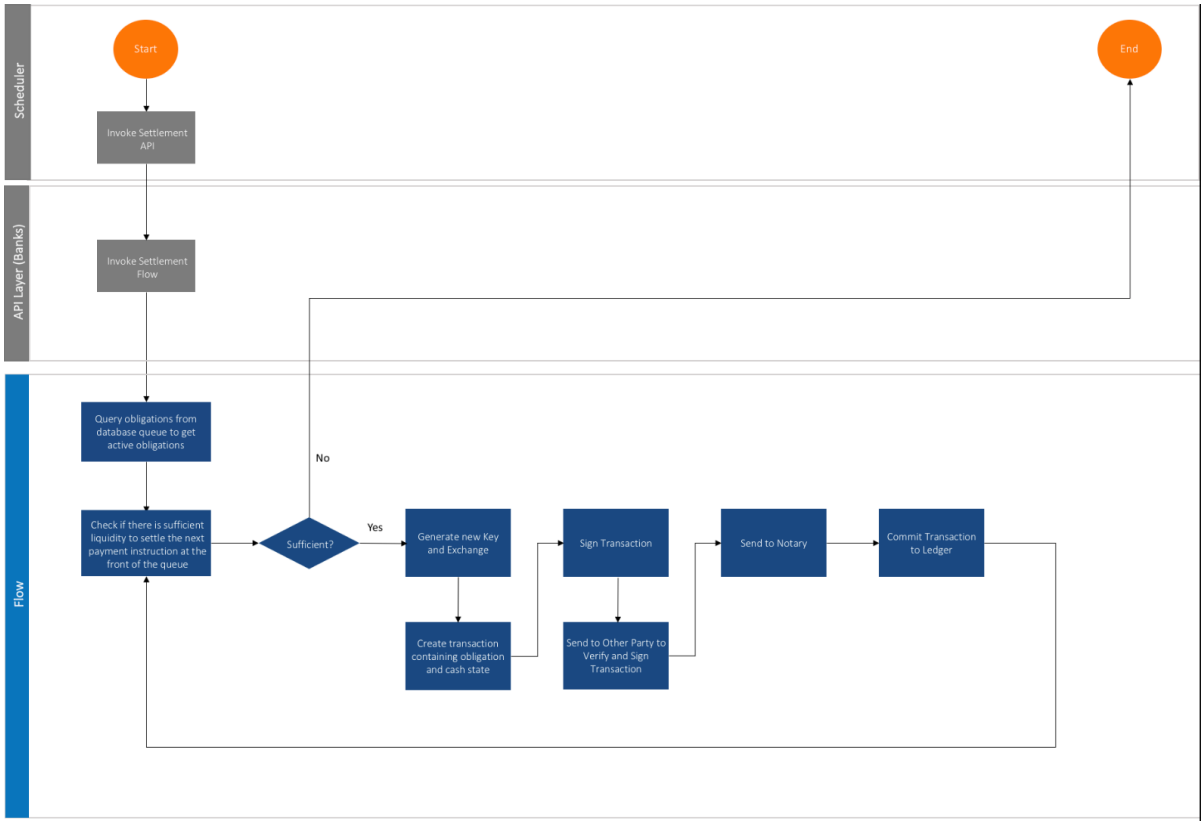


Figure 6: Queue Settlement Process Flow

2.2.2 Queue Management

As part of queue management, there are several features exposed as REST API in the CorDapp. These functions include:

- Queue cancellation
- Queue reprioritisation
- Toggle queue hold/resume

2.2.2.1 Queue Cancellation

An Obligation can only be cancelled by the issuer (sender of payment instruction) of the Obligation itself and it has to be cancelled at both sides. To cancel Obligation, sender will prepare transaction containing Obligation to be cancelled, sign it, and propose the transaction to the receiver of the Obligation. Receiver party would verify the transaction and sign it. When transaction is completed, Obligation states would be destroyed from both parties' vault and the Obligation record in queue would be updated to status '2' (Cancelled). Once Obligation is cancelled, it cannot be reactivated back.

2.2.2.2 Queue Reprioritisation

Reprioritising of Obligation in Corda would only require update of priority field in the database queue. Where there is a request to update certain Obligation, Corda flow will take updated status and update the Obligation in queue table with new priority status. Since priority is only known to the sender of the Obligation, reprioritisation in Corda does not involve the receiver of the Obligation.

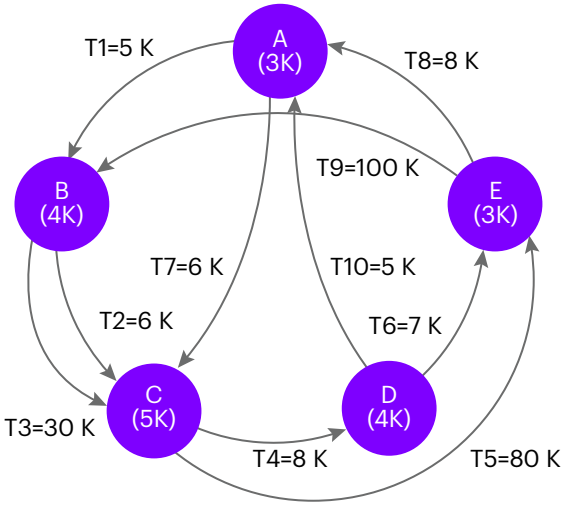
2.2.2.3 Queue Hold/Resume

Similar to Obligation reprioritisation, hold and resume of Obligation only involves the sender of the Obligation, therefore the receiver would not know if the Obligation status has changed. When Obligation is hold, respective Corda flow will update the Obligation status in the queue to '1' (HOLD), when obligation is resumed, the obligation status in the queue would be updated to '0'.

2.3 Gridlock Resolution

To illustrate the design for gridlock resolution across the 3 platforms, this section will refer to a common gridlock scenario as per described below. There are more examples of gridlock/deadlock scenarios described in Section 7 Testing which also includes the resolution per workstream.

- There are 5 participating banks (Bank A, Bank B, Bank C, Bank D and Bank E) with starting balance of $3,000, $4,000, $5,000, $4,000 and $3,000 respectively
- There is a total of 10 payment instructions (T1, T1, T3, T4, T5, T6, T7, T8, T9 and T10), each of which have the sender and receiver detailed in the table below where a negative value indicates amount to be paid while a positive value indicates amount to be received
- All payment instructions are of *Normal* priority
- The banks have insufficient liquidity to settle the first payment instruction in their outgoing queues



5 BANKS

- A Bank A
- B Bank B
- C Bank C
- D Bank D
- E Bank E

10 TRANSACTIONS

	A (K)	B (K)	C (K)	D (K)	E (K)
Starting balance	3	4	5	4	3
T1	-5	+5			
T2		-6	+6		
T3		-30	+30		
T4			-8	+8	
T5			-80		+80
T6				-7	+7
T7	-6		+6		
T8	+8				-8
T9		+100			-100
T10	+5			-5	

Figure 7: Gridlock Resolution Scenario

2.3.1 Algorithm description

The three stages of Corda are Detect, Plan and Execute. The gridlock resolution mechanism will run repeatedly to settle the queued payment instruction (obligation) in a gridlock.

Given a series of bilateral obligations, the liquidity savings mechanism (LSM) requires a 3-stage process:

1. Detect phase to discover netting opportunities (in the form of cycles) from the network to collect Obligations and the cash limit from each node it's willing to offer as part of the gridlock resolution mechanism.
2. Plan phase to calculate a viable solution from all the possible permutations of the netting opportunities discovered from the network.
3. Execute phase to implement the planned solution by constructing an atomic transaction of the net payments and to-be-settled Obligations for finalisation.

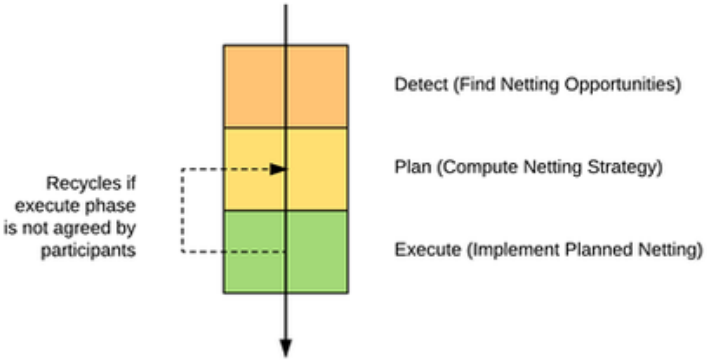


Figure 8: Gridlock Resolution Stages

2.3.1.1 Detect Phase (Finding netting opportunities)

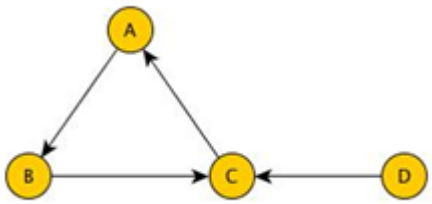


Figure 9: Detect Phase Illustration

The decentralised detect phase is based around a concept of a recursive graph scanning algorithm. The graph consists of nodes with edges in the graph representing one or more unsettled fund transfers between nodes.

At a general level, the detection algorithm runs as follows:

- Any node with Obligations either made by it, or to it, may initiate such a scan. An initiator will leverage on Corda's flows to start the communication it needs to discover the network. The initiator computes a unique scan ID which contains a hash of the current time and attach it to a data structure (SCAN-REQUEST) that it's transmitting to its neighbours.
- The initiator sends a SCAN-REQUEST request to each node with which it shares an Obligation.
- Any node sending a SCAN-REQUEST request starts a timeout timer and waits for a SCAN-ACK message to be returned to it before the timer expires. If the timeout timer expires then it presumes that the request has failed and that the recipient will not respond. Any messages received after a timeout must be ignored.
- Nodes that receive a SCAN-REQUEST request must check if the scan ID is not older than the scan ID the recipient has recorded so far. A scan ID which is older than the scan ID recorded means that a LSM is currently running and the recipient will ignore the SCAN-REQUEST.
- If the SCAN-REQUEST has a valid scan ID, the recipient must respond with a scan acknowledgement message (SCAN-ACK). The SCAN-ACK contains no information other than that the SCAN-REQUEST was received. A node that issues a SCAN-ACK subsequently is committed to issue SCAN-RESPONSE which is a data structure that consists of the obligations and the cash limit the recipient is willing to offer up in the gridlock resolution.
- At the same time, the recipient of the SCAN-REQUEST propagates the scan request to each of its counterpart nodes which it has obligations to and from.

- The SCAN-RESPONSE will indicate the final status of a request. This may indicate a set of edges found during scanning, or that another scan has already been seen with a lower scan ID within the last 20 seconds.
- If a node receives a SCAN-RESPONSE reporting a scan collision in which a lower value scan ID (treating scan IDs as a large integer) has already been seen then the scan will be stopped. If the recipient of the SCAN-RESPONSE is, itself, waiting to return data to an earlier requester then it will issue a SCAN-RESPONSE message to its requester indicating the scan collision.
- When a node has received all pending SCAN-RESPONSE messages it aggregates all the results into one larger message, removing any duplicates.
- When the final SCAN-RESPONSE message is received by the initiator then its final aggregation of results will represent all the edges in the graph, and the full graph can be trivially recomputed.

Any node may initiate a scan, but let us first consider node A starting it.

1. A will initiate the scan and request nodes B and C continue it.
2. Node B will subsequently request that node C continue the scan, while node C will request both nodes B and D continue.
3. Node D will respond with the details of the edge CD.
4. At this point it is possible that either node B, C or A may take different actions, dependent on the timing of the scan, but node C will bundle the reply from D regarding edge CD, while adding details of the edge AC. It may also add details of the edge BC. Node B may reply solely with details of the edge AB, but may also offer details of the edge BC
5. Node A will collate the replies from B and C, forming a view of a graph with edges AB, AC, BC and CD.

This can be depicted as per below:

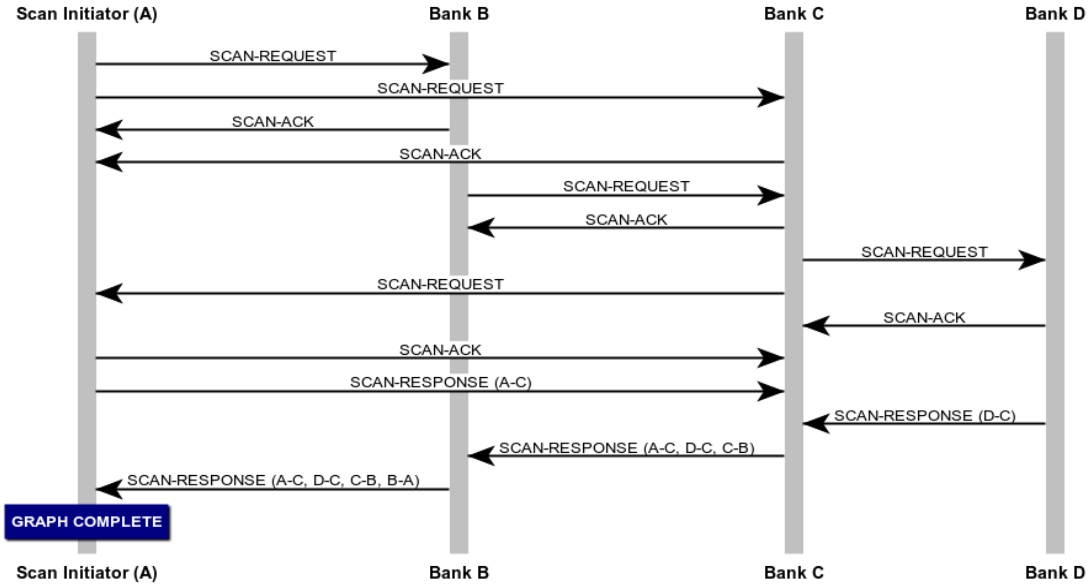


Figure 10: Detect Phase Sequence Diagram

2.3.1.2 Plan Phase

Having identified opportunities for netting we must compute the best available net payment set that we can find, as there may be many different sets, each with different implications. The choices of strategies should be able to evolve over time, as there has been little research into decentralised netting.

It is worth noting that at the point where we start to compute a strategy we may be doing so with either a real or pseudo-anonymous identify for each node. This stage must be agnostic of approach defined by stage 1.

- *Bilateral planner*

The simplest types of strategies that can be calculated only considers bilateral unsettled payments. As such, the strategy is somewhat limited because it cannot be used to compute solutions involving payment loops between three or more parties.

In this strategy, two parties who have one or more pending obligations to each other will see some number of those payments settled via a single net payment transaction.

The result of this strategy is a set of obligations that can be met via zero or one payments (zero payments will be required if the net result of the Obligations is zero).

- *Multilateral planners*

Bilateral approaches to liquidity saving strategies are very straightforward, but cannot resolve problems in which more complex cyclic payments are required to achieve netting. In these cases, we must augment bilateral approaches with multilateral ones.

There are a number of potential strategies for computing multilateral netting solutions, all of which are based on having a view of some, or ideally all, potential payments that are unsettled within the network.

The result of this strategy is a set of Obligations that can be met via zero, one or more than one payments.

Strategy

The detect phase provides details of net payments flows that are required within the system, and this allows us to find cycles of payments within the payment graph using standard algorithmic approaches. These cycles represent opportunities for liquidity saving.

In addition to being able to find cycles, we are able to assess the amount of liquidity that can be freed up via a netting operation as this is represented by the smallest single net payment required between any of the parties.

During the planning phase the node which has run detection computes the set of potential netting resolutions and attempts to have these executed. If one particular plan cannot be executed then the execute phase will return to the planning phase to attempt to execute a different netting operation.

There is no clear optimal strategy for computing a netting plan, but in general, those plans that involve liquidity saving for the largest numbers of payers will result in clearing the most obligations from the system. It is worth noting that if a potential participant does not wish to implement a specific plan as a result of insufficient liquidity, or the need to preserve liquidity for a later point in time then this approach will naturally iterate from the potentially most beneficial to less beneficial solutions, that can be fulfilled.

2.3.1.3 Execute Phase

Once a netting solution has been planned then it must be executed. Execution involves generating the necessary sets of payments and Obligation settlements and having all relevant parties agree that these are correct. The payments will be collected into a single atomic transaction such that all either succeed or fail as one operation. This ensures that there are

no failure scenarios in which a netting payment is made but where the other associated payments are not made.

It is worth noting that a netting strategy may ultimately fail. Corda LSM computations do not block the real-time settlement of Obligations and so by the time we have computed a netting approach we may find that obligations have changed. Similarly, one or more would-be participants may decide that they do not wish to adopt a particular netting strategy in order to preserve critical liquidity levels. Should this occur then the resolve stage will fail and will recycle back to the compute strategy stage, allowing other strategies to be tried.

- *Bilateral executor*

The simplest approach to executing netting occurs where there are only two participants to the netting operation, requiring either zero or one payments. In this scenario, there is no need to use confidential identities. As such the payment can be constructed using a very simple Corda flow.

- *Atomic payment set executor*

Where we require three or more parties to be involved in a netting solution we must construct a more complex payment transaction. This requires a customized Corda flow.

As we have several parties, there is also a need to obscure the details of those parties such that their real identities are not revealed by the atomic transaction that is constructed. This is not particularly problematic, as the detect phase always operates in terms of confidential identities.

The node running the executor phase initiates sub-flows to each participant. Each will be asked to construct a payment with valid input states, and these will be bundled together into one transaction. The transaction will then be sent to each participant to be checked, verified, and signed, before being finalised through the notary service.

Note that where possible, payments should be made using existing bilateral relationships (as defined by the bilateral obligation graph discovered during each scan). The detected graph does not necessarily elucidate all bilateral business relationships, only those with net obligations, and we should not make assumptions about nodes that aren't connected on the graph.

In the following example, it is possible to perform an atomic net settlement on the basis of existing bilateral relationships:

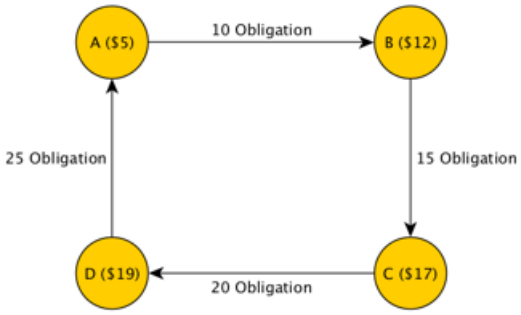


Figure 11: Scenario

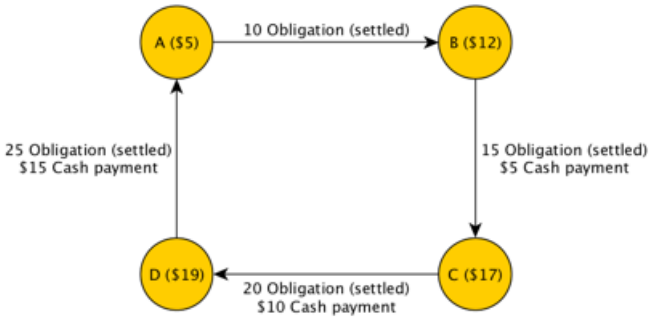


Figure 12: Resolution

This results in the final balances:

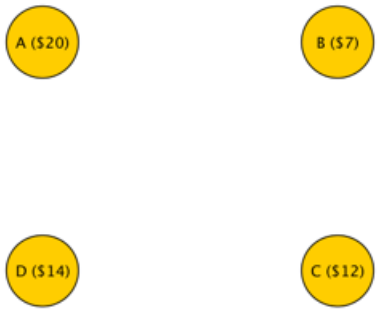


Figure 13: Final Balances

However, not all possible gridlock scenarios can be resolved on this basis, for example if the balances of one or more participants are too low. The example below shows the same scenario except we have reduced node C's balance by $10 so it can't afford to pay the $10 Cash payment to node D.

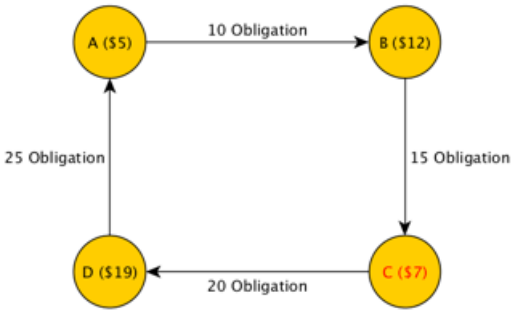


Figure 14: Gridlock with insufficient balance

In this case the gridlock *can* still be resolved, but it requires a payment between *unconnected* nodes, from C to A.

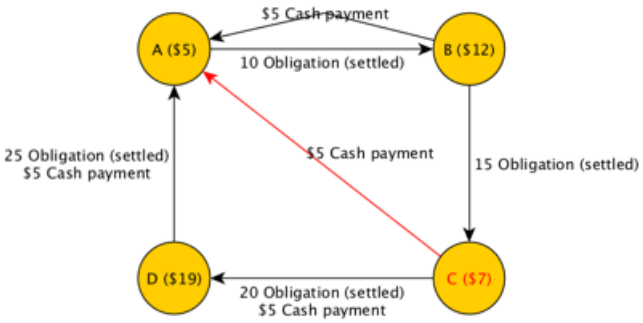


Figure 15: Gridlock requiring payment between unconnected nodes

This results in the final balances as below:

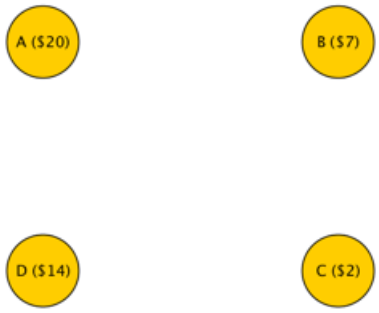


Figure 16: Final balances

As expected, these are the same final balances as in the earlier case, except node C's balance is $10 less.

2.3.2 Illustration using a gridlock scenario

This section describes the process involved when a gridlock resolution cycle is triggered in Corda as per the sample scenario described in Section 4.3.

2.3.2.1 Stage 1: Detect

1. Corda LSM allows any nodes initiate gridlock resolution, in this illustration let put that Bank C initiates gridlock resolution.
2. Bank C initiates a flow to require all neighbouring nodes to propagate a scan request to discover available queued payment instructions in the network.
3. A recipient of the scan request must propagate the same request to all its neighbours to return a scan response message containing its queued payment instruction for gridlock resolution.
4. Upon receiving a scan request, the recipient must respond with a scan acknowledgement message and generate a random key which will anonymise its identity when responding with a scan request.
5. A recipient of a scan request will respond with a scan response based on either one of two criteria
    1. The receiver of its outgoing queued payment instruction is the sender of the scan request.
    2. All its recipients of the scan request that the it had sent to has returned with the scan response.

3. When there are more than one queued payment instructions from one sender to the same receiver, the payment instruction to be selected will be based on the following criteria:
    1. Highest Priority
    2. First-In First-Out (FIFO)

6. Having collated all replies from each neighbour, the recipient of the scan request will send the scan responses back to the requester.
7. Strictly speaking, by responding to the request recursively and via propagation, each of the nodes in the propagation process can observe the transaction amount from each of the queued payment instruction in the scan responses. However, the identities of the sender and receiver participating in the queued payment instruction are anonymised and thus preserving the privacy of the parties involved. At the same time, the data structure used to facilitate the storage of these information is in volatile memory and will be purged upon gridlock resolution completion.

Table 2: Bank B has two queued payment instructions to the same receiver Bank C. Only transaction T2 is selected based on FIFO criteria

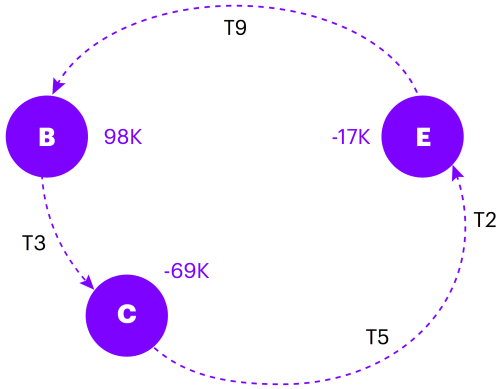
	A (K)	B (K)	C (K)	D (K)	E (K)
Starting balance	3	4	5	4	3
T1	-5	+5			
T2		-6	+6		
T3		-30	+30		
T4			-8	+8	
T5			-80		+80
T6				-7	+7
T7	-6		+6		
T8	+8				-8
T9		+100			-100
T10	+5			-5	

Bank B have two queued payment instructions to the same receiver, Bank C - T2 and T3. Both have the same priority but T2 has an older timestamp than T3. Based on FIFO, T2 is selected and T3 is excluded from this gridlock resolution cycle.

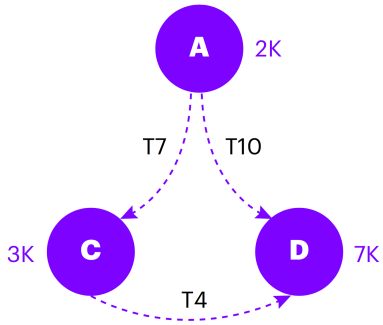
2.3.2.2 Stage 2: Plan

1. Bank A will compute cycles based on the scan responses with its respective sender-receiver of the queued payment instruction
2. Bank A construct an in-memory graph representation based on the sender-receiver edge
3. Based on the cycles, Bank A will calculate the obligation sum (total value of obligations) in each of the cycles

The diagram below illustrates how the obligation sum is calculated.



- Obligation sum = 6 + 80 + 100 = 186
- Netted balance for B, C and E are $98,000, -$69,000 and -$17,000 respectively



- Obligation sum = 6 + 8 + 5= 19
- Netted balance for A, C and D are $2,000, $3,000 and $7,000 respectively

Figure 17: Illustration of how obligation sum is calculated

4. Rank the cycles based on obligation sum

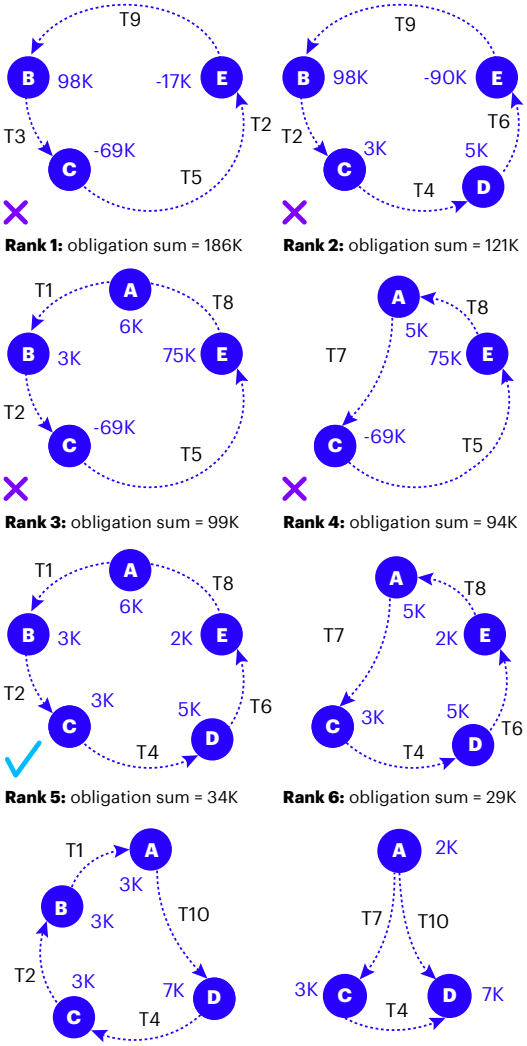


Figure 18: Illustration of how cycles are ranked and eliminated

The cycles are ranked based on highest Obligation sum, i.e. from the largest to the smallest Obligation sum. The first four cycles are invalid as they produce deficits in netted balance. Cycle A-B-C-D-E will be chosen for the resolution as it is the next in rank (Rank 5). The remaining cycles will not be considered though they do not produce deficit in the netted balance.

2.3.2.3 Stage 3: Execute

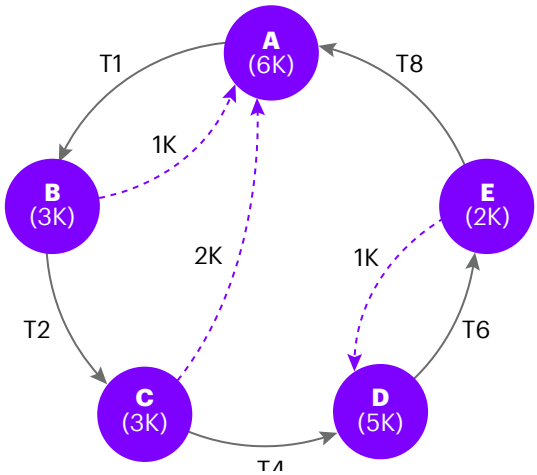


Figure 19: Net settlement transactions to resolve the selected gridlock cycle

Table 3: Status of ten transactions after first round of gridlock resolution

	A (K)	B (K)	C (K)	D (K)	E (K)
Starting balance	3	4	5	4	3
T1	-5✓	+5✓			
T2		-6✓	+6✓		
T3		-30	+30		
T4			-8✓	+8✓	
T5			-80		+80
T6				-7✓	+7✓
T7	-6		+6		
T8	+8✓				-8✓
T9		+100			-100
T10	+5			-5	
Closing balance	3	3	3	5	2

1. Bank A initiates execute based on the input from planning phase.
2. Bank A triggers sub-flows to each participant (B, C, D, E) to construct payment with valid input states to be bundled together in to 1 transaction.
    a. Bank B will construct payment $1000 to Bank A

b. Bank C will construct payment $2000 to Bank A

c. Bank E will construct payment $1000 to Bank D

3. Resolution for Cycle A-B-C-D-E will be executed. Bank A will receive $3000, Bank B will pay $1000, Bank C will pay $2000, Bank D will receive $1000 and Bank E will pay $1000. T1, T2, T4, T6 and T8 are settled.

Table 4: Remaining gridlock transaction after first round of gridlock resolution

	A (K)	B (K)	C (K)	D (K)	E (K)
Starting balance	6	3	3	5	2
T3		-30	+30		
T5			-80		+80
T7	-6		+6		
T9		+100			-100
T10	+5			-5	

The three stages of Detect, Plan and Execute will be reiterated for the second gridlock resolution cycle with the remaining Obligations, T3, T5, T7, T9 and T10. However, all the cycles are invalid as they produce deficits in netted balance. Nevertheless, unilateral payments can be performed to settle Obligations T7 and T10. Bank A has enough liquidity to settle the Obligations of $6000 with Bank C and Bank D has $5000 to resolve the Obligations with Bank A as well.

2.4 Pledge and Redeem

2.4.1 Pledge

Pledge request to DLT would be initiated by bank system to MEPS+, then MEPS+ would perform all required validations to ensure the request is valid. Upon successful verification, MEPS+ would debit the bank account in RTGS and invoke MAS Central Bank Node to perform pledge in DLT. In Ubin Phase 2 prototype implementation, pledge is implemented with:

1. Central bank issues cash states with amount as requested with Central Bank as the owner.
2. Central bank creates transaction to generate payment instruction to the bank that initiates pledge request.

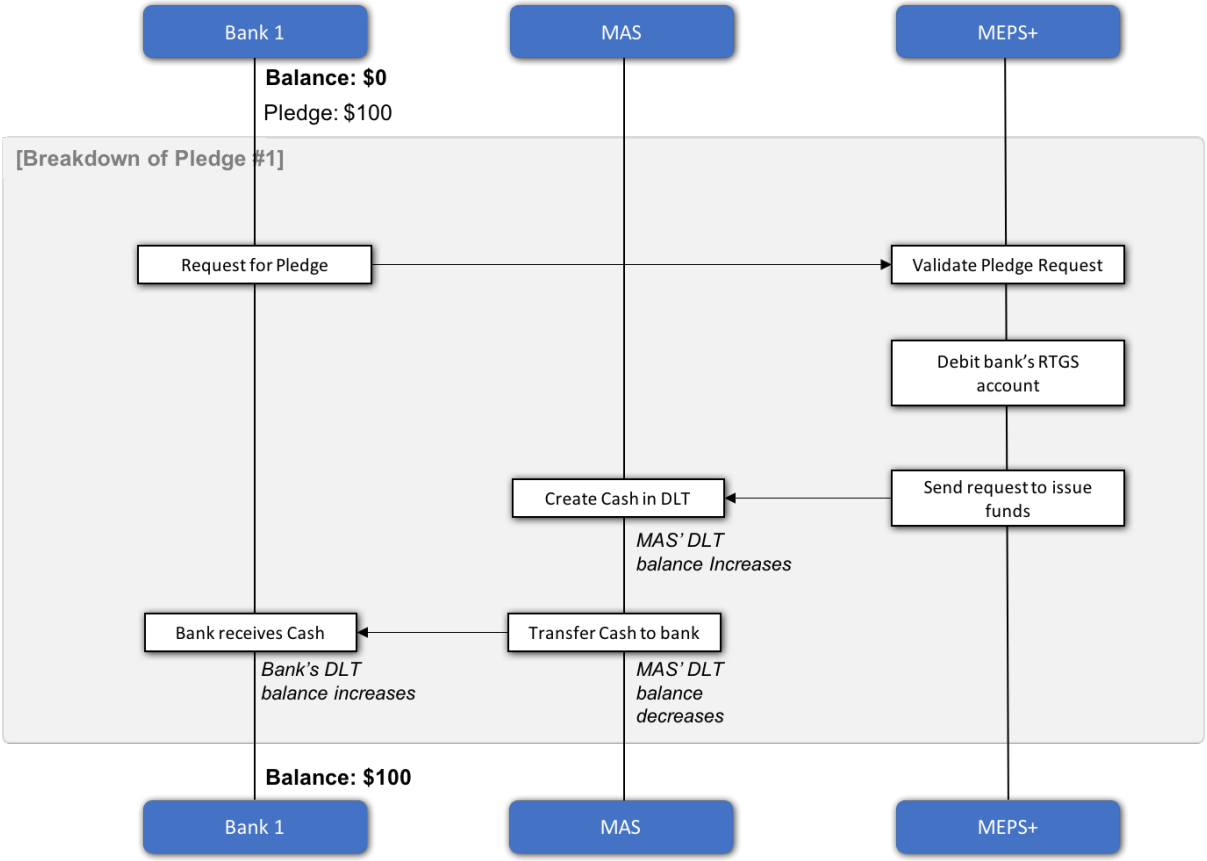


Figure 20: Sequence Diagram for Pledge

2.4.2 Redeem

When a bank performs redeem, the cash is destroyed from the network. This is to ensure there is no net increase of overall money in both the DLT and external systems. In Corda, only the issuer of the cash states can destroy the cash states. Ubin Phase 2 implementation of Redeem is as follows:

1. Bank system initiates redeem request to its own Corda node
2. Upon received redeem request, the DLT node would create a transaction containing payment instruction to MAS with amount as requested and redeem receipt in form of Redeem states to track the redeem requests.
3. Once transaction is completed, the cash states to be redeemed belongs to MAS Central bank and redeem state is created in both initiator bank and Central bank vault.
4. At this point in time, the bank has not received the redeemed money in the RTGS account. Central bank needs to approve the redeem request.

nce central bank approves redeem request, the cash states and redeem states would be destroyed and Central Bank would invoke MEPS+ to credit the money to the bank's RTGS account.

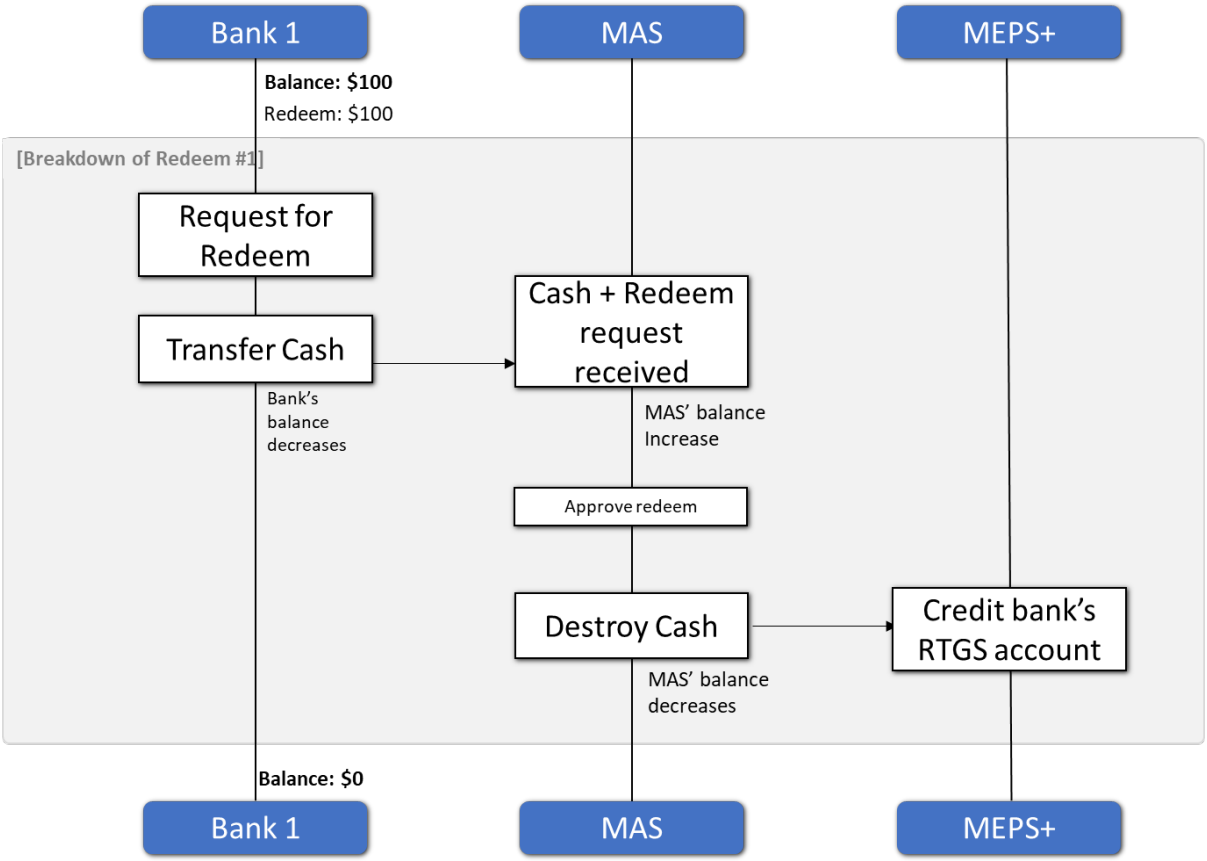


Figure 21: Sequence diagram for Redeem

2.5 Balance Enquiry

Each bank queries its own vault for the unconsumed Cash states and perform a summation on this to evaluate the total balance. Each of the bank in the network can also view historical of the transactions by querying transaction snapshot from transaction storage to get completed transaction such as, transfer funds, pledge, and redeem.

MAS can initiate a flow to broadcast a request to all the banks to query their vault for the unconsumed Cash states. Each bank will verify that the identity of the initiator is MAS before responding to MAS with the *stateAndRef* of the Cash states in its vault. MAS will validate this *stateAndRef* which is done in the Corda platform whereby it resolves the dependencies on those states received with the full history to validate the authenticity of the states. Subsequently, MAS will compute the balances for each bank by performing a summation of the cash amount from the Cash states.

2.6 Manage Accounts

2.6.1 Onboarding a new bank

In a permissioned network, a regulator (or the appointed network operator) would maintain authority to control the banks' membership in the network. By granting a bank to join the network, it effectively allows the newly enrolled bank to send and receive payment instructions with any nodes on the network. A model must be in place where request for enrolment would trigger an automated workflow. Assuming a bank has its individual node, when a node starts up initially, a certificate signing request can be created based on the node's legal identity from the configuration file. The information consists of elements such as the node's legal name and

network map service it is pointing to. The node will generate a new pair of private and public keys that represent its identity. The public key is inserted into a certificate which is sent to the doorman service in the form of a certificate signing request.

The doorman can simply be the one controlling the root CA or acting as the intermediate CA. In a production environment, the signing can simply be done by network operator (i.e. using Hardware Security Module). The doorman initiates a human workflow process which requires validation of the identity of the node, and involves standard verification and checks to ensure that the requester is a valid agent of the entity and is empowered to create a node. When the workflow issues the approval of the node's registration, the certificate is signed over by the doorman service, with its root identity, thus validating the identity of the node. The certificate is then returned to the node which allows permission for the node to join the network, and thus it has successfully registered in the network map for discovery by other peers. The entire process can operate continuously and aimed at not having any downtime whereby existing nodes are not affected by new joiners in any way.

2.6.2 Suspending a bank

As the regulator in the permissioned network, it is vital to have the higher authority to suspend a bank aimed at preventing it from performing or receiving future transactions. Corda can adopt and reuse common and proven framework, one of it being PKIX for the management of keys and identities. To suspend a bank, typically a certificate revocation process can be utilised where there exist a certificate revocation list (CRL) which consist of certificates used by banks that can no longer be trusted in the network. It is possible that revocation of a certificate can be temporarily removed from the list and reused in the future, but a good practice requires revocation lists to be immutable and append-only. For a bank to be removed from suspension, it is preferred to have the old certificate fully revoked and then reissued.

Because a signed certificate is a key part of the identity of a node used during the establishment of message queues between nodes, when a certificate is revoked and not accepted as valid then the node effectively is unable to communicate with other nodes to update the ledger, and is therefore removed from the network. Additionally, this approach ensures other nodes on the network are not affected.

A long-term suspension of bank should be followed by a long-term resolution where it may involve resolution from business, governance or a legal authority. The smart contracts held by the suspended banks (i.e. Obligation) should no longer be valid and therefore a higher authority will have an overriding authority provisioned by the smart contract to allow transfer or expiry of said contract. An emergency override for exceptional use-cases can be provisioned within the contract code, which will also depend on the business and governance requirements.

2.6.3 Updating a bank's well-known name

It is recommended a bank maintains its well-known name off-ledger where it can freely change it (i.e. due to mergers) as long as the unique identity associated with the bank in the network stays the same. In the event that a regulator would want to revoke the old name and reissues a new legal identity, a workflow needs to be in place.

The identity of a node is represented by the X.509 certificate issued to the node. This certificate contains the distinguished name of the node and is signed over by the root CA of the network. A new legal name would require a new certificate to be resigned and re-approved by the CA to re-join the network. There are implications to existing states (either consumed or unconsumed) where it would continue using the old identity. Therefore, a transfer process must be provisioned to allow the states to be transferred to the new identity. A future development by R3 will enable a new certificate with a different name to be issued for the same public key, and that the new identity will be able to be treated as an alias for the original identity.

2.6.4 Storing a bank's public keys

A regulator with higher authority can have access to and permission to store the public keys of all banks participating in the network to securely identify them. All well-known public keys and formal names for network participants are held within the network map so the identities of each participants can easily be established at the network level. The network map is highly available and decentralized to cater for high resiliency and minimal disruption to other nodes.

If necessary, confidential identities can also be created on a per transaction basis (once nodes have already established the transaction participant's identity); these confidential identities are then used to sign over the transaction. This means that while the transaction is legally binding, only the participants of the transaction can know who they are. If a regulator wants to understand these identities then the nodes will need to sync them to the regulator by custom flows.

2.6.5 Updating a bank's public key

In a precautionary process where banks have to rotate their public keys in a timely manner or an unexpected event where a bank's private key is compromised, a regulator would require authority to enforce and facilitate the reissuance of a bank's new public key.

This process requires the use of certificate revocation workflow and the transfer of states to the new keys. There is repercussion if the private key has been lost, where the signing of transactions to move or change states is no longer possible, except if composite keys have been used to control the states. If the private key has been compromised, then it is still possible to move or change states and this should be done immediately unless the revocation check is extended to the Notary (and it is validating signatures).

There are multiple key types involved in Corda. Some of the keys below can sign certs for other legal entities inside an organisation.

- Notary transaction signing key
- Oracle transaction signing key
- Party's transaction legal key
- Party's transaction anonymous key
- Doorman certificate signing key
- TLS keys for all the above

Assets are typically signed by a party's anonymous keys for which there is no accompanying cert (on the notary, network map and doorman side). If the anonymous key is lost, the solution is to transfer it as fast as possible. If the legal key is lost, then the assets controlled by the anonymous key can still be moved under a new legal key.

Another option to circumvent this problem is to always sign with composite anonymous keys (e.g. 3 out of 5). Key management becomes more complex and it will affect transaction speeds, but now 3 of the keys must be compromised.

For a node's certificate, the following events might drive changes:

1. A node has been identified by its operator or by the network operator as compromised in some way at some prior time, but the event was detected retrospectively. Assurance that the node was in a trusted state prior to the event is poor. A manual reconciliation will need to take place to reconcile current states and determine how much of the node's vault is trustworthy. Key material will need to be replaced. A process of recertification may need to take place before a new certificate may be issued to that node. This event would trigger an emergency revocation of the node's certificate.
2. A node has been identified by its operator as potentially compromised. The event has been detected in real time. Assurance that the node was in a trusted state prior to the

event is good. The node has been taken out of service in real time and zero, or a very limited number of signatures have been created using the key in a compromised state. The node can reasonably be restored to a trusted state. No recertification of the node is required. A new certificate will be issued to the node to reflect the change in public key. This event would trigger an orderly revocation of the node's certificate.

3. A node is subject to a transfer of administrative responsibility. The point at which transfer will take place is known. Cryptographic material will need to be changed at the point of transfer, a new certificate will be issued to the node to reflect the change in public key. This event would trigger an orderly revocation of the node's certificate.
4. A node is required to change its key material or certificate attributes as a result of legal name change, policy, regulation or edict from the Network Operator. The change may take place in a controlled manner. No cryptographic material is compromised. A new certificate will be issued to the node to reflect the change in key material. This event would trigger a routine reissue of the node's certificate.

2.7 Versioning

2.7.1 Accessing CorDapps version

Corda can confirm and expose the versions of other nodes, and of the CorDapps running on these nodes. CorDapps will define minimum version numbers for both characteristics to successfully transact and that failure to meet these would result in transactions being rejected. At the transaction level, each participant must process due diligence in terms of validating the received flows' corresponding version and the CorDapp appName hosting the flow to determine the backward compatibility and implementation run by the other party. A recipient of the flow can accept (by signing) or reject the flow if it fails to meet certain degree of standard or requirements (i.e. version)

2.7.2 Upgrading CorDapps version

A regulator may want to enforce the deployment of the latest CorDapps code version to all nodes on the network. An approach involving a minimum acceptable version will ensure the safe operation of the network. Deployment of new versions of CorDapps is straightforward, and existing states can be upgraded to the latest version. All parties including the central party and regulator must agree to implement the new CorDapp version. All parties will check the version of CorDapp by each node and only accept the session request if it, firstly, has that flow loaded, and secondly, has the same version of contract codes.

In regard to forcefully enforcing a CorDapp version upgrade, there are many practical considerations where this intent may be unwise, where alternatively it should be handled as a contractual issue. Forcefully upgrading a CorDapp risks removing, modifying, or adding new Corda state objects, with an unquantifiable impact on other CorDapps, or node applications. These risks include reduced transaction performance, increased transaction latency, increased storage requirements, increased memory usage, and the failure of other CorDapps, or other applications (i.e. reporting, analytics, etc.) that might be dependent on the representation of Corda state objects. In essence, this represents the same class of problem that would occur if a regulator were to be able to enforce an upgrade of any other piece of IT infrastructure without allowing the operator of that infrastructure to perform the necessary due-diligence checks and taking any necessary precautionary steps prior to deployment.

It is also inevitable that contract code will evolve and a new version will replace the old. Each of the participants in the upgrading contract willing to upgrade the state object must initiate the authorization via RPC call to trigger a ContractUpgradeFlow. After all parties have registered the intention of upgrading the contract state, the regulator node can initiate the upgrade process by running the contract upgrade flow. The Instigator will create a new state and sent to each participant for signatures, each of the participants (Acceptor) will verify and sign the

proposal and returns to the instigator. The transaction will be notarised and persisted once every participant verified and signed the upgrade proposal.

2.7.3 Upgrading Platform version

In the event that a regulator wants to enforce that all nodes on the network is on the same platform version, the same implications from upgrading CorDapps applies where the upgrade of any piece of IT infrastructure must be performed with due-diligence. Corda nodes will check the version of Corda that the counterpart nodes are running, and each version of Corda will have a minimum counterpart version of Corda required for interoperability, supported by API and over-the-wire protocol stability. This means that global, simultaneous update of Corda nodes by all participants should be avoided.

The Network Map Service (NMS) can use the Platform Version to enforce a minimum version requirement for any registering nodes. It will be a legal obligation to advertise the correct platform version. The NMS will not enforce a maximum version requirement. Changing the NMS's minimum version requirement will only take effect on new registrations. Existing nodes which don't meet the new minimum requirement will not be de-registered. Policy for upgrading the network is a separate process outside the scope of this design. The Platform Version will also be set in the header of every Message Queue (MQ) message.

As stated, Corda will check a counterpart's Corda version during messaging handshake process, but that would still allow 2 parties to run an old version of Corda between each other if they used the cached copy of the network map. The extra control point is that the network map will maintain the configuration of minimum versions allowed on the network, and will only publish addresses of nodes that meet this criterion, and this will force network participants to maintain currency with a recent version.

3 Technical Specifications

3.1 Transaction Validity

The distinct UTXO model in Corda requires input states to link one or more inbound transaction by its transaction id that created it, resulting in an immutable chain of asset lineage. In every transaction, each of the Corda nodes will 'walk-the-chain' to verify each input was generated from historical valid transactions which ensures the authenticity of the chain.

Ensuring all transactions occurring within the system are validated is an important feature in Corda, the sender bank will validate the outgoing transaction by attaching its own signature onto it before the transaction is sent out to the receiver. The receiving bank will then validate the incoming transaction by running the "contract verify", which will ensure that all the rules of the state is followed, while going through the chain of the state historical signature and transition. The notary has the final authority to sign off the final signature when it has validated that all proposed input states have not been spent and is valid for use. All completed transactions that has the notary signature will prevent another node from revoking or annulling the transaction. Therefore, this will ensure that the transactions are final and irrevocable.

3.2 Privacy

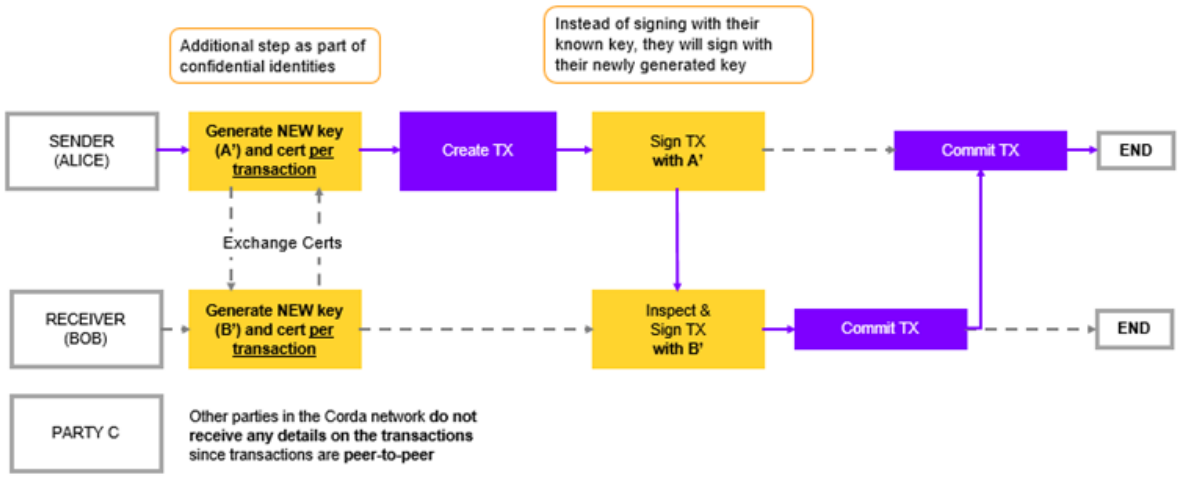


Figure 22: High level flow of transaction with Confidential Identities

Corda distributes the ledger based on a need-to-know basis instead of a global broadcasts method, and each peer only sees a subset of facts on the entire ledger, therefore only the sender and receiver party will process, validate and record the transaction. Additionally, at the start of any transaction, leveraging on Corda's concept of **Confidential Identities**, a new unique pair of public key and certificate is generated and exchanged between the sender and receiver of the payment instruction. These new **Anonymous** identities are only known to both sender and receiver parties. This helps to shield the parties that are involved in payment instruction so that parties who are not involved are not able to identify the participants. This is important to preserve privacy especially due to the nature of the UTXO model employed by Corda.

In the diagram above, a sender initiating a peer-to-peer transfer of funds would first generate new key pairs and certificate before constructing a new transaction. Both the sender and the receiver involved in the transaction would send the new confidential identities to each other. The public keys of these confidential identities are used in the transaction when generating the output states, commands and signing of the transaction. After all of the validation and signing by the participants have been performed, the notary then verifies the uniqueness of the states and sign with its signature as well. Upon notarisation, both the sender and receiver would commit the final transaction with its output states to its respective ledgers. This approach allows the privacy of the transaction to be maintained in real time and in future transactions where the lineage of states used as input states do not reveal the identities of the past participants.

In this process, Confidential Identities assures that only the sender and receiver parties can identify the identities in the payment instruction, however other details in transaction do remain visible. In a peer-to-peer transaction, there is an assumption that it is a shared fact between the transacting parties therefore the details need not be hidden since no other parties will receive this transaction. In the case of Gridlock Resolution, the transaction amount does get revealed to participating parties but the identities are still hidden. It is a trade-off between hiding identities versus transaction details and future effort is still on going to tackle this specific area.

3.3 Technical Matrix

Application	Version
Corda	1.0.1 (Ubin specific)

4 Interface Specifications

Refer to: https://github.com/project-ubin/ubin-docs/blob/master/api/UbinPhase2-CordaAPI.pdf

5 Key Observations and Findings

5.1 Privacy

The Corda network distributes the ledger based on a need-to-know basis instead of a global broadcast method, so only parties involved in a particular transaction have visibility of transaction details. This model inherently addresses privacy concerns.

The Corda prototype strengthens privacy in its design with an additional layer of Confidential Identities added to each transaction, whereby only parties involved in a transaction can identify the participants. These participants exchange the fresh key and certificate using Corda's 'Swap Identities Flow' and the new keys are used for the output, command and signature of the transaction. While both the sender and receiver details are anonymised, the transaction amount is not. The transaction amount is required to enable the 'planning' phase of gridlock resolution algorithm to compute the best gridlock resolution opportunities based on the amount of the queued payment instructions.

In the current prototype of a small network, exposing transaction amount in the network may lead to privacy concerns where it may be possible for a network member to attempt to graph the network and deduce the sender and receiver of each of the queued payment instructions. However, such a mapping would be more complex for larger payment networks than MAS MEPS+ that already has 63 participating banks and average of 6,000 transactions in a single gridlock resolution cycle.

5.2 Scalability and performance

In the design of the Corda workstream, adding a new participating node involves only the installation of the new node itself. Minimum change is required to existing nodes or existing network setup in the process of adding new nodes.

In Corda, transactions are only sent on a need-to-know basis and each peer only sees a subset of facts on the entire ledger. This alleviates the scalability and performance issue commonly faced by traditional DLT platforms, which store and update the entire ledger on every peer. At the same time, in industries where high transaction throughput is appreciated, multiple notary services allow load balancing and an increase in transaction throughput.

The distinct UTXO model in Corda requires input states to link one or more inbound transactions by their hash, resulting in an immutable chain of asset lineage. This lineage chain could be 'long and heavy' especially after a long duration and many cycles of transactions. In every transaction, each of the Corda nodes will 'walk-thechain' to verify each input was generated in a sequence of valid transactions, validating the authenticity of the chain. This

can be easily overcome with implementing an expiry to the digital assets. In other words, the pledged funds will need to be recycled after a period of time.

Additional exceptional scenarios were considered during the project:

EXCEPTIONAL SCENARIO	OBSERVATIONS
Impact of injecting transaction(s) to the network during gridlock resolution	New payment instructions can be processed while gridlock resolution is running. In addition, any obligation that is used during a gridlock resolution can be cancelled, or reprioritised without disruption. In the case where the incoming transaction uses the same states involved in gridlock resolution process, the first transaction to be notarised would succeed and the second would fail, because Notary would reject the transaction due to double spending.

5.3 Resiliency

If any of the participating bank nodes are unreachable, for example machine failure, network issues, etc., the network can still operate for all transactions that do not require the involvement of the failed nodes. At the same time, nodes can be shut down and restarted at will due to Corda's 'Flow Checkpointing mechanism', ensuring data is never lost and flow progression is protected. In addition, fund transfer transaction can still proceed among participating nodes whom are not involved in the gridlock resolution cycle.

For this prototype, a single Simple Notary service was used, introducing a potential single point of failure. If the single notary failed, transactions cannot be completed. As an observation, further enhancement can be done to implement the notary service as a cluster potentially being operated by multiple parties. Such a solution can be designed to support load balancing to increase transaction throughput, multi-threading of incoming transactions and minimising latency for geographically diverse transacting parties.

Exceptional Scenario	Observations
Impact of removing 1 participating bank during gridlock resolution	The current prototype does assume that the production version ensures High Availability (HA), therefore it does not support removal of any participant node during gridlock resolution cycle.
Impact of removing MAS during gridlock resolution	There is no impact to the network (other participating nodes) as MAS is not required to orchestrate the gridlock resolution as well as any transactions that MAS is not a participant of.

5.4 Finality

In Corda, the notary service provides the point of finality in the transaction where the presence of a notary signature indicates transaction finality. By obtaining a notary signature, participants of the transaction can be equally sure that the input states are unconsumed (unspent) by prior transactions.

Due to the notary model, a transaction's proposed changes are either all accepted or none are. Any queued payment instruction that is involved in gridlock resolution can be modified

freely (reprioritised or cancelled), and new fund transfers can be initiated and settled in real time. The netting solution can guarantee atomicity when it fails either because the graph is no longer valid due to modified payment instruction or a decrease in balance.