UBIN PHASE 2

Technical Documentation

Quorum

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

Table of Contents

1 Platform and System Designs

Quorum is created for the financial services industry as an Ethereum-based distributed ledger that supports transaction and contract privacy. Quorum shares private information in a point to point manner on a need-to-know basis. In addition to privacy, Quorum adds further enterprise centric features on top of Ethereum such as transaction finality, performance benefits and network permissioning. Since it is designed to operate in permissioned networks, Quorum removes Ethereum's Proof-Of-Work and Proof-Of- Stake consensus mechanisms, replacing these with a selection of voting-based consensus mechanisms that users can choose from. It is a fork of the Go Ethereum client(geth), and is designed to be developed in line with future geth releases.

In Ubin Phase 2 design uses Quorum's Raft consensus mechanism, a formally verified consensus mechanism that is based on the etcd Raft implementation that underpins widely used software such as Kubernetes. Raft provides faster blocktimes (in the order of milliseconds instead of seconds) and transaction finality (the absence of forking). In addition to Raft, Quorum Constellation provides transaction privacy. Further the design implements Zero Knowledge Security Layer(ZSL), a protocol designed by the team behind Zcash, that leverages zk-SNARKS – a variant of zero-knowledge proofs (ZKP) – to enable the transfer of digital assets on a distributed ledger without revealing any information about the Sender, Recipient, or quantity of assets that are being transferred, and without requiring any central party to affect the transfer.

For this project, the prototype was developed using ***Quorum version 1.5 (Ethereum version 1.5 with Constellation version 0.1.0)*** with Solidity as the programming language and Node.js for the application layer.

All nodes in a Quorum network run the same set of components; there are no centralised or function-specific nodes that require different configurations. This ensures that the network is truly decentralised with no single point of failure.

Ubin Phase 2 design goals:

- Transaction Privacy requires transaction details that is available only to the parties participating in the transaction and a few other selected parties. This is solved by constellation and the transaction hash is injected into the entire network but the transaction payload is only shared to the nodes that are part of the private transaction.
- RAFT consensus mechanism is used to resolve the settlement finality on the real-time gross settlement basis based on deterministic rather than probabilistic finality. Since RAFT can only have a single leader node, there is no likelihood of a blockchain fork and therefore it can guarantee settlement finality once a transaction is completed.
- Decentralized Processing eradicates the centralization of processing and thus, avoiding single points of failure by creating a distributed and resilient infrastructure. Ethereum is a blockchain that is made for full decentralization. It has been tested expansively and is the foundation on which Quorum is built on.
- Liquidity optimisation ran Queuing, Netting, Gridlock and Deadlock resolution algorithms operating on private transactions without the need for centralized processing which ZK-SNARKS (Zero Knowledge succinct non-interactive argument of knowledge) or Zero Knowledge Proofs is used. This is a technology that allows each node on the network to validate every transaction without having any knowledge of that transaction.

1.1 Logical Architecture

1.1.1 DApp - Application and API

The decentralised application (DApp) functions as the orchestrator of the payment process and acts as a bridge between the public and private contracts. It contains all orchestration flow and. logic. The DApp accepts requests from client through a RESTful API interface and invokes the appropriate sequence of smart contract calls. The DApp, utilising the Web3 library to interact with smart contracts via RPC through HTTP and listen to the event to be raised which will trigger calls to the appropriate services. The DApp also calls an off-chain binary located on the Quorum node to generate and verify ZKP proof.

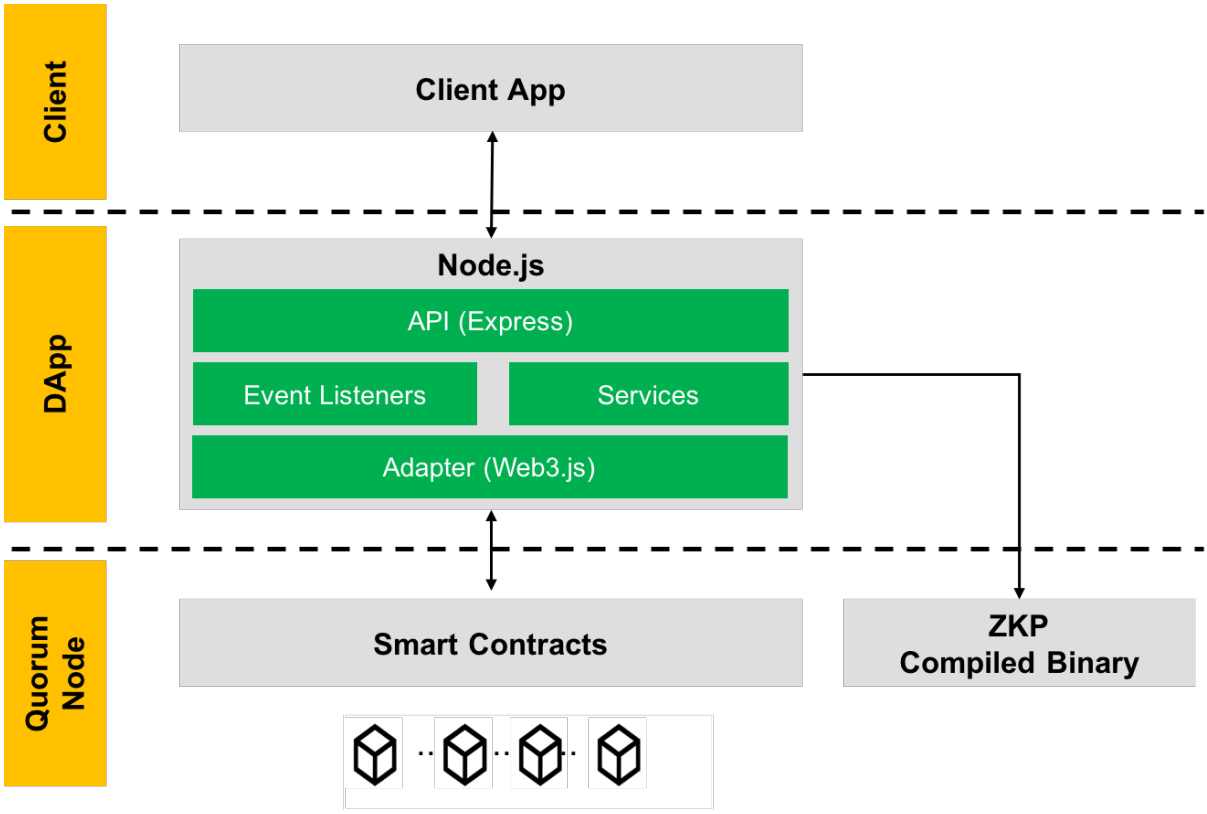


Figure 1: Logical Architecture

1.1.2 Event listeners

Events emitted by the smart contract are used in two ways by the DApp:

1. Return values when performing call transactions - The DApp will invoke smart contract methods by sending transactions and will receive responses by events that are emitted in the form of returning promise values.
2. Listening to events that get emitted due to actions taken by other parties on the network - A bank receives notification that another party has acted by listening to events that get emitted. For example, when a sender submits a fund transfer, an event will get emitted that notifies the receiver

The DApp is stateless and relies on the data stored in smart contracts to carry out operations.

1.1.3 Services

Smart contract method invocations are performed by the DApp's services layer. The event listeners will respond to emitted events by calling functions in Services.

Services are broken down by the following area:

- bank
- funds (cash)
- netting
- queue management

1.1.4 Quorum Node

Key components in the Quorum design as depicted below:

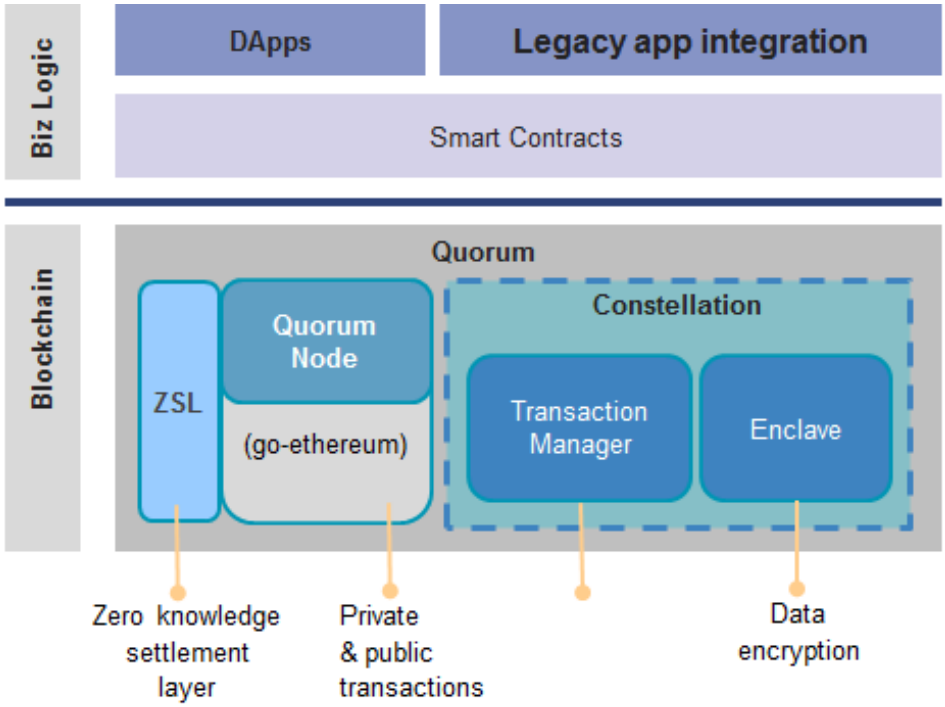


Figure 2: Key Components

- Quorum Node: geth fork - enhanced to provide privacy, permissioning and configurable consensus
- Constellation: Manages transaction privacy and holds encrypted transaction payloads, in this case the inventory of the encrypted “Unconfirmed” and “Confirmed” payment transactions
- Smart Contracts: Public smart contracts (i.e. network wide) and private smart contracts
    - Public contracts (available to all participants) : netting & zero knowledge proof validation
    - Private contracts (for involved parties only): Payment transfer and bank balance visibility
- Zero knowledge security (ZSL) Layer: Component that integrates zero knowledge proofs into Quorum. Provides decentralised privacy during Liquidity Savings Mechanism execution and payment transfers. Zero knowledge proofs (ZKP) are a critical feature of the design. ZKP through zk-SNARKs (a novel form of zero knowledge cryptography) proves to the entire network that a particular transaction can be carried out and the initiating bank has enough balance to perform the transaction, *without revealing anything about the sender, recipient or balance to anyone else on the network.*

For Ubin Phase 2, each Quorum node (banks and MAS) is hosted in individual Azure virtual machines as illustrated below:

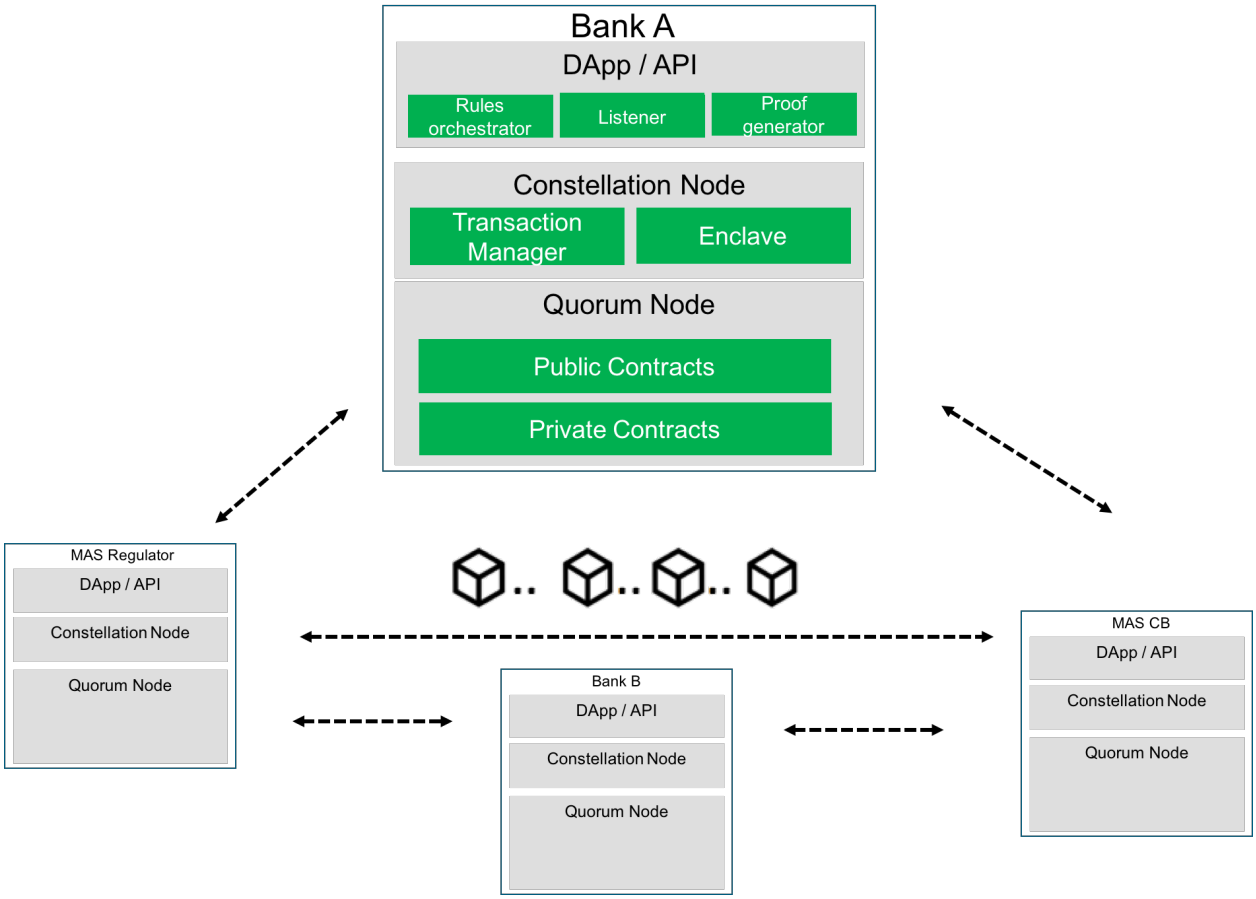


Figure 3: Ubin Phase 2 Quorum Network

1.2 Physical Architecture

Ubin Phase 2 prototype on Quorum has 13 nodes deployed across 13 virtual machines (VMs) in Azure. Each virtual machine per node (bank) is configured with:

- E2 v3 (2 vcpu, 16 GB RAM)
- 30 GB OS disk

The network contains the following set of node roles:

- 1 MAS Regulator node
- 1 MAS Central Bank node
- 11 Bank nodes

The Raft network is configured and setup using the Quorum Network Manager which generates the genesis file and setups up geth and constellation on each node.

https://github.com/jpmorganchase/quorum/wiki/Quorum-Overview (see appendix for setup steps)

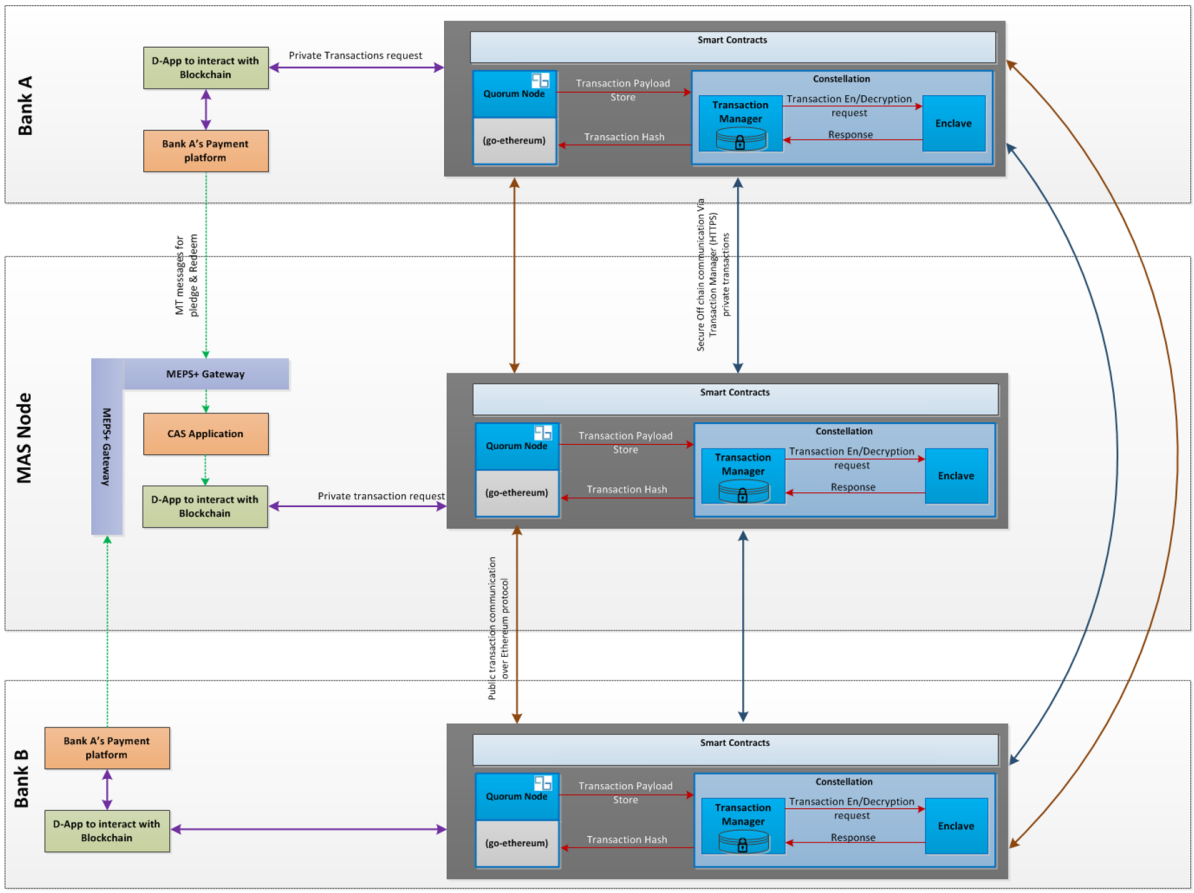


Figure 4: Physical Architecture

2 Functional Specifications

2.1 Fund Transfer

In Quorum's design, funds transfer is executed privately between two parties, with no other party seeing the details of this. Balance validation is performed by the entire network through the use of zero-knowledge proofs. To achieve this, both private and public smart contracts are utilized: private contracts allow for the bilateral payment transactions between two parties. Public contracts store shielded salted balances for each of the participants in the network. For every transaction, the sender and receiver generate Zero- Knowledge-proofs and submit the same for verification by the entire network. Once the network verifies the proofs, the shielded salted balances for the sender and receiver are updated in the public contract. The transaction is marked as complete on the private contract only after the completion of the proof verification and update of shielded salted balance. The decentralised application (DApp) functions as the orchestrator of the payment process.

- The payment instruction for fund transfer is initiated from the sender's DApp. The DApp sends the payment instruction as a transaction to private contract that is available only to the sender and receiver.
- The contract performs a liquidity check.
    - If there are insufficient funds, smart contract returns response with a flag, and DApp would continue to orchestrate with adding transaction to the hold queue.
    - If the sender has sufficient funds the payment would be set as 'unconfirmed' and notify both sender and receiver to generate proofs.
- The sender's DApp generates a zero-knowledge proof that takes in a simple mathematical formula of ***starting balance - amount sent = ending balance***. At the

same time, the receiver's DApp also generates its proof with a different mathematical formula of ***starting balance + amount received = ending balance***.

- The sender and receiver's DApps submits the proposal containing the proof along with salted hashed balances and salted hashed payment amounts to the network for verification. The hashing of payment instruction amounts uses the SHA-256 algorithm with a unique salt dynamically generated per transaction. Once the network verifies these proofs, the shielded salted balance on the public contract is updated and the event is published to sender and receiver DApps.
- The receiver's DApp then send an instruction to private contract to update the payment instruction status as confirmed.

This process therefore guarantees that parties are updating their balances correctly post-payment, without revealing anything about the payment to anyone else, and without requiring a central party to ensure balance integrity.

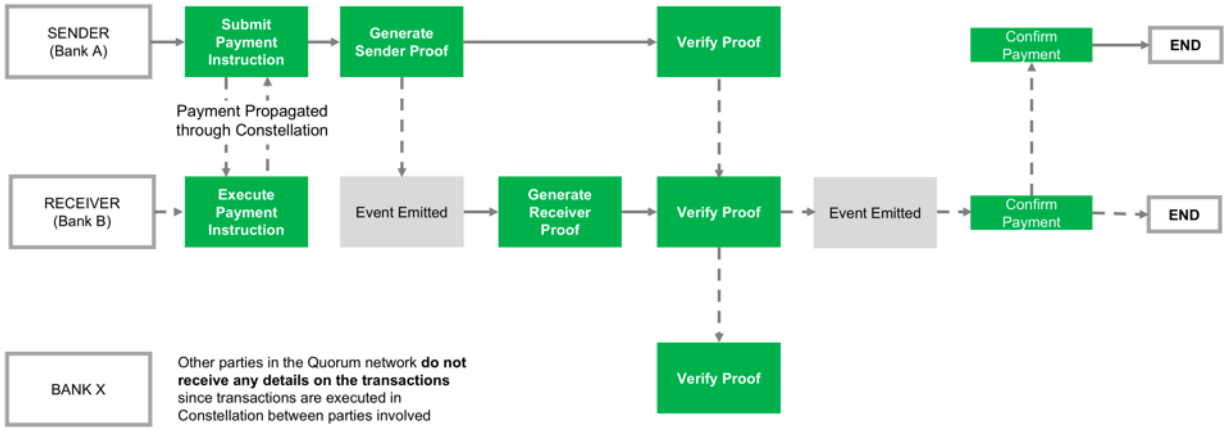


Figure 5: High Level Fund Transfer Flow

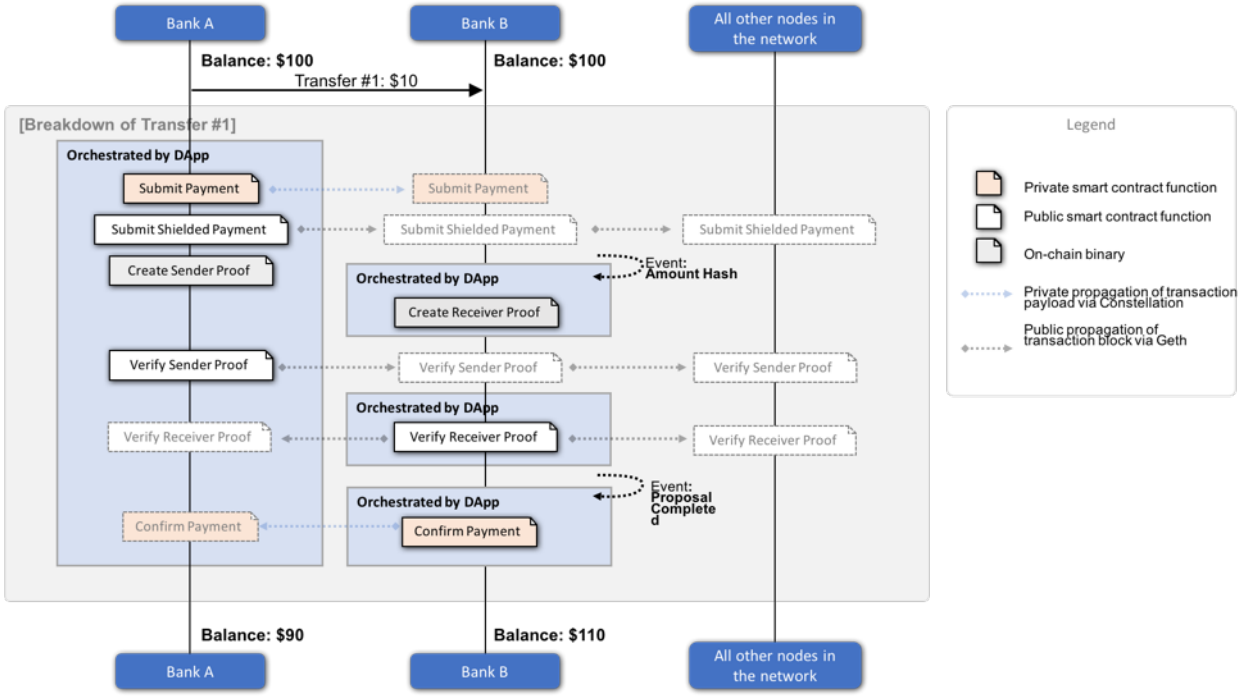


Figure 6: Sequence Diagram for Fund Transfer

The figure above depicts how Quorum orchestrates a fund transfer. In this example, Bank A performs a transfer of $10 to Bank B. Because Bank A has sufficient balance, the transfer is processed successfully and results in a change to both banks' balances.

Note that in the following explanation of the fund transfer flow, the terms *fund transfer* and *transaction* carry different meanings. *Fund transfer* refers to the action of sending funds between banks. The term *transaction* refers to sending an instruction to execute code on the smart contract.

1. The fund transfer is initiated by Bank A through its decentralised application (DApp). The DApp generates a **privateFor** transaction with a payload containing the payment details (e.g. fund transfer amount) and the public key of Bank B. Quorum recognises this transaction as a private transaction between Bank A and Bank B. The encrypted payload of the transaction is shared securely with bank B and hash of the encrypted payload is shared with the network. This invokes a function in the Payment Agent smart contract which creates a record of the payment for only the 2 parties involved. The Payment Agent smart contract also performs a liquidity check on Bank A's balance. The other nodes possess a record of the transaction happening but do not have visibility of the transaction payload. At this stage, the transaction is marked as unconfirmed.

2. Bank A's DApp hashes the fund transfer amount with a dynamic salt. The hashed amount is then sent in a **public** transaction which invokes a function in the SGDz smart contract. This creates a record of the payment for all participants in the network. The payment is visible to all nodes but has its payment amount shielded. An Amount Hash event is emitted upon successful creation of the shielded payment. This event triggers the creation of Receiver Proof by Bank B's DApp (described in Step #5).

3. Bank A's DApp invokes an on-chain binary to create a zero-knowledge security layer (ZSL) proof from the sender's side. This Sender Proof is generated using the following parameters (note that all amounts and balances are salted):
    1. Hash of Bank A's ending balance after the fund transfer
    2. Hash of the fund transfer amount
    3. Hash of Bank A's starting balance before the fund transfer
    4. Bank A's ending balance after the fund transfer
    5. Fund transfer amount
    6. Bank A's starting balance before the fund transfer

4. Bank A's DApp submits a **public** transaction containing the proof generated in Step #3. The transaction executes a function to verify the Sender Proof. Since it is a public transaction and will be executed by all nodes to verify the proof. ZSL is used for verification, enabling verification to be performed without having to reveal any fund transfer details.

5. This step is executed once Step #2 emits the Amount Hash event, and can be concurrent with Steps #3 and #4. Bank B's DApp watches for the Amount Hash event, and when the event is detected it begins to create the Receiver Proof using the parameters listed below. Note that all amounts and balances are salted:

    1. Hash of Bank B's starting balance before the fund transfer
    2. Hash of the fund transfer amount
    3. Hash of Bank B's ending balance after the fund transfer
    4. Bank B's starting balance before the fund transfer
    5. Fund transfer amount
    6. Bank B's ending balance after the fund transfer

6. Bank B's DApp submits a **public** transaction containing the proof generated in Step #5. The transaction executes a function to verify the Receiver Proof. Upon verification, a Proposal Completed event is emitted to signal that the proofs have been verified and the fund transfer is legitimate.
7. Bank B's DApp watches for the Proposal Completed event and generates a final **privateFor** transaction to complete the fund transfer. This transaction, with Bank A as the counterparty, invokes a function in the Payment Agent smart contract that changes the balances of the respective banks. Based on the payment record created in Step #1, Bank A's balance is deducted while Bank B's balance is increased in a single atomic transaction.

8. Once the public side balances are updated, the same is notified to both Bank A and Bank B's DApps. Bank B's Dapp then triggers a function call on the smart contract to mark the status of transaction as confirmed.

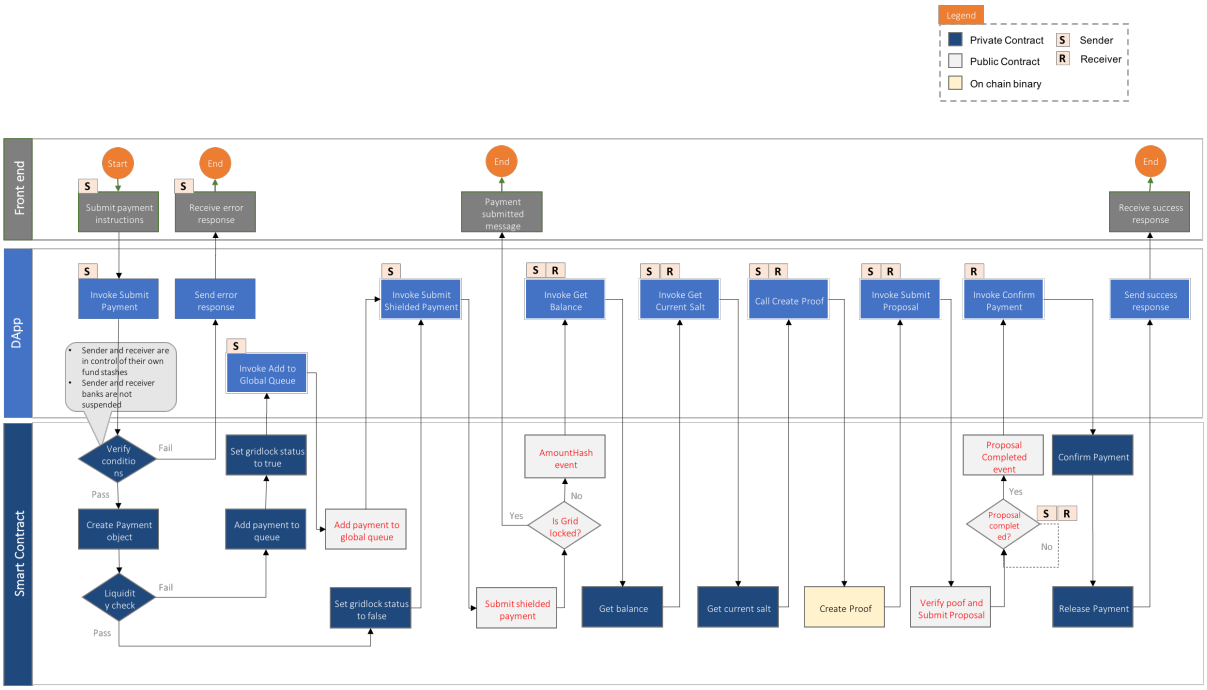


Figure 7: Fund Transfer Process Flow

2.2 Queue Mechanism

Each bank maintains its own Private Queue which is a list of payment instructions that are not yet settled. Additionally, the system uses a Global (pubic state) Gridlock Queue to track all queued payments system-wide and trigger gridlock resolution. Both queues only hold the reference ID of the payment instruction, so that privacy of queues is maintained. Information about the payment instruction such as timestamp, amount, status, priority level and receiver is stored in the payment object itself in the private queue. Only the counterparties to the payment have access to this data.

When a fund transfer is performed, the system checks whether the bank has sufficient liquidity to make the transfer. Insufficient liquidity results in the payment instruction being added to both the Private and Global Gridlock queues.

The gridlock queue is maintained in a two-level sort by High priority and then by timestamp FIFO (ascending order). Therefore, all High priority queued payments sorted by timestamp will appear above all Normal priority payments sorted by timestamp.

For a transaction between two banks with Bank A acting as the sender bank, Bank A will submit the transaction payment. If there are insufficient funds, the transaction will go into queue according to the priority levels. If there are any pending payment instructions in the queue, the transaction will be placed at the end of the queue as per priority and timestamp.

2.3 Queue Settlement

When the bank receives additional funds through fund transfer, pledge or gridlock resolution, there will be an attempt to use the additional funds to settle the payment instructions in queue. The settlement order is based on priority level and FIFO as per the general queue mechanism logic established for Ubin Phase 2. Upon settlement, the payment instructions are removed from both Private and Global Gridlock queues.

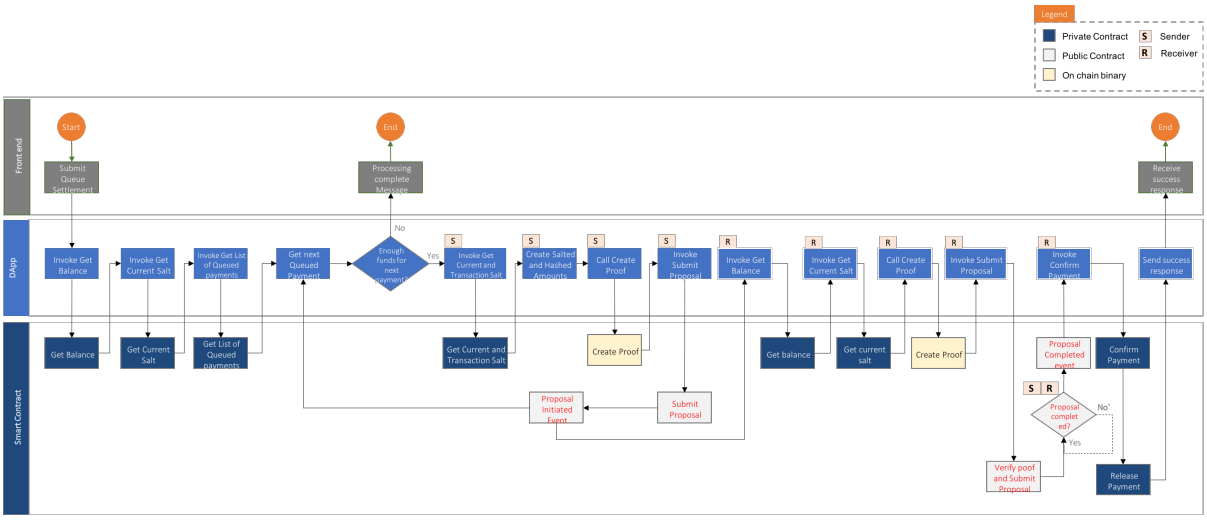


Figure 8: Settle Queue Process Flow

2.4 Queue Management

Several actions can be taken by the bank on the payments in their outgoing queue as part of queue management. The available actions are hold/unhold payments, changing between high and normal priority, and cancelling a payment. For any payment in grid lock queue, user can:

- Cancel the payment. In such cases the payment will be removed from the global and local grid lock queues
- Marked as a on hold payment. In such cases, the payment is removed from global and local grid lock queues and will be added to a hold payments queue.
- Change the priority of the payment

2.4.1 Change Status

A bank can change the status of a queued payment to Hold, this will freeze the payment from participating in any LSM or settling process. Putting a payment will also increase the banks position balance as they effectively no longer have an outgoing payment to another bank.

Position is calculated by taking the bank's current liquidity and adding all pending incoming and outgoing funds transfers form and to other banks:

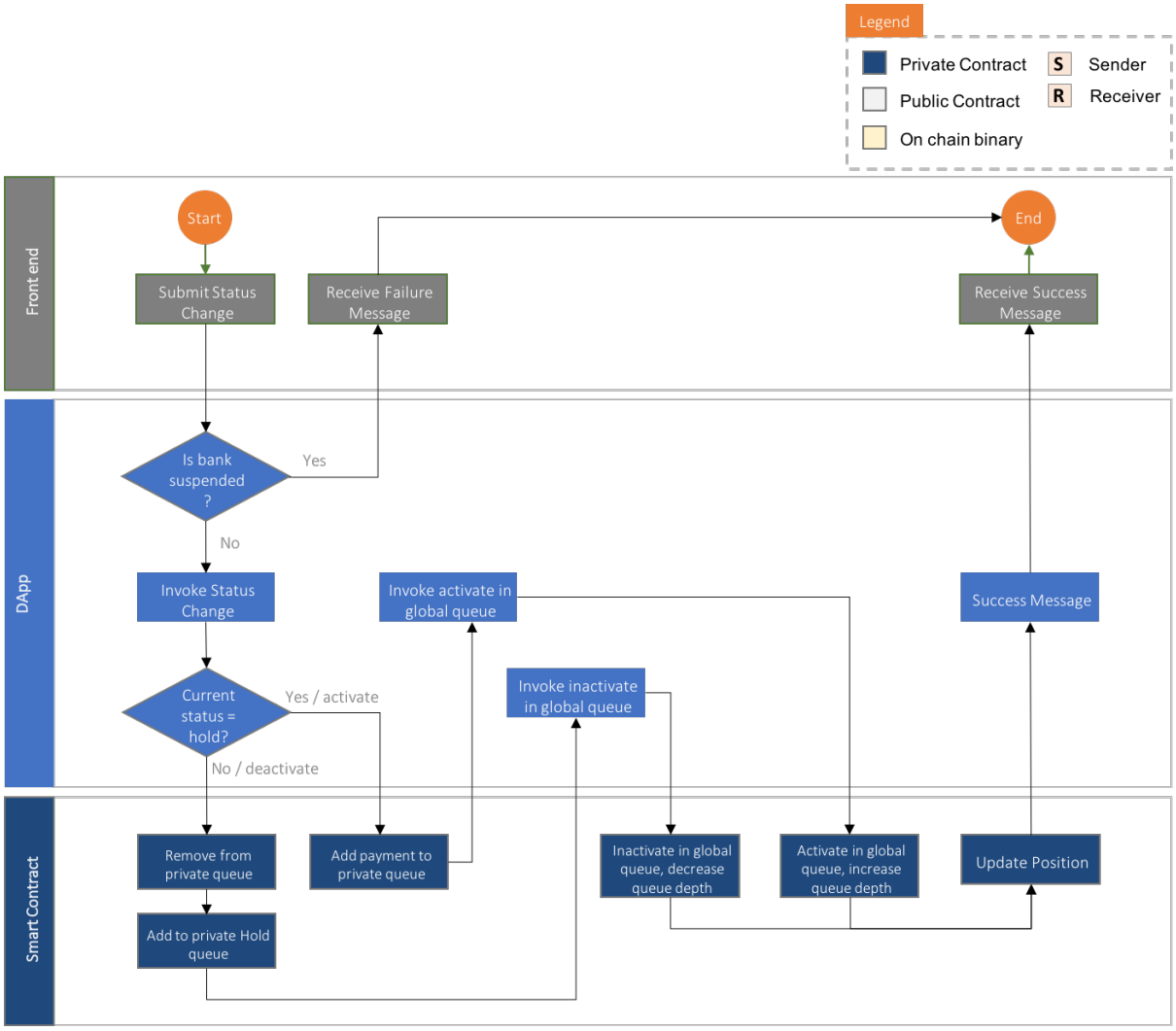


Figure 9: Change Status Process Flow

1. User views their outgoing funds transfers that are queued due to lack of liquidity.
2. A queued funds transfer is selected, status changed to Hold, and submitted from the front end to the DApp.
3. The DApp checks to make sure the bank has not been suspended before invoking the smart contract function to change status.
4. The payment is removed from the private gridlock queue and added to another structure that holds payments on Hold status.
5. The DApp will then inactivate in the global gridlock queue.
6. The position balance is changed to reflect the decrease in outgoing funds transfers.
7. A response is returned to the DApp which will then formulate a message back to the caller.

2.4.2 Change Priority

A bank may change the priority level of an outgoing payment between High and Normal. Changing the priority will shift the queued payment in the current list between the High priority and Normal priority list segments.

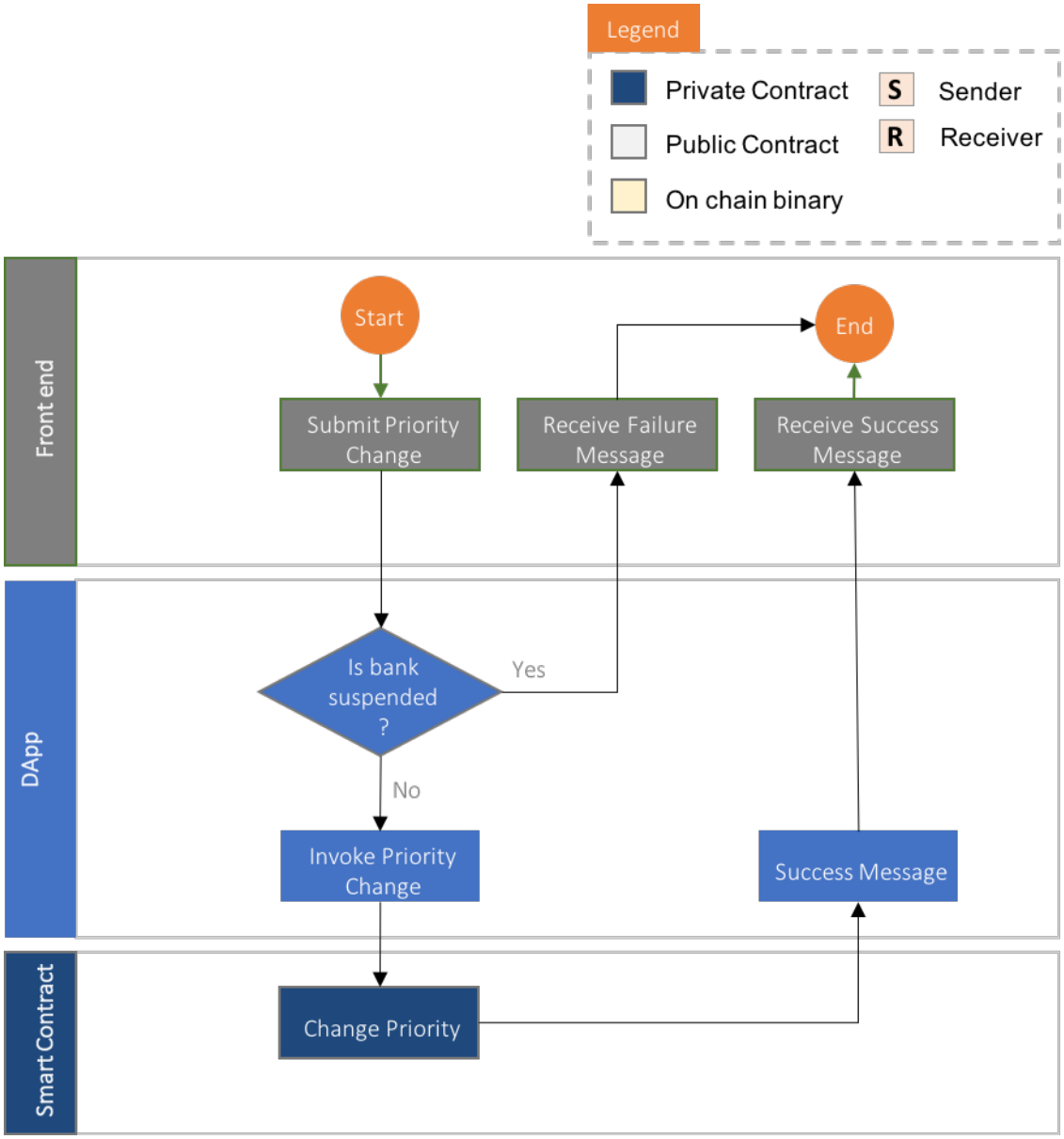


Figure 10: Change Priority Process Flow

1. User views their outgoing funds transfers that are queued due to lack of liquidity.
2. A queued funds transfer is selected, priority changed from normal to high, and submitted from the front end to the DApp.
3. The DApp checks to make sure the bank has not been suspended before invoking the smart contract function to change priority.
4. The funds transfer priority is changed in the appropriate payment structure within the smart contract.
5. A response is returned to the DApp which will then formulate a message back to the caller.

2.4.3 Cancel Payment

A bank may cancel a payment in their outgoing queue if they no longer wish to have the payment instruction. The payment will be removed from both the private and public gridlock queues.

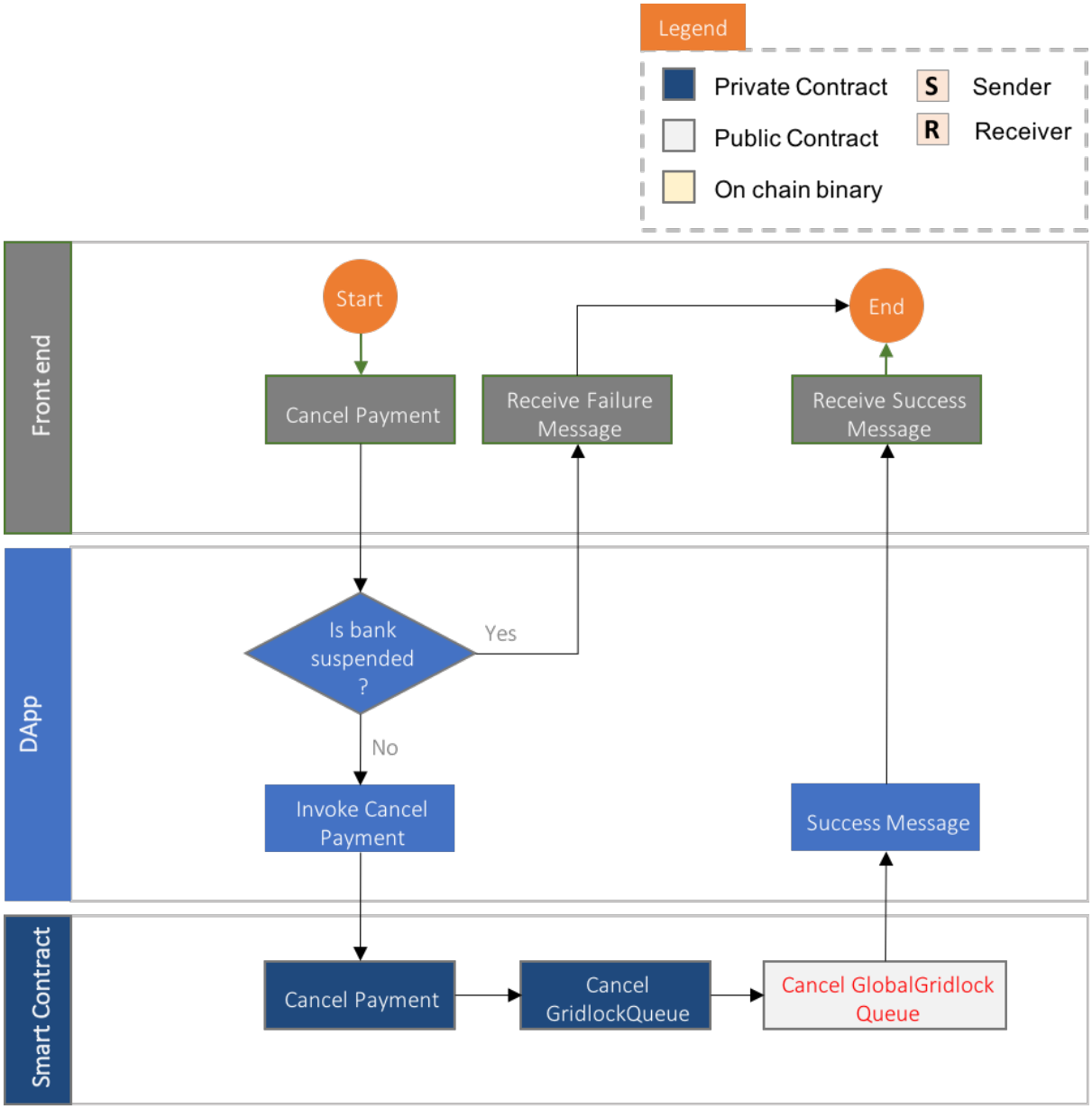


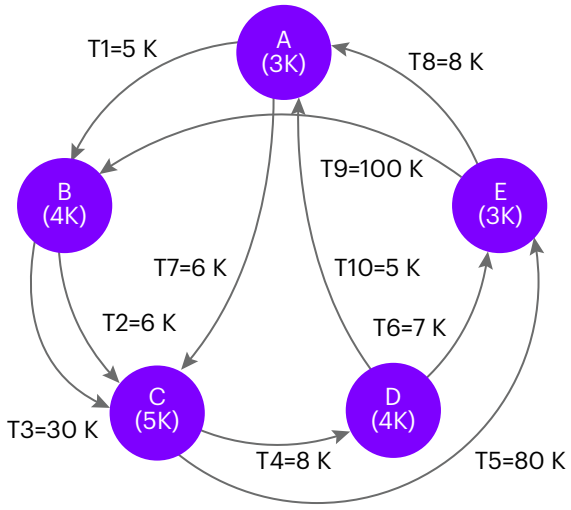
Figure 11: Cancel Payment Process Flow

1. User views their outgoing funds transfers that are queued due to lack of liquidity.
2. A queued funds transfer is selected for cancel and submitted from the front end to the DApp.
3. The DApp checks to make sure the bank has not been suspended before invoking the smart contract function to cancel funds transfer.
4. The payment is removed from the private gridlock queue.
5. The payment is removed from the global gridlock queue.
6. The position balance is updated.
7. A response is returned to the DApp which will then formulate a message back to the caller.

2.5 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 12: Gridlock Scenario

2.5.1 Stages

For the Quorum prototype, the EAF2 algorithm is used for gridlock resolution whilst maintaining balance privacy and meeting the goal of decentralised processing. Quorum workstream's implementation of decentralised gridlock resolution is driven by a cycle of 4 states; Normal, Line Open, Resolving, and Settling. The states are maintained in a smart contract that is synchronised to all nodes. Node behaviour is determined by the current system state. The state dictates how the nodes should handle payment instructions and function during the gridlock resolution process. An event is emitted by the Quorum node each time the state changes. The DApp listens for state change events and orchestrates smart contract execution.

During the ***Normal*** state, which is the default state, banks may transfer funds to each other. Fund transfers are stored as payment instructions in the system, and are either settled immediately (if the sending bank has sufficient liquidity) or are queued. The Quorum node remains in Normal state until the number of payment instructions in the Global Gridlock Queue reaches a predefined threshold. This threshold, which is configurable, serves as an automatic trigger to move the node into the next state - Line Open.

2.5.1.1 Stage 1 – Line Up

The **Line Open** state kicks off the gridlock resolution process.

- Banks that have outgoing or incoming queued payment instructions 'line up' (as in line up in a virtual order) by executing a smart contract function.
- The order in which they execute this function results in the sequence in which each bank will execute the gridlock resolution algorithm.
- Whilst this order is dictated by when each bank executes this smart contract function, and is therefore somewhat arbitrary, i.e. determined by hardware and network latency, it can also be configured to be pre-determined.
- The POC design implements the arbitrary option as the EAF2 algorithm has an inherent bias towards the bank that gets to process their gridlock payments first. Therefore, an arbitrary/random sequence introduces fairness into this process. The next state **Resolving** begins when all banks have lined up or when a timeout is reached

First bank → **Last bank**

Oldest transaction ↓ **Latest transaction**

	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 13: Banks Lined Up

2.5.1.2 Stage 2 – Resolving

In the next state, **Resolving**, each bank executes gridlock resolution according to the EAF-2 algorithm.

- The first bank evaluates its queued payment instructions, moving from newest to oldest, it checks whether each outgoing payment instruction would cause the bank to go into deficit based on the current position.
- Payment instructions that cause deficits are *inactivated*, while payment instructions that do not cause deficits remain *activated*.
- This process is replicated in sequence by each bank as per the sequence established in Line Open state. Banks are not able to reactivate payments.
- The cycle continues as each bank must reprocess their set of payment instructions if an incoming payment is inactivated by the sending bank.
- The process runs in a round robin fashion until all banks submit their set of payment instructions that can be settled without getting into deficit. This initiates the next state – Settling.

Resolution is calculated based on the order of banks and payment instructions. Payment instructions causing deficit will be invalidated.

First bank → **Last bank**

Oldest transaction ↓ **Latest transaction**

			Deficit		Deficit
	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			1 ~~-80~~		~~+80~~
T6				-7	+7
T7	-6		+6		
T8	+8				-8
T9		+100			-100
T10	+5			-5	
Netted balance	5	73	~~-41~~	0	~~-18~~

Figure 14: Bank C inactivates payment

First bank → Last bank

Oldest transaction ↓ Latest transaction

	A (K)	B (K)	C (K)	D (K)	E (K)
					Deficit
Starting balance	3	4	5	4	3
T1	-5	+5			
T2		-6	+6		
T3		-30	+30		
T4			-8	+8	
T5		1	~~-80~~		~~+80~~
T6				-7	+7
T7	-6		+6		
T8	+8				-8
T9		2	~~+100~~		~~-100~~
T10	+5			-5	
Netted balance	5	~~73~~	39	0	~~-98~~

Figure 15: Bank B inactivates payment

First bank → Last bank

Oldest transaction ↓ Latest transaction

	A (K)	B (K)	C (K)	D (K)	E (K)
		Deficit			
Starting balance	3	4	5	4	3
T1	-5	+5			
T2		-6	+6		
T3		3	~~-30~~	~~+30~~	
T4			-8	+8	
T5			1	~~-80~~	~~+80~~
T6				-7	+7
T7	-6		+6		
T8	+8				-8
T9		2	~~+100~~		~~-100~~
T10	+5			-5	
Netted balance	5	~~-27~~	~~-39~~	0	2

Figure 16: Bank A inactivates

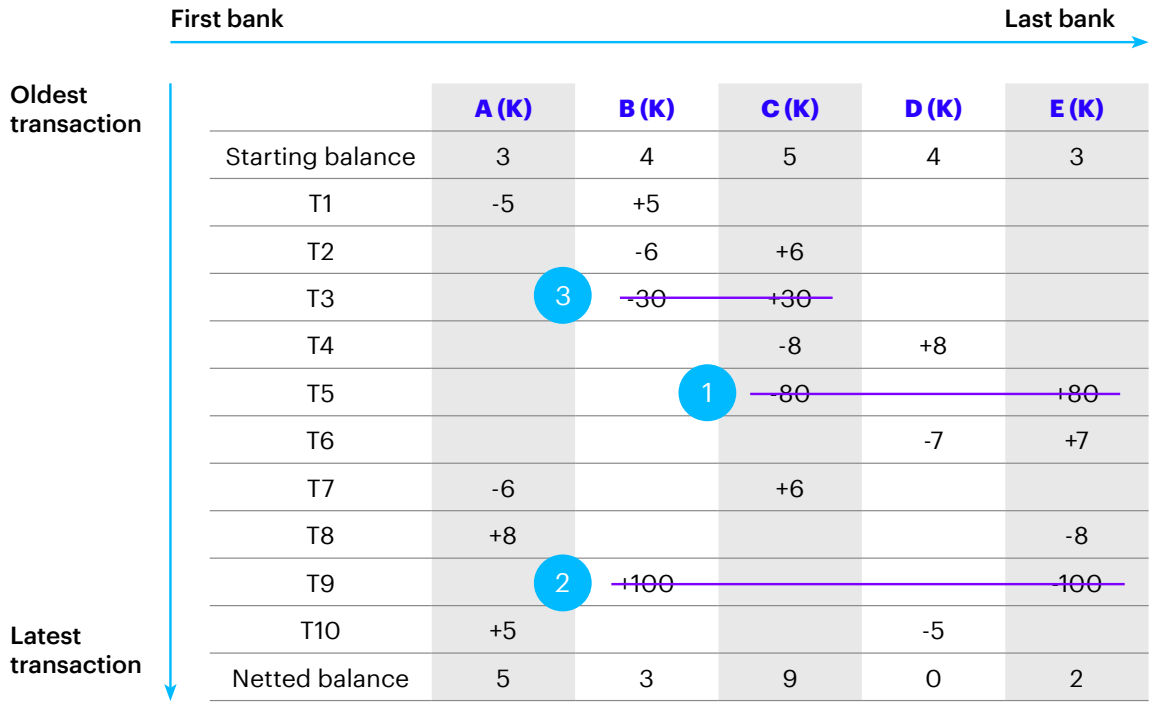


Figure 17: Payment inactivations complete

Example:

- Bank C is the 3rd bank and has a deficit of $41,000. T5 will be invalidated.
- The netted balance for Bank C and Bank E are $39,000 and -$98,0000.
- To remove the deficit in Bank E, T9 will be invalidated as well. When T9 is invalidated, it affects the netted balance of Bank B.
- Thus, T3 is invalidated as well to prevent deficit in Bank B. During the Settling state, all successful transactions are marked as completed.
- T3, T5 and T9 remain unsettled and stay in the queue. They will be picked up during the next gridlock resolution cycle.

2.5.1.3 Stage 3 – Settling

During the final **Settling** state, banks with active payment instructions, post resolve round, begin generating zero-knowledge proofs for their respective incoming and outgoing payments and submit them for validation by the entire network.

- The proofs are validated by all nodes on the network.
- Once validation is complete, the shielded salted balances of the participating banks are updated in the public contract and all the transactions which were part of netting are marked as confirmed according to the gridlock resolution outcome.
- This is performed in a single atomic transaction. When this is complete, the gridlock resolution state returns to the **Normal** state.

2.5.2 Sequence Flow

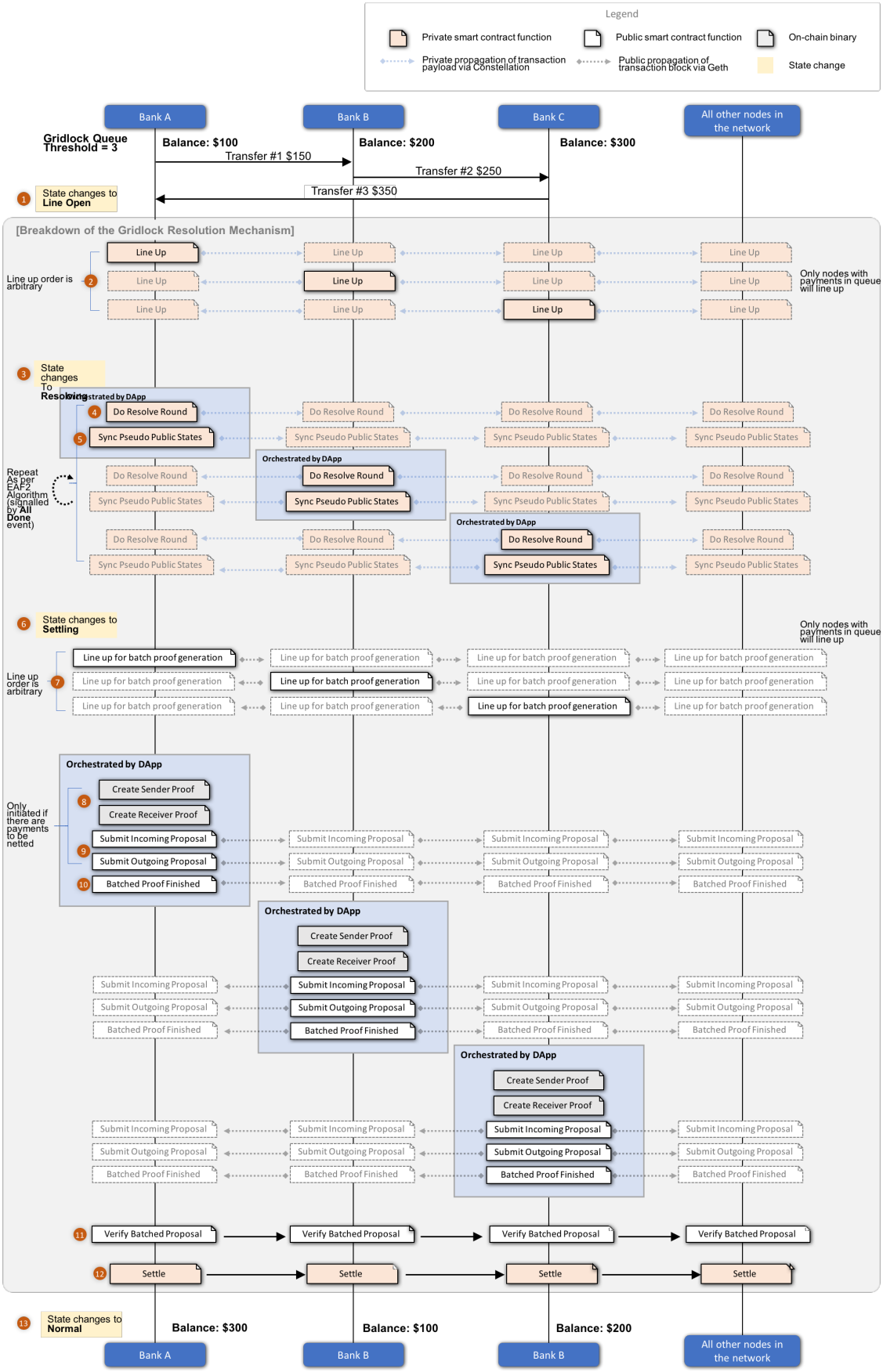


Figure 18: Netting Sequence Diagram

1. Gridlock queue threshold reached, set to 3 in this diagram, a Line Open event is raised and the gridlock resolution state changes to Line Open to let banks line up.
2. Banks have a predefined length of time to join the line up, currently set to 3 seconds, by calling the lineUp method in the smart contract. This registers the bank as being a participant in the gridlock resolution process. The method establishes a sequence so that each bank knows when it should participate in the netting process.
3. Banks will start to execute the resolving round based on the order established in step 2.
4. In the first bank in the sequence inactivates payments until it is no longer in a non-deficit state. It submits a list of transaction hashed with a “active” or “inactive” state. The state of the gridlocked transaction hashes is updated accordingly.
5. After a resolving round, the bank will update the Pseudo Public State to hand off the resolving round to the next bank in line. An event is then raised indicating the next bank in the sequence that should participate. The process continues until all the banks have indicated that they will be in a non-deficit state if only the active transactions are performed as a single atomic transaction or until all the transactions are marked as inactive in which case the queued payments will be in a deadlocked state and additional liquidity will need to be injected into the system.
6. Once the process is complete, the “AllDone” event is raised and is received by the Listener on the DApp and that triggers the proof generation process to start.
7. The DApp calls out to the proof generator on the node and generates the proofs required to form part of the shielded transaction. These are done using the libsnark based proof generator. These proofs are a little different to the standard payment proofs in that each node needs to submit a set of “receiver proofs” for all incoming payments as well as a set of “sender proofs” for all outgoing payments.
8. Each bank submits a public (shielded) transaction including the proofs to the smart contract.
9. Once the batch proposals are submitted, the bank will wait for all banks to submit their proposals.
10. A check is run to determine if there is a match from outgoing hash amounts to ingoing hash amounts. i.e. a payment from Bank B to Bank D needs to be reflected by Bank B as an outgoing payment and Bank D as an incoming payment. This is
11. Once the check is completed, all the proofs are submitted to the contract, the verification process runs.
12. Banks will then verify the batch proofs.
13. The netting process is marked as complete and the hashes of the balances and final private balance are updated in the settling phase.

2.6 Pledge and Redeem

2.6.1 Pledge

Pledge transactions will typically be initiated in the traditional route. Banks will send a request via MEPS+ to MAS for pledging funds to digital currency account. Once MAS approves the request, the physical account in CAS system of MAS is credited with the pledge amount. This activity triggers a request to the blockchain platform for crediting the bank's stash with equivalent amount. The pledge transaction will be initiated form the MAS node. The DApp on MAS node will perform the following to complete the pledge transaction:

1. The DApp first checks if the bank is suspended and if yes, the transaction will be terminated
2. The DApp generates a dynamic salt for the new stash balance of the bank

3. The DApp makes a call to the private contract to decrease the true balance for the bank. This balance position will be visible to the owning bank only. None of the other banks in the network will be able to see this
4. The DApp then submits a shielded balance update request to the public contract. The shielded balance value is set in the public contract for bank and since it's a public transaction the same value will be propagated and visible to all nodes in the network.

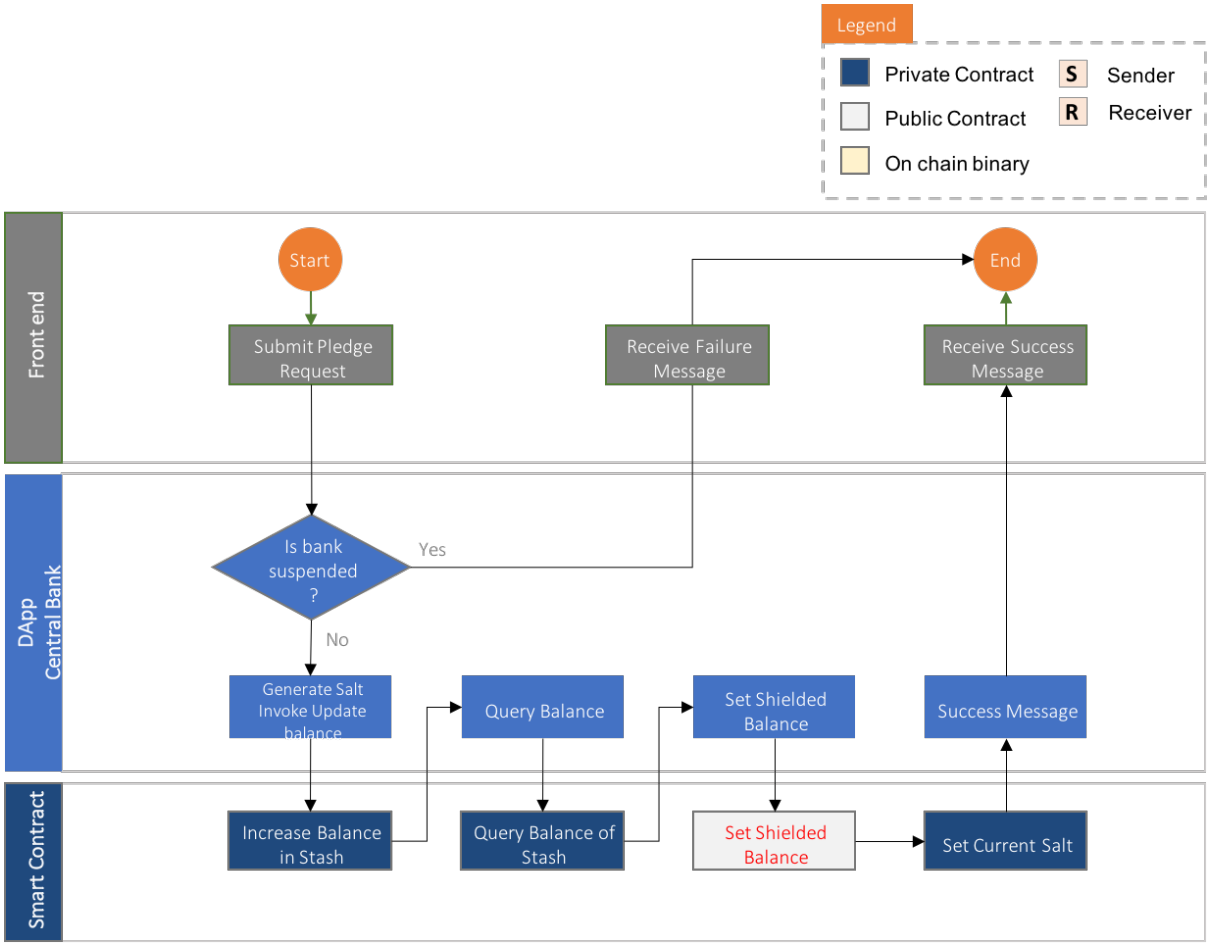


Figure 19: Pledge Process Flow

1. A bank decides on the pledge amount to request to the MAS Central Bank and enters the amount into the front end.
2. A request is submitted to the MAS Central Bank's DApp API.
3. The DApp checks to make sure the bank has not been suspended before invoking the smart contract function to add funds to the bank's stash and update the shielded balance.
4. The new transactional salt that was generated is saved as the current salt as processing completes.
5. A response is returned to the DApp which will then formulate a message back to the caller.

2.6.2 Redeem

- The DApp first checks if the bank is suspended and if yes, the transaction will be terminated

- The bank must have sufficient funds in the DLT to be redeemed.
- The DApp generates a dynamic salt for the new stash balance of the bank
- The DApp makes a call to the private contract to update the true balance for the bank. This balance position will be visible to the owning bank only. None of the other banks in the network will be able to see this
- The DApp then submits a shielded balance update request to the public contract. The shielded balance value is set in the public contract for bank and since it's a public transaction the same value will be propagated and visible to all nodes in the network.

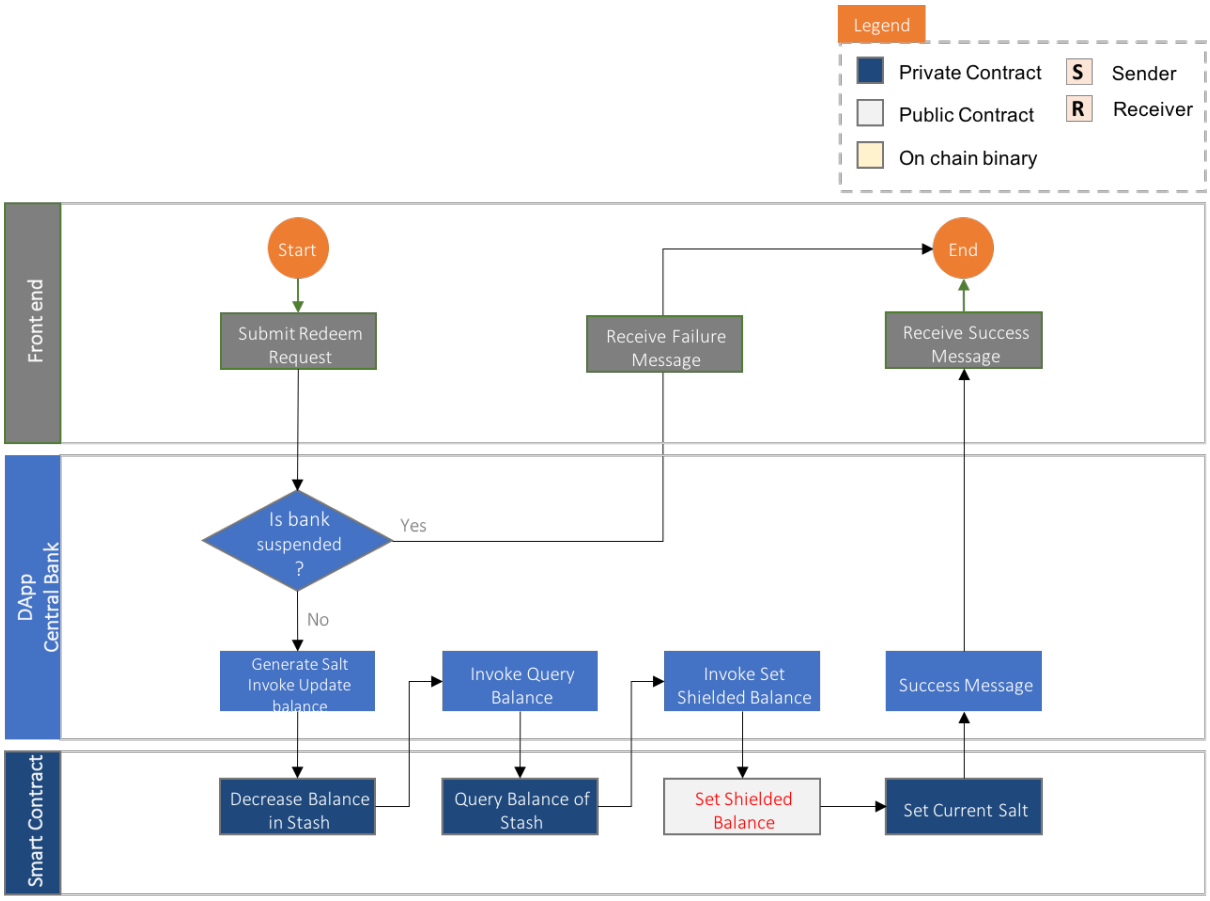


Figure 20: Redeem Process Flow

1. A bank decides on the redemption amount to request to the MAS Central Bank and enters the amount into the front end.
2. A request is submitted to the MAS Central Bank's DApp API.
3. The DApp checks to make sure the bank has not been suspended before invoking the smart contract function to remove funds from the bank's stash and update the shielded balance.
4. The new transactional salt that was generated is saved as the current salt as processing completes.
- A response is returned to the DApp which will then formulate a message back to the caller.

2.7 Balance Enquiry

A bank's DApp will query its own balance via the PaymentAgent contract which in turn queries the Stash contract.

The regulator DApp will connect to each of the bank's Quorum nodes using the bank's default account and query as the owning bank would. To enable MAS node for this query, MAS node should store the banks IP address of the Quorum nodes. The current implementation does not have a password configured for this default account.

As a future consideration, it is possible for each bank to create a special Ethereum account for the regulator, and pass it to the regulator during onboarding. The regulator can then hit the endpoint of the geth rpc of the respective banks, and query the private smart contract with the regulator account. Note that in this setup, MAS will have one regulator account for each bank. Additionally, the regulator accounts provided by banks should have read access only.

2.8 Manage Accounts

2.8.1 Onboarding a new bank

It is quite possible that after the initial network is created, a new participant wants to join the Quorum network. Quorum with RAFT consensus in the latest version supports dynamic membership and allows a node to join the network after the initial network set up is completed.

However, the current Ubin design and development has happened on an older version of Quorum which does not allow a new node to join post initial network creation. Once upgraded to the latest version, it will be possible to add new nodes after the initial network creation.

Ubin design has private contracts which are deployed using “privateFor” for the entire network at the time of network creation. Currently, it is not possible to extend these private contracts to new nodes which have joined the network with dynamic membership feature. This is future consideration for Quorum platform. Considering the above two, in the current design it is not possible to on-board a bank dynamically into the network.

2.8.2 Updating Bank Name

Bank name change is a rare event and this can happen because of some of the following reasons:

- A bank merging with another bank
- A bank opting to change its name because of change of ownership

In either case, the change of bank name is a prolonged process and needs to adhere to all legal and compliance guidelines. Further it needs to be decided how the historical data prior to bank name change will be handled and under which bank name. Traditionally this is handled by migrating certain years of transaction history to a historical data store and running servicing and customer queries from the same. A similar approach can be used for blockchain based applications as well.

Further, at the time of bank name changes there should not be any in-flight pending transactions. This will ensure that data integrity is maintained.

In the current design, the bank name is managed at each Stash level. A new function can be added to update the Stash name. This will start reflecting in future transactions. The past transactions can be queried from the historical off chain database. The approach to update the bank name is for future consideration discussion.

2.8.3 Keys Management

There are two possible approaches for identity management, generating and distributing the public keys and updating the keys in the lifecycle:

- CA-based system using a traditional Certificate of Authority
- Blockchain based identity management

Refer to Appendix item 3 for a detailed explanation of the two options above.

Once a key management system is in place the steps to update the keys can be done with the following steps:

- The bank will create a new Ethereum account (private/public key pairs) using his own nodes.
- The bank will send the new public key to MAS.
- MAS perform necessary checks and other 'KYC' activities for the new public key.
- MAS will perform the update by calling registerStash() in the PaymentAgent contract.

2.8.4 Bank Suspension

The MAS acting as regulator role has the authority to regulate the participants in the network. This includes the ability to suspend and unsuspend banks from the network. Once a bank is suspended, it will not be able to perform certain transactions in the network.

2.8.5 Suspend

When a bank is suspended, the bank will no longer be able to perform certain activities.

Blocked functions:

- Fund transfers - sending and receiving
- Participate in gridlock resolution
- Settle queue
- Pledge
- Redeem
- Cancel fund transfers
- Change status of fund transfers
- Change priority of fund transfers

Available functions:

- View Balance
- View Transaction history
- View incoming and outgoing queues

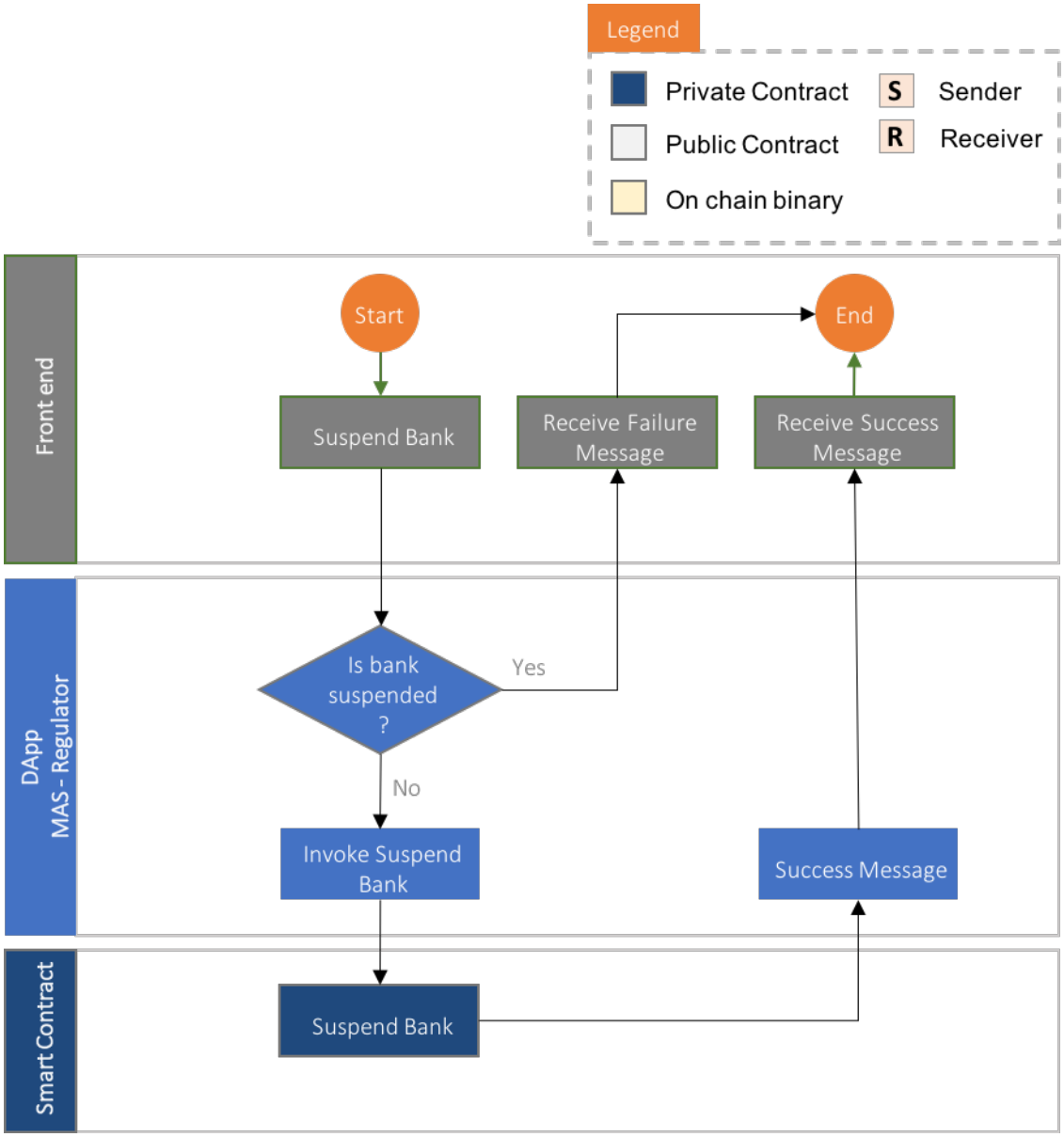


Figure 21: Suspend Bank Process Flow

2.8.6 Unsuspend

A bank can subsequently be unsuspended by the regulator which will restore all the functions that were affected by the suspension.

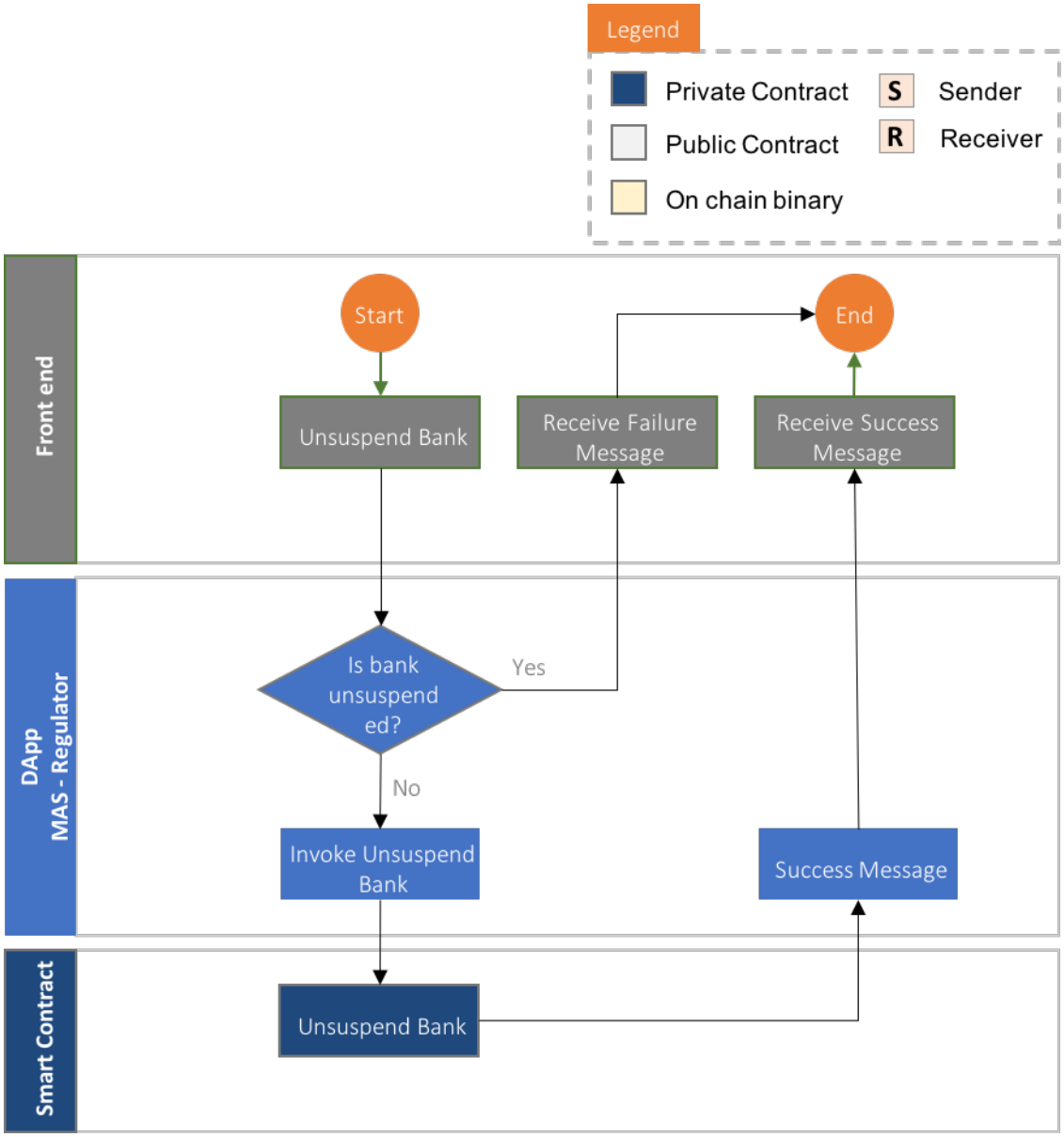


Figure 22: Unsuspend Bank Process Flow

2.9 Versioning

A participant on the DLT network should be able to check the version of the code that is running on any node so that they can check if the node is up to date and is consistent with the other nodes. The latest code version number shall be tracked outside of the system. Anyone can release the latest DLT version to all nodes on the network so that all the participants will have access to the same set of DLT features. All nodes on the network will then be updated to the latest DLT platform version.

3 Technical Specifications

3.1 Transaction Validity

Quorum achieved transaction validity through Zero Knowledge Security Layer (ZSL). The design leverages Quorum's Zero Knowledge Settlement Layer (ZSL), a protocol designed by

the team behind Zcash, that leverages zk-SNARKS – a variant of zero knowledge proofs - to enable the transfer of digital assets on a distributed ledger, without revealing any information about the Sender, Recipient, or quantity of assets that are being transferred, and without requiring any central party to affect the transfer. Zero knowledge proofs provide de-centralized privacy during Liquidity Savings Mechanism execution and payment transfers. Zero knowledge proofs are a critical feature of the design. Zero knowledge proofs prove to the entire network that a particular transaction can be carried out and the initiating bank has enough balance to perform the transaction, without revealing anything about the sender, recipient or balance to anyone else on the network.

It should be noted that in the current design of Ubin, on the public side the network will see the participants of a transaction but will not see any details of the transaction pay load.

The proof design is based on the following work:

- Quadratic Arithmetic Programs: from Zero to Hero - Vitalik Buterin (https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649)
- Introduction to zk-snarks with example - Christian Lundkvist (https://media.consensys.net/introduction-to-zksnarks-with-examples-3283b554fc3b)

3.1.1 Proofs

Proofs are generated off chain by both the sender and receiver of a funds transfer. When the sender of a payment proves that his balance has updated correctly and the receiver of the same payment proves that his balance has updated correctly then the total amount of money available on the blockchain remains consistent.

Hashing is performed by using SHA-256 represented in HEX format.

Basic formula of the proof is A = B + C and the proof will check to make sure the amounts correctly add up. The proof generator will also check to make sure that the balance after is greater than zero ensuring the sender has enough liquidity to perform the funds transfer.

3.1.2 Proof Method

Proof generation is performed off chain by a compiled binary installed on the bank's Quorum node. The DApp layer will invoke method zsl_createABCProof() to generate the proof by submitting the following parameters:

- Hash value 1
- Hash value 2
- Hash value 3
- Salted value 1
- Salted value 2
- Salted value 3

3.1.2.1 Salt

Salt is a randomly generated 16-byte number and added to the right half of the 32-byte amount to be hashed.

Two types of salts are used:

Transaction salt (Tx salt) - randomly generated by the sender for each funds transfer.

Current salt - the last Tx salt used to update the bank's shielded balance.

3.1.2.2 Amount Hash

This is the hashed value of an input amount that is a 32-byte number with the first 16 bytes containing the amount and the later 16bytes containing a salt value.

3.1.2.3 Sender Proof

Start Balance = End Balance + Transaction Amount

Table 1 : Sender Proof Parameters

Parameter	**Tx Salt**	**Current Salt**
End balance hashed	X	
Transaction amount hashed	X	
Start balanced hashed		X
End balance salted	X	
Transaction amount salted	X	X
Start balance salted	X	

3.1.2.4 Receiver Proof

End Balance = Start Balance + Transaction Amount

Table 2: Receiver Proof Parameters

Parameter	**Tx Salt**	**Current Salt**
Start balance hashed		X
Transaction amount hashed	X	
End balance hashed	X	
Start balance salted		X
Transaction amount salted	X	
End balance salted	X	

3.2 Privacy

Quorum supports both public and private transactions within the permissioned network. The public transactions are broadcast to all the nodes within the network and are processed like regular Ethereum transactions. Private transaction details are sent as encrypted blobs directly to the specified recipients by Quorum's privacy service Constellation. This transmission is secure, point-to-point, on a need-to-know basis. The way a private transaction is managed within Quorum is as depicted below:

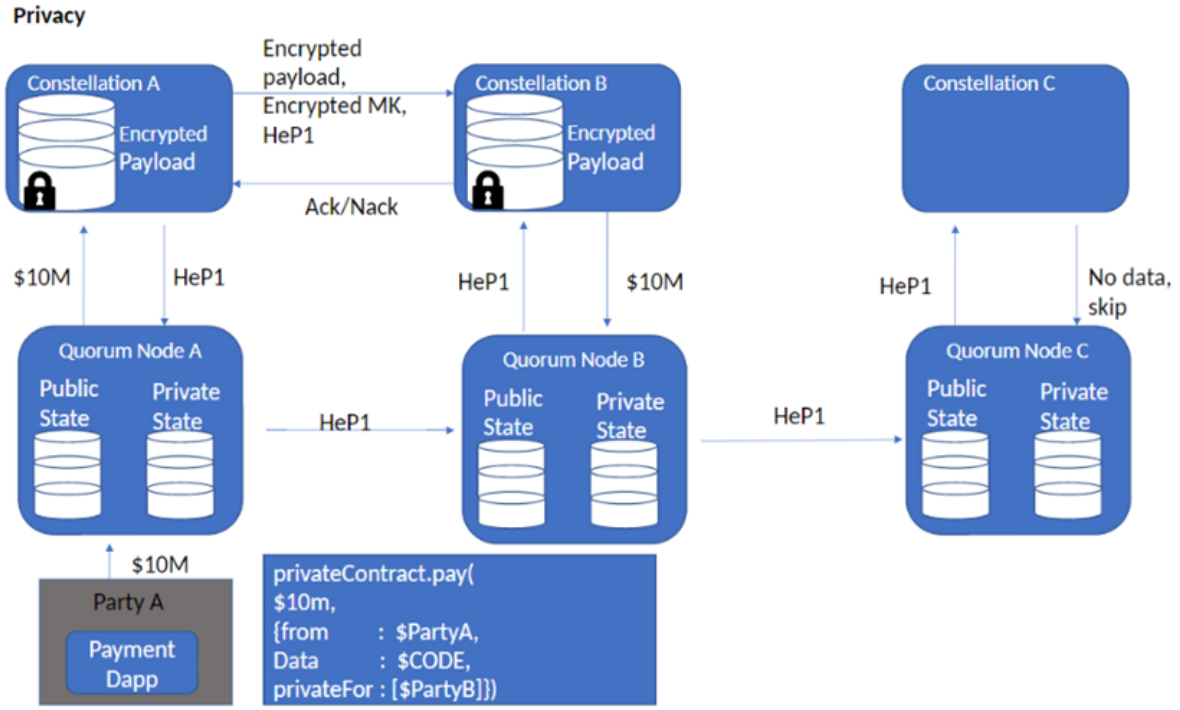


Figure 23: Privacy Components

As seen above, the transaction payload is available only to the participants of the transactions and rest of the network simply sees a hash of the encrypted payload. The key benefit of propagating these hashes to all participants is one of security and resiliency: should a party to a private transaction require validation of the existence of that transaction at some point in the future (i.e. after a data loss event or in case of dispute between the parties to a private transaction), they can confirm this with the rest of the network by comparing to the hashes that the network holds, thereby not needing to trust the information held at the counterparty. Private transactions allow banks to execute payment instructions private for the specified receiving bank only.

In addition to the above, the current design has also incorporated zero knowledge proofs for managing the shielded salted balance of each participating bank. The true balance position of any participant bank is visible to itself only. Any changes in balance movement as part of transaction execution can happen only via submission of proofs by the sender and receiver banks, followed with the verification of these proofs by the entire network. This allows balance validation by the network without knowledge of another party's true balance and thus avoids double spending in a truly decentralised way.

The prototype incorporates the following components for privacy:

- All payment transactions are executed as private transactions. This ensures that the transaction payload is visible only to the sender and receiver banks.
- Payment transfers on the public contract identify the sender and receiver but shield transfer amounts by storing hash values.
- Account balances are kept in private contracts and are only accessible from the account owner's node
- Dynamic salt is used when hashing transaction amounts, starting balance and ending balance, uniquely for each transaction
- Zero knowledge proofs allow transactions being validated publicly without revealing sensitive information.

A sample transaction flow for funds transfer using the privacy model of Constellation and zero knowledge proofs is depicted below:

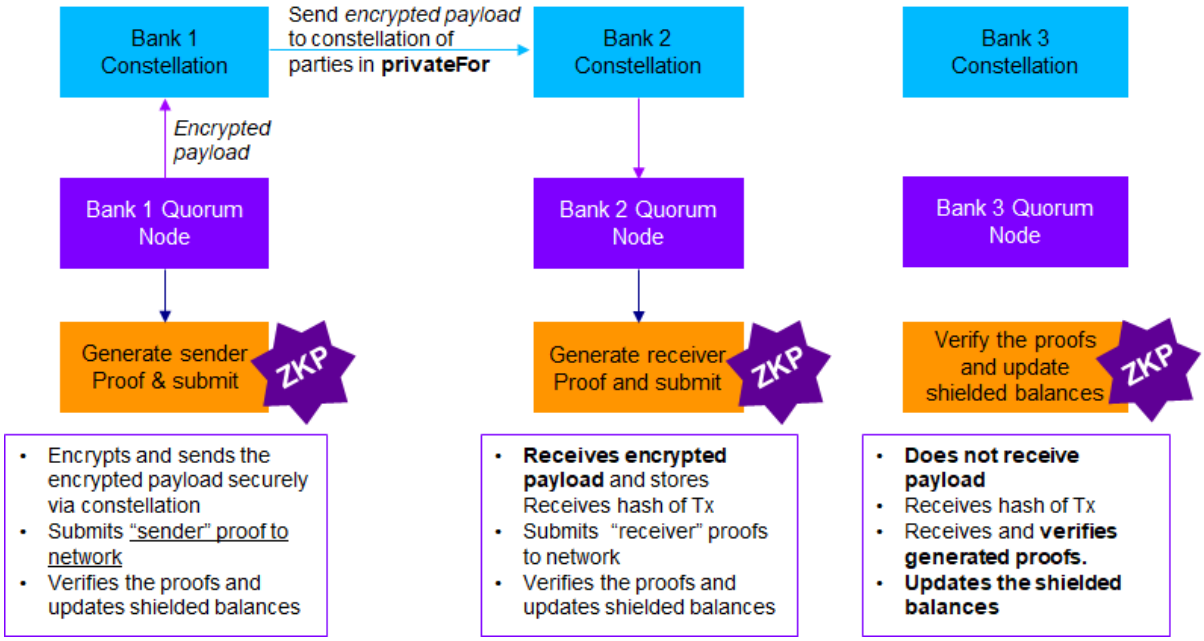


Figure 24: Privacy in Fund Transfer

- Sender will submit a payment to its node. All nodes will receive a hash of the encrypted transaction payload but not the details of the transaction. Only the counter party will receive the encrypted payload. The transaction is marked as unconfirmed at this stage.
- Both the sender and receiver DApps auto generate zero knowledge proofs which prove that the sender bank has sufficient balance to execute the payment and the balance of both sender and receiver will be correctly updated after the payment is executed. These proofs are then submitted to the network for verification.
- All nodes receive and verify the zero knowledge proofs and upon successful verification, the balances of sender and receiver are updated.
- The sender and receiver will then mark the payment as completed.

3.3 Key Technical Matrix

Application	**Version**
Quorum	1.5
Go	1.7.3
Node.js	6.11.3
NPM	3.10.10

4 Interface Specifications

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

5 Key Observations and Findings

5.1 Privacy

Quorum supports both public and private transactions within the permissioned network. The public transactions are broadcast to all the nodes within the network and are processed like regular Ethereum transactions. The private transactions are sent directly to the specified recipients by Quorum's privacy service Constellation as encrypted blobs. It does this by sending the transaction payload only to the involved participants and the rest of the network can only see a hash of the encrypted payload. The key benefit of propagating these hashes to all participants is one of security and resiliency: should a party to a private transaction require validation of the existence of that transaction at some point in the future, they can confirm this with the rest of the network by comparing it to the hashes that the network holds, thereby not needing to trust the information held at the counterparty. Private transactions allow banks to execute payment instructions to the specified receiving bank only. In addition to the above, the current design also incorporates zero knowledge proofs (ZKP) for managing the shielded salted balance of each participating bank. The true balance position of any participant bank is only visible to itself. Any change in balance movement as part of transaction execution can only happen via submission of ZKP by the sender and receiver, followed by verification of these proofs by the entire network. This allows balance validation by the network without knowledge of the true balance and thus avoids double spending in a truly decentralised way.

The prototype also incorporates the following components for privacy:

- Payment transfers on the public contract identify the sender and receiver but shield transfer amounts by storing hash values
- Account balances are kept in private contracts and are only accessible from the account owner's node
- Dynamic salt is used when hashing transaction amounts, starting balance and ending balance, uniquely for each transaction

5.2 Scalability and performance

Quorum Network Manager (QNM) – an open source tool for creating and managing Quorum networks – was used in Ubin Phase 2 to setup a Raft based Quorum network. The tool automates basic network setup tasks and configuration. The current version of Quorum supports dynamic addition of new nodes, however this could not be tested as a lower version of Quorum was used.

It was observed that the current ZKP generation process takes approximately 4 seconds to generate with a total transaction processing time of 5 seconds for a fund transfer. There is currently research and development work underway to improve the performance of ZKP algorithms. These include plans to increase proof generation and validation speed and lower the memory requirements.

EXCEPTIONAL SCENARIO	OBSERVATIONS
Impact of injecting transaction(s) to the network during the gridlock resolution cycle	After a gridlock resolution cycle has begun and is ongoing, new payment instructions submitted will be inserted into the Gridlock queue as 'Inactive' and will not be processed till the current resolution cycle is complete.

5.3 Resiliency

Quorum inherits Ethereum's block propagation mechanism. If any Quorum node goes down and is disconnected from the network for any reason, the rest of the network can still function as normal. However in such a scenario, no transactions with the unavailable node will be allowed. The intrinsic resiliency of Ethereum ensures that the transaction history is automatically synchronised when the disconnected node comes back online.

This demonstrates that high availability is built into the core platform itself. There are other observations in the current Quorum prototype that can be further enhanced. For example, when the Raft leader is chosen at the time of network set up, the Raft leader could be randomly elected for each transaction for a more resilient design. Future Quorum release is expected to include Byzantine fault tolerant consensus mechanism that rotates the leader before every block creation. Also given that the DApp orchestrates payment flow, manages the proof generation and submission and acts as a link between the public and private contracts, it is another area which requires design focus to ensure overall network resiliency.

Exceptional Scenario	Observations
Impact of removing 1 participating bank during gridlock resolution	Gridlock resolution in Quorum is designed to be less dependent on the participants. In a case where one of the nodes goes down during gridlock resolution process, remaining available nodes are able to proceed and complete ongoing gridlock resolution.
Impact of removing MAS during gridlock resolution	Gridlock resolution is not dependent on the participation of the MAS node. The other participants can complete gridlock resolution successfully.

5.4 Finality

This prototype leverages the Raft consensus model where the elected Raft leader commits new blocks to the chain after verifying the block's transactions and all followers update to the latest block in lock-step. Once a block is committed to the chain it cannot be reversed, thereby providing transaction finality. Currently, the Raft leader is elected during network creation. However, when the nodes in the network detect that the Raft leader is down, the network will elect a new Raft leader, allowing new transactions to be processed.

During the execution phase of the gridlock resolution cycle, the netting process is orchestrated by the DApp where each settlement transaction is processed individually. Atomicity of netting is ensured where the shielded balances are only updated once all the relevant proofs are validated. If one proof fails the entire netting round fails.

The execution time taken for ZKP generation may pose a concern on transaction finality, as participating nodes may drop off during the gridlock resolution cycle (Settling state). As per the current design, if a node drops off during Settling state, the related proofs for the netting will not be received by the network and as such the entire settlement will be invalidated. However, with the expected improvement in ZKP algorithm in the near future, this may no longer be a concern.

5.5 Known Issues

- After deadlock in netting cycle is encountered, payments which are in the queue could not be settled/released via settlement even though the bank has sufficient balance.

- If there is any hold payment in queue at the point in time where gridlock resolution start, gridlock resolution could not be completed.
- Concurrent processing of payment that involves the same node is not supported due to limitation in smart contract.
- In the case where proposal verification submitted by both parties do not match, smart contract will emit Proposal Initiated Event which potentially cause the Dapp goes to infinite loop.
- Geth node would crash if any of the parameters passed in the proof generation is incorrect.

6 Appendix

6.1 Approach for regulator to view bank balances

Each bank will create a special Ethereum account for the regulator, and pass it to the regulator during on-boarding. For the regulator to query the balance, it will hit the endpoint of the geth rpc of the respective bank, and query the private smart contract with the regulator account. Note that in this setup, MAS will have one regulator account for each bank.

To minimise security concerns, the MAS regulator account provided by the banks should have read access only.

6.2 Approach to for concurrency limitation

Currently, to complete one bilateral transfer, both sender and receiver need to provide the proof for their balance update proposal. To ensure atomicity, the transfer is only executed when both sender and receiver proofs are validated in the z-contract.

This design imposes concurrency and resiliency limitation:

- To generate proof the receiver needs to have the transaction amount, which is determined by the sender, so if the bank is down, the other banks cannot transfer funds to it.
- Since salted hash is used to shield balance, all banks need to keep track of their current salt. And the salt can only be updated when the transaction is confirmed. And as mentioned above, the sender needs to wait for the receiver to submit its proof. Therefore, shielded transaction needs to be processed sequentially (and you have to wait until one transaction is confirmed to process the next one), which may complicate the DApps' orchestration flow.

To address the above constraint, an alternative design was conceived, this borrows an idea from UTXO (Unspent Transaction Output) based blockchains. Instead of having one single account balance, each bank will have a 'receipt box'. A sender can create a shielded note and put it to receiver's 'receipt box' when the receiver is 'not at home' (node down), and unilaterally update its own balance by submitting the balance update proposal. The cash contained in the receipt will, at this point, become part of the receiver's liquidity legally. Later on when receiver 'comes back to home' (node up again), it can combine the receipt value into its main account balance by submitting an another proposal. By decoupling the sender's proof and receiver's proof in a single transfer, concurrency is achieved.

To manage receipts, the system can be designed where you can spend your receipts directly (almost like in UTXO blockchains), or you can set a rule saying that you can only spend from your main account, so you will need to incorporate the receipts into your main account before you spent them. It's not yet clear which approach is better.

6.3 Identity Management - Generation and distribution of public key

There are two possible approaches for identity management, generating and distributing the public keys and updating the keys in the lifecycle –

1. CA-based system using a traditional Certificate of Authority
2. Blockchain based identity management

6.3.1 CA-based system

In a traditional identity system, a Certificate Authority (CA) is used to provide the following:

The CA creates a certificate (digitally signed statement) that binds a user's identifier (normally an ID number & name) to a public key. Other participants in the identity system would trust the signature from the CA and can thus be confident that a signature from a specific public key belongs to a particular identifier. The certificates from the CA create a Public Key

Infrastructure (PKI) that can be used by services to validate users authenticating with those services.

If the user/institution loses their key, the CA can reissue a certificate with the new key.Often the CA is also responsible for identity verification, i.e. making sure that the actual person/institution holding the public key corresponds to the correct identity. Thus when viewing a certificate from the CA mapping an identifier to a specific public key, one can be sure that the mapping to the public key is correct and that the identifier really identifies the person or institution linked to the public key. The CA also manages a certificate revocation service which can be used to revoke certificates corresponding to lost or compromised keys. The X509 certificate standard was created in 1988 and together with the XAdES signature standard were originally created for a world that was primarily off-line. Someone should be able to verify your identity and/or signature without access to the internet and/or private networks.

While the eIDAS European Regulation on Digital Signatures only was made law in the EU in 2014, the origins together with XAdES are at least 15 years old and based on the idea that signatures should be verifiable offline.

A key part of this is the concept of the Trust Service Provider. This is a licensed third party who acts as a CA, verifies digital signatures and verifies certificates, timestamps, signatures and links to identities.

This off-line first technology together with its reliance on Trust Service Providers for even basic interactions adds time, cost and friction in the process of validating identities.

6.3.2 Blockchain-based systems

In a smart contract capable blockchain like Ethereum, some of the CA functionality can be automated and replaced using smart contracts. This involves separating out the following:

- the abstract mapping from identifier to key
- the identity verification

The mapping from identifier to keys can be fully automated using smart contracts in a decentralized way. When the institution is on boarded they go through a process where:

- A public/private key pair is generated (in a secure environment like a Hardware Secure Module (HSM)
- A transaction is then sent to a “factory” smart contract generating an (essentially random) identifier for the institution along with a mapping from this identifier to the institution's public key. Through this mechanism the blockchain becomes a decentralized and automated PKI system

If the institution's private key is ever compromised, smart contracts also enable a recovery mechanism by way of remapping the identifier to a different public key.

This can be done in several ways, one of the simplest being a multi-signature mechanism.

It works as follows:

- During (or after) on-boarding, the user selects one or more *recovery delegates*. These delegates are themselves recorded in the user's smart contracts as having the power to replace the user's key. For an RTGS payment system this would typically be the regulator or some other trusted third party.
- If an institution's private key is compromised, a new public/private key pair is generated. The institution would then approach their recovery delegates and ask them to send a blockchain transaction registering the new key.

When a threshold of delegates has sent such transactions the user's identifier will now point to the updated key. Using the above onboarding and recovery systems the first two elements of the certificate authority flow can be automated and done completely on the blockchain. The benefits of this is that a blockchain can provide a PKI with a *larger trust boundary* than a traditional CA, since the users of the system do not need to trust one specific entity, but only need to trust the general integrity of the blockchain platform and the smart contracts on it.

In addition, the job of the Trust Service Provider is also done by the blockchain itself. Blockchain transactions are by their nature verified, timestamped and safely archived. Off chain signatures can be verified using the blockchain as well and, if needed, can be timestamped via the blockchain.

A blockchain-based PKI requires less infrastructure than a centralized PKI (since the blockchain is in place anyway) and is less vulnerable to attack. If an attack does success the damage is localized to one or a small number of identities. This localization makes it less attractive to marshal resources for an attack since the granular target is far less valuable than if one can take control of an entire centralized PKI.

The third step - identity verification - is a step that does require a trusted third party to perform. In order to carry out the verification of an identity, a trusted third party would perform whatever checks they deem necessary (typically part of the onboarding process) and then create and sign a verifiable claim with the identifier of the trusted third party as the issuer and the identifier of the new institution as the subject. The data would specify attributes about the institution such as name, branch, address etc. Separating out the automatic mapping from identifier to keys from identity verification leads to a very flexible system.

6.4 Salt life cycle

The last transactional salt that corresponds with the current shielded balance needs to be known to produce the proof accompanying a balance update proposal. The hash preimage of your shielded balance is an unsigned integer representation of your stash balance concatenated by the salt. The salt is stored on chain as the DApp was designed to be memoryless.

Here are the scenarios where the salt is updated:

1. Pledge / redeem: this is considered as a trusted action, so no proof is used. MAS as central bank will generate a new salt off-chain, and set it directly to the corresponding node.

2. Normal transfer: the salt is set in the confirmPmt function

3. Netting: this is the case where the implementation has room for improvement. In netting a batch of chained proposals to the SGDz contract is submitted. As the last proposal's end balance will be the new shielded balance, you will need to use the salt in the last proposal as your new current salt.

6.5 Transaction History to be offloaded onto external storage / SQL

It is desirable that the on chain transaction history be offloaded to off chain databases. The off chain data then can be queried for any historical data searches or for performing data analytics. Quorum supports historical transaction data to be offloaded to standard RDBMS like databases. A reference implementation flow using Apache Kafka and Storm is shown below:

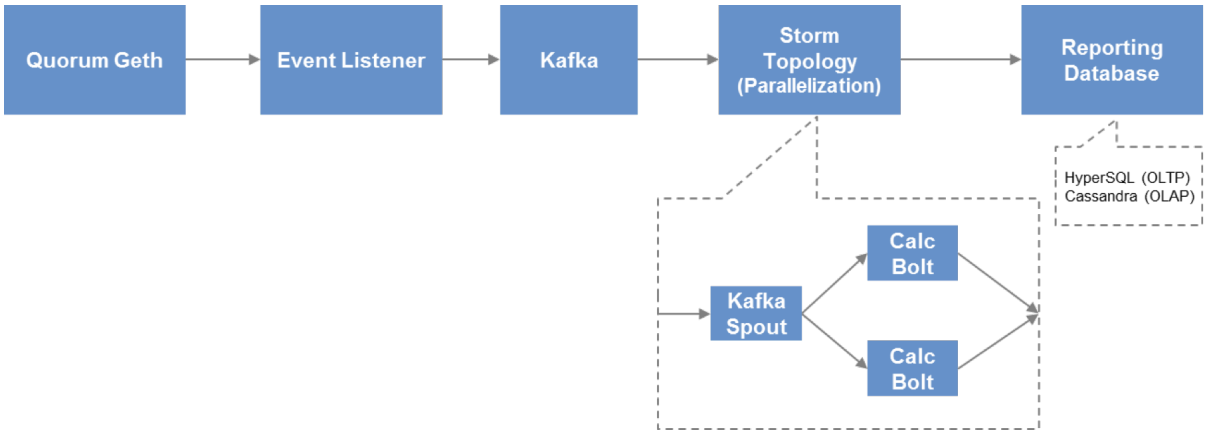


Figure 25: Off Chain Transaction Storage Flow

Events triggered by the contract are picked up by the Event Listener which records the transaction into the Reporting database. Database update performance can be increased by parallelizing the event processing via Kafka/Storm.

BlockApps team recently open sourced Quorum-RDBMS reference implementation. The details of the same can be found here: https://github.com/blockapps/quorum-rdbms-implementation. This deployment allows Quorum on chain data to be offloaded to PostgreSQL database. The reference architecture for this implementation is as below:

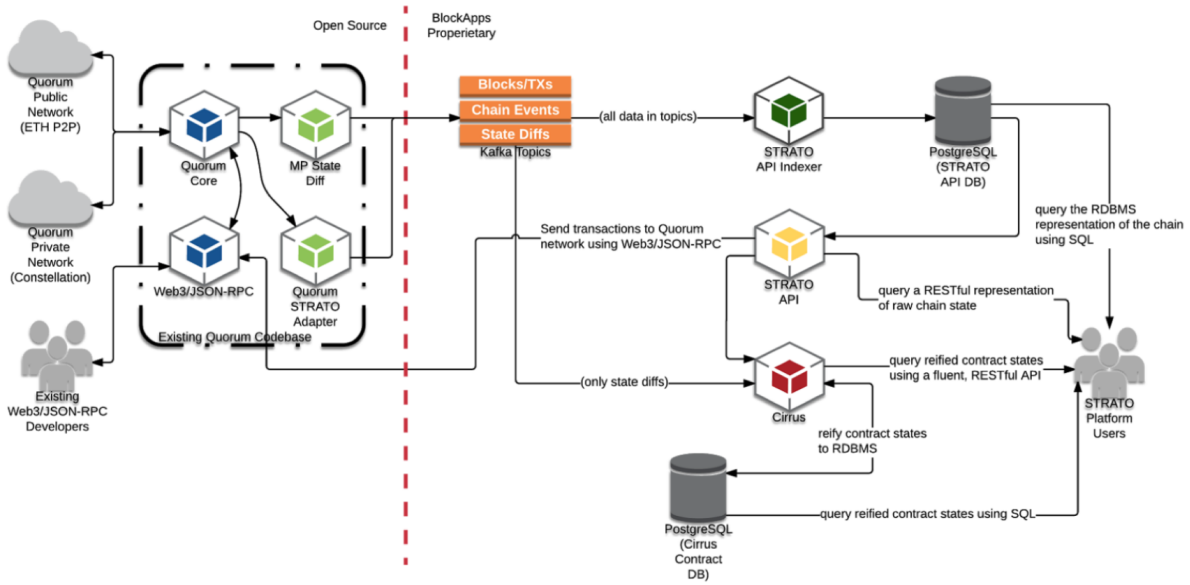


Figure 26: Quorum-RDBMS Reference Architecture

6.6 Workaround for processing transactions during LSM

In the current design, when LSM starts, new transactions will be auto added into the grid lock queue with inactive as the status i.e. will not processed even if there is enough liquidity as LSM is in progress.

A potential design: Split the stash balance into two parts, LSM reserve and normal. When not doing LSM, the smart still operate in the same way as of now, but to support transaction processing during LSM the banks needs to make sure that they met the LSM reserve at all

times. During gridlocked resolution, instead of trying only to make each bank not in deficit, the aim changes to make sure that the ending position is larger than LSM reserve. By doing that, transactions can still be processed with amounts less than LSM reserve.

Example: A bank has LSM reserve = 10, Total liquidity = 20. During gridlock resolution. During LSM the bank will inactivate the payments until the ending position is larger than 10. And it can send out transactions with LSM reserve, i.e. it can send out transactions of total amount less than 10 during LSM.

6.7 Contract Registry

During the lifetime of a project, contracts might need to be updated. Updates to contracts might arise from

- A bug was that was found
- A better algorithm that needs to be used
- Additional functionality required in a contract (e.g. adding Delivery vs Payment DVP hooks to a currency contract)

This section deals with introducing several best practices for managing contracts. This section includes an example contract registry that keeps track of the latest version of a contract, a relay/proxy contract that forwards data and calls to another contract as well as working code (and tests) for separating state and logic.

6.7.1 Upgrading Contracts

Designing an effective upgrade system for smart contracts is an area of active research, and all of the complications in this document cannot be covered. However, there are two basic approaches that are most commonly used.

The simpler of the two is to have a registry contract that holds the address of the latest version of the contract. A more seamless approach is to have a contract that forwards calls and data onto the latest version of the contract. Whatever the technique, it's important to have modularization and good separation between components, so that code changes do not break functionality, orphan data, or require substantial costs to port.

In particular, it is usually beneficial to separate complex logic from data storage, so that there is no need to recreate all of the data in order to change the functionality. It's also critical to have a secure way for parties to decide to upgrade the code. Depending on the contract, code changes may need to be approved by a single trusted party, a group of members, or a vote of the full set of stakeholders.

Example 1: Use a registry contract to store latest version of a contract

In this example, the calls are not forwarded, so users should fetch the current address each time before interacting with it.

```
contract SomeRegister {
    address backendContract;
    address[] previousBackends;
    address owner;

    function SomeRegister() {
        owner = msg.sender;
    }
```

```
    modifier onlyOwner() {
        if (msg.sender != owner) {
            throw;
        }
        _
    }

    function changeBackend(address newBackend) public
    onlyOwner()
    returns (bool)
    {
        if(newBackend != backendContract) {
            previousBackends.push(backendContract);
            backendContract = newBackend;
            return true;
        }

        return false;
    }
}
```

There are two disadvantages to this approach:

Users must always look up the current address, and anyone who fails to do so risks using an old version of the contract

One will need to think carefully about how to deal with the contract data, when the contract is replaced

Example 2: Use a DELEGATECALL to forward data and calls

The alternate approach is to have a contract forward calls and data to the latest version of the contract:

```
contract Relay {
    address public currentVersion;
    address public owner;

    modifier onlyOwner() {
        if (msg.sender != owner) {
            throw;
        }
```

```
        _
    }

    function Relay(address initAddr) {
        currentVersion = initAddr;
         owner = msg.sender; // this owner may be another contract with multisig, not a single
contract owner
    }

    function changeContract(address newVersion) public
    onlyOwner()
    {
        currentVersion = newVersion;
    }

    function() {
        if(!currentVersion.delegatecall(msg.data)) throw;
    }
}
```

This approach avoids the previous problems, but has problems of its own. One must be extremely careful with how the data is stored in the contract. If the new contract has a different storage layout than the older one, then the data may be corrupted.

Additionally, this simple version of the pattern cannot return values from functions, only forward them, which limits its applicability. (More complex implementations attempt to solve this with in-line assembly code and a registry of return sizes.)

Separating state and logic

This section covers an example of 2 smart contracts, one keeping state (specifically balances) and one that deals with the logic of how the state may be manipulated. The state contract keeps balances (the state) and allows manipulation of its state via get and set functions. The getfunction is usable by anyone, however the set function is restricted to be used by only the contractManager, which is the deployer of the contract. Should the logic contract need to change, the state contract has a setLogicContractAddress function that can be invoked by the contractManager to update the address of the logic contract that can use the set method.

State contract:

```
pragma solidity ^0.4.11;

contract state {

  address public contractMaintainer;
```

```
  address public logicContract;

  mapping (address => uint256) public balanceOf;

  modifier onlyBy(address _account){
    require(msg.sender == _account);
    _;
  }

  function state(){
    contractMaintainer = msg.sender;
  }

  function setLogicContractAddress(address _logicContract)
    onlyBy(contractMaintainer)
    returns (bool success){
    logicContract = _logicContract;
    return true;
  }

  function get(address _get) returns (uint256 balance){
    return balanceOf[_get];
  }

  function set(address _set, uint256 _value)
    onlyBy(logicContract)
    returns (bool success){
    balanceOf[_set] = _value;
    return true;
  }
}
```

Logic contract

Below is the state contract. Its constructor takes in the address of the state contract to use. The version of the logic contract shown below has a reference to the state contract's get and setmethods, along with only issueNewTo and balanceOf methods.

Adding a transfer method would require deploying a new logic contract. Since the state has been separated into a separate contract, this is not a problem. It could easily add a transfer method the below contract, deploy it, and then change the logicContract address of the state

contract and then start using the new logic contract. In conjunction with this, using a relay or contract registry could allow this upgrade to happen seamlessly.

```
pragma solidity ^0.4.11;


contract state {
  function get(address _address) returns (uint256 balance);
  function set(address _address, uint256 _value);
}


contract logic {


  uint256 public initialSupply;


  address public contractMaintainer;
  address public stateContractAddress;


  modifier onlyBy(address _account){
    require(msg.sender == _account);
    _;
  }


  function logic(address _stateContractAddress){
    contractMaintainer = msg.sender;
    stateContractAddress = _stateContractAddress;
  }


  function issueNewTo(address _address, uint256 _value) onlyBy(contractMaintainer){
    state stateContract = state(stateContractAddress);
    var balance = stateContract.get(_address);
    stateContract.set(_address, _value+balance);
  }


  function balanceOf(address _address) returns (uint256 balance){
    state stateContract = state(stateContractAddress);
    return stateContract.get(_address);
  }
}
```