1 Introduction

2 Approach

2.1 Keys

Each device has: user id + password
Server login is: hash1(user id), hash1(password)
Symmetric Crypto keys is: hash2(user id — password)
Server has finite length queue of entries + max_entry_identifier + server login key

2.2 Entry layout

Each entry has:

1. Sequence identifier
2. Random IV (if needed by crypto algorithm)
3. Encrypted payload

Payload has:

1. Sequence identifier
2. Machine id (most probably something like a 64-bit random number that is self-generated by client)
3. HMAC of previous slot
4. Data entries
5. HMAC of current slot

A data entry can be one of these:

1. A transaction:
    - Contains a sequence number, a set of key value pair updates and a guard condition that can be evaluated.
    - Must have the same arbitrator for all its key value pair updates and reads within the guard condition

2. A Commit
   Commits a transaction into the block chain. Until a transaction is committed, no client can be sure if that transaction's key value updates will be used to update the state of the system. Once an arbitrator commits a transaction then that transaction becomes a permanent state change in the system. Transactions should be committed and aborted in order of their sequence numbers.

3. An Abort
   An abort is used to show that a transactions key value update should not be used in the state change of the system. This occurs when the guard of a transaction evaluates to false meaning that the conditions under-which this transaction should be committed no longer exists in the system (another transaction could have been committed first that would have changed the system in a way that makes the current transaction invalid).

4. New Key:
   This creates a new key and assignes an arbitrator to that key. Only the first new key message for a given key is valid. Once a new key message is inserted into the block chain it is never removed and no other new key entries for the same key name can be inserted into the block chain.

5. Slot sequence entry: Machine id + last message identifier
   The purpose of this is to keep the record of the last slot from a certain client if a client's update has to expunge that other client's last entry from the queue. This is kept in the slot until the entry owner inserts a newer update into the queue.

6. Queue state entry: Includes queue size
   The purpose of this is for the client to tell if the server lies about the number of slots in the queue, e.g. if there are 2 queue state entry in the queue, e.g. 50 and 70, the client knows that when it sees 50, it should expect at most 50 slots in the queue and after it sees 70, it should expect 50 slots before that queue state entry slot 50 and at most 70 slots. The queue state entry slot 70 is counted as slot number 51 in the queue.

7. Collision resolution entry: message identifier + machine id of a collision winner

The purpose of this is to keep keep track of the winner of all the collisions until all clients have seen the particular entry.

2.3 Live status

Live status of entries:

1. Transaction is live if it has not been committed or aborted yet.
2. Abort is live until the machine ID that created the transaction that is being aborted inserts into the block chain a message with a sequence number greater than the abort (that client sees the abort).
3. Commit is dead if for all key value updates in the commit there is a commit with the same key value update that is newer (larger sequence number). The committing client (arbitrator) will see those newer commits since it is the one that generates them.
4. New Key messages are always kept alive. Keys can not be deleted. Deleted keys will cause arbitration to fail if a key is deleted then reassigned to a new client device for arbitration.
5. Slot sequence number (of either a message version data or user-level data) is dead if there is a newer slot from the same machine.
6. Queue state entry is dead if there is a newer queue state entry. In the case of queue state entries 50 and 70, this means that queue state entry 50 is dead and 70 is live. However, not until the number of slots reaches 70 that queue state entry 50 will be expunged from the queue. Further all entries before the 50 entry will also not be expunged until the queue size has reached 70
7. Collision resolution entry is dead if this entry has been seen by all clients after a collision happens.

When data is at the end of the queue ready to expunge, if:

1. If any entry is not dead it must be reinserted into the queue.
2. If the slot sequence number is not dead, then a message sequence entry must be inserted.

Validation procedure on client:

1. Decrypt each new slot in order.
2. For each slot: (a) check its HMAC, and (b) check that the previous entry HMAC field matches the previous entry (in case of a gap do not check for slots on gap margins).
3. That no slots are slots we have seen before (server trying to pass old slots).
4. For all other machines, check that the latest sequence number is at least as large (never goes backwards).
5. That the queue has a current queue state entry.
6. That the number of entries received is consistent with the size specified in the queue state entry and/or the queue is growing in size.

2.4 Resizing Queue

Client can make a request to resize the queue. This is done as a write that combines: (a) a slot with the message, and (b) a request to the server. The queue can only be expanded, never contracted; attempting to decrease the size of the queue will cause future clients to throw an error.

2.5 The Arbitrator

Each key has an arbitrator that makes the final decision when it comes to whether a specific transaction containing that key updates the state of the system or is aborted. This ensures that clients can make offline updates and then push those updates to the server at a later time. The arbitrator then tries to merge those updates and if possible will commit them into the current working state of the system. If not possible then the arbitrator will abort that transaction. The arbitrator arbitrates on transactions in order of transaction sequence number.

3 Server Algorithm

$s \in SN$ is a sequence number
$sv \in SV$ is a slot's value
$slot_s = \langle s, sv \rangle \in SL \subseteq SN \times SV$

State

SL = set of live slots on server

max = maximum number of slots (input only for resize message)

n = number of slots

Helper Function

$MaxSlot(SL_s) = \langle s, sv\rangle \mid \langle s, sv\rangle \in SL_s \wedge \forall\langle s_s, sv_s\rangle \in SL_s, s \geq s_s$

$MinSlot(SL_s) = \langle s, sv\rangle \mid \langle s, sv\rangle \in SL_s \wedge \forall\langle s_s, sv_s\rangle \in SL_s, s \leq s_s$

$SeqN(slot_s = \langle s, sv\rangle) = s$

$SlotVal(slot_s = \langle s, sv\rangle) = sv$

Get Slot:

Returns to the client the slots that have a sequence number that is greater than or equal to the sequence number that is in the requese.

```
1: function GetSlot($s_g$)
2:     return $\{\langle s, sv\rangle \in SL \mid s \geq s_g\}$
3: end function
```

Get Slot:
Puts a slot in the server memory if the slot has the correct sequence number. Also resizes the server memory if needed.

```
 1: function PutSlot($s_p, sv_p, max'$)
 2:     if ($max' \neq \emptyset$) then                          ▷ Resize
 3:         $max \leftarrow max'$
 4:     end if
 5:     $\langle s_n, sv_n \rangle \leftarrow MaxSlot(SL)$                  ▷ Last sv
 6:     if ($s_p = s_n + 1$) then
 7:         if $n = max$ then
 8:             $\langle s_m, sv_m \rangle \leftarrow MinSlot(SL)$          ▷ First sv
 9:             $SL \leftarrow SL - \{\langle s_m, sv_m \rangle\}$
10:         else                                          ▷ $n < max$
11:             $n \leftarrow n + 1$
12:         end if
13:         $SL \leftarrow SL \cup \{\langle s_p, sv_p \rangle\}$
14:         return ($true, \emptyset$)
15:     else
16:         return ($false, \{\langle s, sv \rangle \in SL \mid s \geq s_p\}$)
17:     end if
18: end function
```

4 Client

4.1 Client Notation Conventions

k is key of entry
v is value of entry
$size$ is a size (target size of the current block chain)
kv is a key-value pair $\langle k, v \rangle$
KV is a set of kv
id is a machine ID
seq is a sequence number
$hmac_p$ is the HMAC value of the previous slot
$hmac_c$ is the HMAC value of the current slot
$Guard$ is a set of$\langle k, v,$logical operator$\rangle$ which can be evaluated to a boolean

$trans$ is a transaction entry , $\langle seq, id, KV, Guard \rangle$
$lastmsg$ is a last message entry, $\langle seq, id \rangle$

$qstate$ is a queue state entry, $\langle size \rangle$
$colres$ is a collision resolution entry, $\langle id, seq_{old}, seq_{new}, true \vee false \rangle$
$newkey$ is a new key entry, $\langle k, id \rangle$, id is ID of arbitrator
$commit$ is a commit transaction entry, $\langle seq_{trans}, KV \rangle$, id is id of arbitrator
$abort$ is an abort transaction entry, $\langle seq_{trans}, id_{trans} \rangle$

de is a data entry that can one of: $trans$, $lastmsg$, $qstate$, $colres$, $newkey$, $commit$, $abort$
DE is a set of all data entries, possibly of different types, in a single message, set of de

$slotDat = \langle seq, id, DE, hmac_p, hmac_c \rangle$
$slot = \langle seq, Encrpt(slotDat) \rangle$

4.2 Client State

4.2.1 Constants

$LOCAL\_ID$ = machine ID of this client.
$RESIZE\_THRESH\_PERCENT$ = percent of slots that need to have live data to trigger a resize.
$RESIZE\_PERCENT$ = percent that we should grow the block chain to.
$DATA\_ENTRY\_SET\_MAX\_SIZE$ = max size that a data entry set can have (in bytes).
$DEAD\_SLOT\_COUNT$ = number of slots to keep dead if possible at the end of the block chain.
MAX_RESCUE_SKIPS = number of skips that are allowed when saving data entries.

4.2.2 Primitive Variables

max_size = max size of the block chain

4.2.3 Sets and Lists

$PendingTransQueue$ = Queue of pending transactions that need to be pushed to the block chain, $\langle PendingTrans \rangle$
$PendingTrans = \langle KV, Guard \rangle = \langle$set of key value pairs, set of guard

conditions$\rangle$.
$Arbitrator$ = set of $\langle k, id\rangle$ containing the key and its arbitrating device.
$LastSlot$ = set of $\langle id, seq\rangle$ containing the machine ID and the largest sequence number from that machine ID.
$LocalSlots$ = set of slots that are in the clients local buffer (initially $\emptyset$), data is decrypted.
$RejectedSlotList$ = ordered list of the sequence numbers of slots that this client tried to insert but were rejected.
$CommittedKV$ = set of committed key value pairs (initially $\emptyset$).
$SpeculatedKV$ = set of speculated key value pairs (initially $\emptyset$).

4.3 Helper Functions

The following helper functions are needed:

$MaxSlot(SL_s) = \langle s, sv\rangle \mid \langle s, sv\rangle \in SL_s \wedge \forall \langle s_s, sv_s\rangle \in SL_s, s \geq s_s$
$MinSlot(SL_s) = \langle s, sv\rangle \mid \langle s, sv\rangle \in SL_s \wedge \forall \langle s_s, sv_s\rangle \in SL_s, s \leq s_s$

Get Byte Size:
Get the size in bytes of the thing that is passed in.

```
1: function GetSize($a$)
2:     return Size in bytes of $a$
3: end function
```

Error:
Prints an error message and halts the execution of the client.

```
1: function Error($msg$)
2:     $Print(msg)$
3:     $Halt()$
4: end function
```

Get Next Sequence Number:
Get the next sequence number for insertion into the block chain.

```
1: function GETNEXTSEQ($k$)
   ▷ Get the largest known sequence number
2:     $seq_{ret} \leftarrow seq$ such that $\langle id, seq \rangle \in LastSlo \land (\forall \langle id', seq' \rangle \in LastSlo, seq \geq seq')$
3:
   ▷ Add one to the largest seq number to generate the new seq number
4:     return $seq_{ret} + 1$
5: end function
```

Get Arbitrator:
Get the arbitrator for a given key.

```
1: function GETARBITRATOR($k$)
2:     $\langle k_1, id_1 \rangle \leftarrow \langle k_2, id_2 \rangle$ such that $\langle k_2, id_2 \rangle \in Arbitrator \land k_2 = k$
3:     return $id_1$
4: end function
```

Get Arbitrator for KV Set:
Get the arbitrator for a given key value set.

```
1: function GETARBITRATORKV($KV$)
2:     $\langle k, v \rangle \leftarrow \langle k', v' \rangle$ such that $\langle k', v' \rangle \in KV$
3:     $\langle k_1, id_1 \rangle \leftarrow \langle k_2, id_2 \rangle$ such that $\langle k_2, id_2 \rangle \in Arbitrator \land k_2 = k$
4:     return $id_1$
5: end function
```

Check Arbitrator for a Transaction:
Check that the arbitrators for a given set are all the same arbitrator.

```
function CheckArbitrator($PendingTrans_a$)
    $id_{arb} \leftarrow NULL$

    $\langle KV_a, Guard_a \rangle \leftarrow PendingTrans_a$
    for all $\langle k', v' \rangle \in KV_a$ do
        $id' \leftarrow$ GetArbitrator($k'$)

        if $id_{arb} = NULL$ then
            $id_{arb} \leftarrow id'$
        else if $id' \neq id_{arb}$ then        ▷ Check all arbitrators are the same
            Error("Multiple arbitrators for key values in transaction.")
        end if
    end for
    for all $\langle k', v', lop' \rangle \in Guard_a$ do
        $id' \leftarrow$ GetArbitrator($k'$)

        if $id_{arb} = NULL$ then
            $id_{arb} \leftarrow id'$
        else if $id' \neq id_{arb}$ then        ▷ Check all arbitrators are the same
            Error("Multiple arbitrators for key values in transaction.")
        end if
    end for
end function
```

Get all Commits:
Get all commits that are currently in the local block chain. Iterate over all the local slots and extract all the commits from each slot.

```
function GetCommits()
    $ComSet \leftarrow \emptyset$                              ▷ Set of the commits

▷ Iterate over all the slots saved locally
    for all $\langle s_1', \langle seq_2', id', DE', hmac_p', hmac_c' \rangle\rangle \in LocalSlots$ do
        $ComSet \leftarrow ComSet \cup \{c | c \in DE', c \text{is a } commit\}$
    end for
    return $ComSet$
end function
```

Get all Transactions:
Get all transactions that are currently in the local block chain. Iterate over all the local slots and extract all the transactions from each slot.

```
1: function GetTrans()
2:     $TransSet \leftarrow \emptyset$                                ▷ Set of the trans
3:
   ▷ Iterate over all the slots saved locally
4:     for all $\langle s'_1, \langle seq'_2, id', DE', hmac'_p, hmac'_c \rangle\rangle \in LocalSlots$ do
5:         $TransSet \leftarrow TransSet \cup \{c | c \in DE', c \text{is a } trans\}$
6:     end for
7:     return $TransSet$
8: end function
```

Get all aborts:
Get all aborts that are currently in the local block chain. Iterate over all the local slots and extract all the aborts from each slot.

```
1: function GetAborts()
2:     $AbrtSet \leftarrow \emptyset$                                ▷ Set of the aborts
3:
   ▷ Iterate over all the slots saved locally
4:     for all $\langle s'_1, \langle seq'_2, id', DE', hmac'_p, hmac'_c \rangle\rangle \in LocalSlots$ do
5:         $AbrtSet \leftarrow AbrtSet \cup \{c | c \in DE', c \text{is a } abort\}$
6:     end for
7:     return $AbrtSet$
8: end function
```

Get all queue states:
Get all qstates that are currently in the local block chain. Iterate over all the local slots and extract all the qstates from each slot.

```
1: function GetQStates()
2:     $QSet \leftarrow \emptyset$                                ▷ Set of the qstates
3:
   ▷ Iterate over all the slots saved locally
4:     for all $\langle s'_1, \langle seq'_2, id', DE', hmac'_p, hmac'_c \rangle\rangle \in LocalSlots$ do
5:         $QSet \leftarrow QSet \cup \{c | c \in DE', c \text{is a } qstate\}$
6:     end for
7:     return $QSet$
8: end function
```

Get all last message data entrues:
Get all last msg that are currently in the local block chain. Iterate over all the local slots and extract all the last msg from each slot.

```
1: function GetLastMsg()
2:     $LMSet \leftarrow \emptyset$                                   ▷ Set of the last msg
3:
   ▷ Iterate over all the slots saved locally
4:     for all $\langle s'_1, \langle seq'_2, id', DE', hmac'_p, hmac'_c \rangle\rangle \in LocalSlots$ do
5:         $LMSet \leftarrow LMSet \cup \{c | c \in DE', c \text{is a } lastmsg\}$
6:     end for
7:     return $LMSet$
8: end function
```

Check Queue State Live:
A queue state is dead if there is another queue state data entry that has a larger queue state.

```
1: function CheckQStateLive($qstate_a$)
2:     $\langle size_a \rangle \leftarrow qstate_a$
3:     $AllQStates \leftarrow$ GetQState                         ▷ Get all the qstates
4:
5:     if $\exists \langle size' \rangle \in AllQStates, size' > size_a$ then
6:         return false
7:     end if
8:     return true
9: end function
```

Check Commit Live:
A commit is dead if for every key value pair in the commit there is a commit with a larger transaction sequence number that has a key value pair with the same key.

```
1: function CheckCommitLive($commit_a$)
2:     $\langle seq_{a_{trans}}, KV_a \rangle \leftarrow commit_a$
3:     $KSet \leftarrow \{k | \langle k, v \rangle \in KV\}$
4:     $AllCommits \leftarrow$ GetCommits                    ▷ Get all the commits
5:
   ▷ Iterate all commits that are newer in time
6:     for all $\langle seq'_{trans}, KV' \rangle \in AllCommits, seq'_{trans} > seq_{a_{trans}}$ do
7:         $KVSet \leftarrow KVSet \setminus \{k | \langle k, v \rangle \in KV'\}$
8:
9:         if $KVSet = \emptyset$ then
10:            return false                    ▷ All keys have a newer commit
11:        end if
12:    end for
13:    return true                    ▷ If got here then some keys still live
14: end function
```

Check Last Message Live:
The last message is dead if the device in question pushed a slot that has a larger sequence number than the one recorded in the last message data entry.

```
1: function CheckLastMsgLive($lastmsg_a$)
2:     $\langle seq_a, id_a \rangle \leftarrow lastmsg_a$
3:
4:     if $\exists \langle id', seq' \rangle \in LastSlot, id' = id_a \wedge seq' > seq_a$ then
5:         return false
6:     end if
7:     return True
8: end function
```

Check Collision Resolution Live:
Check if a collision resolution data entry is live or not. This done by checking if all clients that we know about have seen the collision resolution entry. This is checked by seeing if all devices have inserted a message with a larger sequence number into the block chain.

```
1: function CheckColResLive($colres_a$)
2:     $\langle id_a, seq_{a_{old}}, seq_{a_{new}}, equal_a \rangle \leftarrow colres_a$
3:
4:     if $\forall \langle id', seq' \rangle \in LastSlot, seq' \geq seq_{a_{new}}$ then
5:         return false
6:     end if
7:     return true
8: end function
```

Check New Key Live:
A new key data entry is always live.

```
1: function CheckNewkeyLive($newkey_a$)
2:     return True
3: end function
```

Check Abort Live:
Check if an abort data entry is live or not. Abort is dead if the device whos transaction was aborted sees the abort. This is checked by seeing if that device inserted a slot into the block chain which has a sequence numberl that is larger than the aborts sequence number.

```
1: function CheckAbortLive($abort_a, seq_a$)
2:     $\langle seq_{a_{trans}}, id_a \rangle \leftarrow abort_a$
3:
   ▷ The device whos transaction was aborted saw the abort
4:     if $\exists \langle id', seq' \rangle \in LastSlot, id' = id_a \wedge seq' > seq_a$ then
5:         return false
6:     end if
7:     return True
8: end function
```

Check Transaction Live:
A transaction is dead if there is an abort for that transaction or if there is a commit for that a transaction that came after this transaction. Since transactions must be committed in order of there insertion, seeing a transaction that is committed and has a larger sequence number than the transaction in question means that the transaction in question was committed at some point.

```
function CheckTransLive($trans_a$)
    $\langle seq_a, id_a, KV_a, Guard_a \rangle \leftarrow trans_a$
    $AllCommits \leftarrow$ GetCommits                    ▷ Get all the commits
    $AllAborts \leftarrow$ GetAborts                      ▷ Get all the aborts

    if $\exists \langle seq'_{abrt}, seq'_{trans}, id' \rangle \in AllAborts, seq'_{trans} = seq_a$ then
        return false
    else if $\exists \langle seq'_{trans}, KV' \rangle \in AllCommits, seq'_{trans} \geq seq_a$ then
        return false
    end if
    return true
end function
```

Check Live:
Checks if a data entry is live based on its type.

```
function CheckLive($datentry, seq$)
    if $datentry$ is a $commit$ then
        return CheckCommitLive($datentry$)
    else if $datentry$ is a $abort$ then
        return CheckAbortLive($datentry, seq$)
    else if $datentry$ is a $trans$ then
        return CheckTransLive($datentry$)
    else if $datentry$ is a $lastmsg$ then
        return CheckLastMsgLive($datentry$)
    else if $datentry$ is a $colres$ then
        return CheckColResLive($datentry$)
    else if $datentry$ is a $qstate$ then
        return CheckQStateLive($datentry$)
    else if $datentry$ is a $newkey$ then
        return CheckNewkeyLive($datentry$)
    else
        Error(”Unknown data entry type.”)
    end if
end function
```

Slot Has Live:
Check if the slot has any live data entries in it. Do this by looking at all the data entries in the slot and checking if they are live

```
function SlotHasLive($slot_a$)
    $\langle s_1, \langle seq_2, id, DE, hmac_p, hmac_c \rangle\rangle \in LocalSlots$
    for all $datentry \in DE$ do
        if CheckLive($datentry, s_1$) then          ▷ an entry is alive
            return true
        end if
    end for
    return false                                  ▷ All entries were dead
end function
```

Calculate Resize Threshold:
Calculate a threshold for how many slots need to have live data entries in them for a resize to take place.

```
1: function CalcResizeThresh($maxsize$)
2:     return $\lfloor maxsize * RESIZE\_THRESH\_PERCENT \rfloor$
3: end function
```

Calculate Block Chain New Size:
Calculate the new size of the block chain which we need if we are to resize the data structure.

```
1: function CalcNewSize($maxsize$)
2:     return $\lceil maxsize * RESIZE\_THRESH\_PERCENT \rceil$
3: end function
```

Should Resize:
Check if the block should resize based on some metric of how many slots in the block chain are filled with live data.

```
1: function ShouldResize()
2:     $LiveSlots \leftarrow \{slot_s | slot_s \in LocalSlots \wedge \text{SlotHasLive}(slot_s)\}$
3:     $resizethreshold \leftarrow \text{CalcResizeThresh}(max\_size)$
4:     return $|LiveSlots| \geq resizethreshold$  ▷ If passes threshold then resize
5: end function
```

Create Queue State:
Generate a queue state data entry.

```
1: function CreateQState($size_a$)
2:     return $\langle size_a \rangle$
3: end function
```

Create Abort:
Generate a abort data entry.

```
1: function CreateAbort($seq_a, id_a$)
2:     return $\langle seq_a, id_a \rangle$
3: end function
```

Create ColRes:
Generate a colres data entry.

```
function CreateColRes($is_a, seq_{a_{old}}, seq_{a_{new}}, isequal_a$)
    return $\langle id_a, seq_{a_{old}}, seq_{a_{new}}, \rangle isequal_a$
end function
```

Create Transaction:
Generate a transaction data entry.

```
function CreateTrans($pendingtrans_a, seq_a$)
    $\langle KV_a, Guard_a \rangle \leftarrow pendingtrans_a$
    return $\langle seq_a, LOCAL\_ID, KV_a, Guard_a \rangle$
end function
```

Create Commit:
Generate a commit data entry.

```
function CreateCommit($seq_a, KV_a$)
    return $\langle seq_a, KV_a \rangle$
end function
```

Create New Key:
Generate a new key data entry.

```
function CreateNewKey($k_a, id_a$)
    return $\langle k_a, id_a \rangle$
end function
```

Data Entry Set Has Space :
Checks if a data entry set has enough space for a new data entry to be inserted.

```
function DEHasSpace($DE_a, de_a$)
    $newsize \leftarrow$ GetSize($DE_a$)
    $newsize \leftarrow newsize +$ GetSize($de_a$)
    return $newsize \leq DATA\_ENTRY\_SET\_MAX\_SIZE$
end function
```

Create Rescued Date Entry:
For commits only the key-value pairs that are most recent (no newer commit that has those key values in it).

```
1: function CreateRescuedCommit($commit_a$)
2:     $AllCommits \leftarrow$ GetCommits
3:     $\langle seq_{a_{trans}}, KV_a \rangle \leftarrow de_a$
4:     $NewKV \leftarrow KV_a$
5:
   ▷ Get rid of all key values that have newer commits
6:     for all $\langle k_a, v_a \rangle \in KV_a$ do
   ▷ Iterate over all commits that are newer than the rescue commit
7:         for all $\langle seq', KV' \rangle \in AllCommits, seq' > seq_{a_{trans}}$ do
8:             if $\exists \langle k', v' \rangle \in KV', k' = k_a$ then
9:                 $NewKV \leftarrow NewKV \setminus \langle k_a, v_a \rangle$
10:                Break
11:            end if
12:        end for
13:    end for
14:    return $\langle seq_{a_{trans}}, NewKV \rangle$
15: end function
```

Create Rescued Date Entry:
Generate the data entry rescued version of the entry. For some data entry types such as commits, the entry is not rescued as is. For commits only the key-value pairs that are most recent (no newer commit that has those key values in it).

```
1: function CreateRescuedEntry($de_a$)
2:     if $de_a$is a *commit* then
3:         return CreateRescuedCommit($de_a$)
4:     end if
5:     return $de_a$                         ▷ No Modification needed
6: end function
```

Check Slot HMACs:
Check that each slot has not been tampered with by checking that the stored HMAC matches the calculated HMAC. Also check thatthe slot number reported by the server matches the slot number of the actual slot.

```
1: function CheckSlotsHmacAndSeq($Slots_a$)
2:     for all $slot_a \in Slots_a$ do
3:         $\langle seq_{a_1}, \langle seq_{a_2}, id_a, DE_a, hmac_{a_p}, hmac_{a_c} \rangle\rangle \leftarrow slot_a$
4:         $calchmac \leftarrow$ GenerateHmac($seq_{a_2}, id_a, DE_a, hmac_{a_p}$)
5:         if $seq_{a_1} \neq seq_{a_2}$ then
6:             Error("Slot sequence number mismatch")
7:         else if $calchmac \neq hmac_{a_c}$ then
8:             Error("Slot HMAC mismatch")
9:         end if
10:    end for
11: end function
```

Check HMAC Chain:
Check that the HMAC chain has not been violated.

```
function CheckHmacChain($Slots_a$)
    $SlotsList \leftarrow Slots_a$ sorted by sequence number

    ▷ Check all new slots
    for all $index \in [2 : |SlotsList|]$ do
        $\langle seq_{a_1}, \langle seq_{a_2}, id_a, DE_a, hmac_{a_p}, hmac_{a_c} \rangle\rangle \leftarrow SlotList[i-1]$
        $\langle seq_{b_1}, \langle seq_{b_2}, id_b, DE_b, hmac_{b_p}, hmac_{b_c} \rangle\rangle \leftarrow SlotList[i]$
        if $hmac_{b_p} \neq hmac_{b_c}$ then
            Error(”Invalid previous HMAC.”)
        end if
    end for

    ▷ Check against slots that we already have in the block chain
    if $|LocalSlots| \neq 0$ then
        $\langle seq, SDE \rangle \leftarrow$MaxSlot($LocalSlots$)
        $\langle seqlast_2, id_{last}, DE_{last}, hmac_{last_p}, hmac_{last_c} \rangle \leftarrow SDE$

        $\langle seq_{a_1}, \langle seq_{a_2}, id_a, DE_a, hmac_{a_p}, hmac_{a_c} \rangle\rangle \leftarrow SlotList[1]$

        if $(seq_{last_2} + 1) = seq_{a_1}$ then
            if $hmac_{a_p} \neq hmac_{last_c}$ then
                Error(”Invalid previous HMAC.”)
            end if
        end if
    end if
end function
```

Check For Old Slots:
Check if the slots are not new. Checks if the "new" slots are actually new or if they are older than the most recent slot that we have.

```
1: function CheckOldSlots($Slots_a$)
2:     $\langle seq_{new}, Dat_{new}\rangle \leftarrow$ MinSlot ($Slots_a$)    ▷ Get the oldest new slot
3:     $\langle seq_{local}, Dat_{local}\rangle \leftarrow$ MaxSlot ($LocalSlots$)    ▷ Get the newest slot seen
4:
5:     if $seq_{new} \leq seq_{local}$ then    ▷ The slots were not newer than what was already seen
6:         Error("Server sent old slots.")
7:     end if
8:
       ▷ Check if slots have the same sequence number but different data entries
9:     for all $\langle seq, Dat\rangle \in Slots_a$ do
10:        if $\exists\langle seq', Dat'\rangle \in (LocalSlots \cup Slots_a), seq' = seq \wedge Dat' \neq Dat$ then
11:            Error("Slot sequence number match but data does not")
12:        end if
13:    end for
14: end function
```

Get All Queue States:
Gets all the queue states from the slots that were passed in.

```
1: function GetQState($Slots_a$)
2:     $QSet \leftarrow \emptyset$
3:
4:     for all $\langle seq'_1, \langle seq'_2, id', DE', hmac'_p, hmac'_c\rangle\rangle \in Slots_a$ do
5:         for all $de' \in DE'$ do
6:             if $de'$ is a $qstate$ then
7:                 $QSet \leftarrow QSet \cup \{de'\}$
8:             end if
9:         end for
10:    end for
11:
12:    return $QSet$
13: end function
```

Check Size With Gap:
Checks that the block chain size is correct when there is a gap in the block chain. This check makes sure that the server is not hiding any information from the client. If there is a gap and there is only 1 queue state in the new slot entries then there must have at least that many slots since the old slot entry must have been purged. If there is more than 1 queue state then the block chain is still growing check the smallest max size and there should be at least that many slots.

```
function CheckSizeWithGap($Slots_a$)
    $QSet \leftarrow$ GetQState($Slots_a$)
    $size_{max} \leftarrow size$ such that $size \in QSet \wedge \forall size' \in QSet, size \geq size'$
    $size_{min} \leftarrow size$ such that $size \in QSet \wedge \forall size' \in QSet, size \leq size'$
    $Slots_{oldmax} \leftarrow \emptyset$

    ▷ If only 1 max size then we must have all the slots for that size
    if $(|QSSet| = 1) \wedge (|Slots_a| \neq size_{max})$ then
        Error("Missing Slots")
    end if

    ▷ We definitely have all the slots
    if then$|Slots_a| = size_{max}$
        return                                        ▷ We have all the slots
    end if

    ▷ We must have at least this many slots
    if then$|Slots_a| < size_{min}$
        Error("Missing Slots")
    end if

end function
```

Check Size:

```
1: function CheckSize($Slots_a$)
2:     $\langle seq_{old_{max}}, Dat_{old_{max}} \rangle \leftarrow$ MaxSlot($LocalSlots$)
3:     $\langle seq_{new_{max}}, Dat_{new_{max}} \rangle \leftarrow$ MinSlot($Slots_a$)
4:
5:     if $(seq_{old_{max}} + 1) = seq_{new_{max}}$ then
       ▷ No Gap so cannot say anything about the size
6:         return
7:     else
       ▷ Has a gap so we need to do checks
8:         CheckSizeWithGap($Slots_a$)
9:     end if
10: end function
```

Process Commit Data Entry:
Process a commit entry. Updates the local copy of commits.

```
1: function UpdateLastMessage($seq_a, id_a, LstSlt_a, updateinglocal_a$)
2:     $\langle id_{old}, seq_{old} \rangle \leftarrow \langle id', seq' \rangle$ such that $\langle id', seq' \rangle \in LastSlot \wedge id' = id$
3:
4:     if $id_a = LOCAL\_ID$ then
5:         if $\neg updateinglocal_a \wedge (seq_a \neq seq_{old})$ then
       ▷ This client did not make any updates so its latest sequence number
       should not change
6:             Error("Mismatch on local machine sequence number")
7:         end if
8:     else
9:         if $seq_{old} > seq_a$ then
10:            Error("Rollback on remote machine sequence number")
11:        end if
12:    end if
13:
14:    $LastSlot \leftarrow LastSlot \setminus \{\langle id, seq \rangle | \langle id, seq \rangle \in LastSlot, id = id_a\}$
15:    $LastSlot \leftarrow LastSlot \cup \{\langle id_a, seq_a \rangle\}$
16:    return $LstSlt_a \setminus \{\langle id, seq \rangle | \langle id, seq \rangle \in LstSlt_a, id = id_a\}$
17: end function
```

Process Commit Data Entry:
Process a commit entry. Updates the local copy of commits.

```
function ProcessCommit($commit_a$)
    $\langle seq_{a_{trans}}, KV_a \rangle \leftarrow commit_a$
    $DKV \leftarrow \{\langle k, v\rangle | \langle k, v\rangle \in CommittedKV \wedge \langle k', v'\rangle \in KV_a \wedge k' = k\}$
    $CommittedKV \leftarrow (CommittedKV \setminus DKV) \cup KV_a$
end function
```

Process Queue State Entry:
Process a queue state entry. Updates the max size of the block chain

```
function ProcessQState($qstate_a$)
    $\langle size_a \rangle \leftarrow qstate_a$
    $max\_size \leftarrow size_a$                ▷ Update the max size we can have
end function
```

Process Queue State Entry:
Process a collision resolution entry.

```
function ProcessColres($colres_a, NewSlots_a$)
    $\langle id_a, seq_{a_{old}}, seq_{a_{new}}, isequal_a \rangle$
    $AllSlots \leftarrow LocalSlots \cup NewSlots_a$
    $index \leftarrow seq_{a_{old}}$

    while $index <= seq_{a_{new}}$ do
        $slt \leftarrow \langle seq'Dat' \rangle$ such that $\langle seq'Dat' \rangle \in AllSlots \wedge seq' = index$
        if $\exists \langle seq'Dat' \rangle \in AllSlots, seq' = index$ then
            $\langle seq, Dat \rangle \leftarrow \langle seq'Dat' \rangle$ such that $\langle seq'Dat' \rangle \in AllSlots \wedge seq' = index$
            $\langle seq, id, DE, hmac_p, hmac_c \rangle \leftarrow Dat$
            if $isequal_a \neq (id = id_a)$ then
                Error("Trying to insert rejected messages for slot")
            end if
        end if

        $index \leftarrow index + 1$
    end while
end function
```

Process New Key Entry:
Process a queue state entry. Adds a key to the key arbitrator set

```
function ProcessNewkey($newkey_a$)
    $\langle seq_a, k_a, id_a \rangle \leftarrow newkey_a$
    $Arbitrator \leftarrow Arbitrator \cup \{\langle k_a, id_a \rangle\}$
end function
```

Process Data Entry:
Process the data entry based on what kind of entry it is.

```
function ProcessDatEntry($slot_a, NewSlots_a, LstSlt_a$)
    if $datentry_a$ is a $commit$ then
        ProcessCommit($dataentry_a$)
    else if $datentry_a$ is a $abort$ then
▷ Do Nothing in this case
    else if $datentry_a$ is a $trans$ then
▷ Do Nothing in this case
    else if $datentry_a$ is a $lastmsg$ then
        $\langle seq_a, id_a \rangle \leftarrow dataentry_a$
        $LstSlt_a \leftarrow$ UpdateLastMessage($seq_a, id_a, LstSlt_a, false$)
    else if $datentry_a$ is a $colres$ then
        ProcessColres($dataentry_a, NewSlots_a$)
    else if $datentry_a$ is a $qstate$ then
        ProcessQState($dataentry_a$)
    else if $datentry_a$ is a $newkey$ then
        ProcessNewkey($dataentry_a$)
    else
        Error("Unknown data entry type.")
    end if
    return $LstSlt_a$
end function
```

Delete Local Slots:
Deletes local slots that are deleted at the server. This keeps the size of the local block chain bounded.

```
1: function DeleteLocalSlots()
2:     $\langle seq_{max}, Dat_{max} \rangle \leftarrow$ MaxSlot($LocalSlots$)
3:     $seq_{min} \leftarrow seq_{max} - max\_size$    ▷ Min sequence number we should keep
4:     $LSDelete \leftarrow \emptyset$
5:     if $|LocalSlots| \leq max\_size$ then
6:         return                                  ▷ Nothing to delete
7:     end if
8:
9:     $LSDelete \leftarrow \{\langle seq', Dat' \rangle | \langle seq', Dat' \rangle \in LocalSlots, seq' > seq_{min}\}$
10:    $LocalSlots \leftarrow LocalSlots \setminus LSDelete$
11: end function
```

Create Speculative KV:
Speculates on what the most recent key value pairs will be based on the latest committed key value pairs and the uncommitted transactions.

```
1: function SpeculateKV()
2:     $AllTrans \leftarrow$ GetTrans
3:     $LiveTrans \leftarrow \{t | t \in AllTrans$,CheckTransLive($t$)$\}$
4:     $CurrKV \leftarrow CommittedKV$
5:     $DKV \leftarrow \emptyset$
6:     for all $\langle seq_t, id_t, KV_t, Guard_t \rangle \in LiveTrans$ ordered by $seq'$ do
7:         if EvaluateGuard($Guard_t, CurrKV$) then
8:             $DKV \leftarrow \{\langle k, v \rangle | \langle k, v \rangle \in CurrKV \wedge \langle k', v' \rangle \in KV_t \wedge k' = k\}$
9:             $CurrKV \leftarrow (CurrKV \setminus DKV) \cup KV_t$
10:        end if
11:    end for
12:    return $CurrKV$
13: end function
```

Validate Update:
Validate the block chain and insert into the local block chain.

```
function ValidateUpdate($NewSlots_a, updatinglocal_a$)
    $\langle seq_{oldest}, Dat_{oldest} \rangle \leftarrow$ MinSlot($NewSlots_a$)
    $\langle seq_{newest}, Dat_{newest} \rangle \leftarrow$ MaxSlot($NewSlots_a$)
    $\langle seq_{local}, Dat_{local} \rangle \leftarrow$ MaxSlot($LocalSlots$)
    $LastSlotTmp \leftarrow LastSlot$

    CheckSlotsHmacAndSeq($NewSlots_a$)    ▷ Check all the HMACs
    CheckHmacChain($NewSlots_a$)    ▷ Check HMAC Chain
    CheckOldSlots($NewSlots_a$)    ▷ Check if new slots are actually old slots
    CheckSize($NewSlots_a$)    ▷ Check if the size is correct

    for all $slot_a \in NewSlots_a$ in order of sequence number do
        if $slot_a \in LocalSlots$ then    ▷ Client already has this slot
            $NewSlots_a \leftarrow NewSlots_a \setminus \{slot_a\}$
            Continue
        end if

        $\langle seq_{a_1}, \langle seq_{a_2}, id_a, DE_a, hmac_{a_p}, hmac_{a_c} \rangle \rangle \leftarrow slot_a$
        $LstSlt_a \leftarrow$ UpdateLastMessage($seq_{a_1}, id_a, LstSlt_a, updatinglocal_a$)

        for all $de_a \in DE_a$ do    ▷ Process each data entry
            $LstSlt_a \leftarrow$ ProccessDatEntry($de_a, NewSlots_a, LstSlt_a$)
        end for

        $LocalSlots \leftarrow LocalSlots \cup \{slot_a\}$    ▷ Add to local Chain
    end for

    if $seq_{oldest} > (seq_{local} + 1) \wedge LastSlotTmp \neq \emptyset$ then
    ▷ There was a gap so there should be a complete set of information on each previously seen client
        Error("Missing records for machines")
    end if

    DeleteLocalSlots( )    ▷ Delete old slots from local
    $SpeculatedKV \leftarrow$ SpeculateKV( )    ▷ Speculate on what will be latest KV set
end function
```

Decrypt Validate Insert Slots:
Decrypts slots, validates (checks for malicious activity) slots and inserts the slots into the local block chain.

```
function DecryptValidateInsert($NewSlots_a, updatinglocal_a$)
    $DecryptedSlots \leftarrow \emptyset$
    $DDat \leftarrow NULL$

    for all $\langle seq', EDat' \rangle \in NewSlots_a$ do
        $DDat \leftarrow$ Decrypt($EDat'$)
        $DecryptedSlots \leftarrow DecryptedSlots \cup \langle seq', DDat \rangle$
    end for

    ValidateUpdate($DecryptedSlots, updatinglocal_a$)
end function
```

Check and Create Last Message Data Entry:
Check if a last message entry needs to be created for this slot and if so create it. The check is done by checking if there are any newer slots with the same id or if there is already a last message slot with a newer sequence number

```
function CheckCreateLastMsgEntry($seq_a, id_a$)
    $AllLastMsg \leftarrow$ GetLastMsg

    ▷ Already Has one
    if $\exists \langle seq', id' \rangle \in AllLastMsg, id_a = id' \wedge seq' = seq_a$ then
        return {}

    end if

    ▷ Not latest slot from that client
    if $\exists \langle seq'_1, \langle seq'_2, id', DE', hmac'_p, hmac'_c \rangle \rangle \in LocalSlots, id_a = id' \wedge seq'_1 > seq_a$ then
        return {}

    end if

    return $\{\langle seq_a, id_a \rangle\}$
end function
```

Mandatory Rescue:
This rescue is mandatory before any types of data entries (excpet queue states) can be placed into the data entry section of the new slot. Returns the data entry Set or null if the first slot could not be cleared (the live data in that slot could not fit in this current slot).

```
1: function MANDATORYRESCUE($DE_a$)
2:     $smallestseq \leftarrow seq$ such that $\langle seq, DE\rangle \in LocalSlots \wedge (\forall\langle seq', DE'\rangle \in LocalSlots, seq \le seq')$
3:     $cseq \leftarrow smallestseq$
4:
       ▷ Check the least slots to rescue and live entries
5:     while $cseq < (smallestseq + DEAD\_SLOT\_COUNT)$ do
6:         $currentslot \leftarrow s'$ such that $\langle s', DE'\rangle \in LocalSlots \wedge s' = cseq$
7:         $\langle seq', \langle seq'_2, id', DE', hmac'_p, hmac'_c\rangle\rangle \leftarrow currentslot$
8:         $DE' \leftarrow DE' \cup$ CHECKCREATELASTMSGENTRY($seq', id'$) ▷ Get the last message too if we need it
9:
10:        for all $de \in DE'$ do                    ▷ Iterate over all the entries
11:            if CHECKLIVE($de, cseq$) then              ▷ data entry is live
12:                $de \leftarrow$ CREATERESCUEDENTRY(de)        ▷ Resize entry if needed
13:                if DEHASSPACE($DE_a, de$) then
14:                    $DE_a \leftarrow DE_a \cup de$        ▷ Had enough space to add it
15:                else if $currentseq = smallestseq$ then
16:                    return $NULL$
17:                else
18:                    return $DE_a$
19:                end if
20:            end if
21:        end for
22:
23:        $cseq \leftarrow cseq + 1$                   ▷ Move onto the next slot
24:    end while
25:    return $DE_a$
26: end function
```

Optional Rescue:
This rescue is not mandatory. This is trying to fill the remaining portion of the slot with rescued data so that no space is wasted. If we encounter a data entry that does not fit move on to the next, maybe that one will fit. Do this until we skipped too many live data entries

```
1: function OptionalRescue($DE_a$)
2:     $smallestseq \leftarrow seq$ such that $\langle seq, DE\rangle \in LocalSlots \land (\forall \langle seq', DE'\rangle \in LocalSlots, seq \leq seq')$
3:     $largestseq \leftarrow seq$ such that $\langle seq, DE\rangle \in LocalSlots \land (\forall \langle seq', DE'\rangle \in LocalSlots, seq \geq seq')$
4:     $numofskips \leftarrow 0$
5:     $cseq \leftarrow smallestseq$
6:
   ▷ Check the least slots to rescue and live entries
7:     while $cseq < largestseq$ do
8:         $currentslot \leftarrow s'$ such that $\langle s', DE'\rangle \in LocalSlots \land s' = cseq$
9:         $\langle seq', \langle seq'_2, id', DE', hmac'_p, hmac'_c\rangle\rangle \leftarrow currentslot$
10:
11:        for all $de \in DE'$ do                          ▷ Iterate over all the entries
12:            if CheckLive($de, cseq$) then                ▷ data entry is live
13:                $de \leftarrow$ CreateRescuedEntry(de)   ▷ Resize entry if needed
14:
15:                if $de \in DE_a$ then                    ▷ Already being rescued
16:                    Continue
17:                end if
18:
19:                if DEHasSpace($DE_a, de$) then
20:                    $DE_a \leftarrow DE_a \cup de$       ▷ Had enoug space to add it
21:                else if $numofskips \geq MAX\_RESCUE\_SKIPS$ then
22:                    return $DE_a$
23:                else$numofskips \leftarrow numofskips + 1$
24:                end if
25:            end if
26:        end for
27:
28:        $cseq \leftarrow cseq + 1$                       ▷ Move onto the next slot
29:    end while
30:    return $DE_a$
31: end function
```

Rejected Messages:

```
function RejectedMessages($DE_a$)
    $seq_{old} \leftarrow seq$ such that $\langle seq \rangle \in RejectedSlotList \wedge \forall \langle seq' \rangle \in RejectedSlotList, seq \geq seq'$
    $prev \leftarrow -1$

    if $|RejectedSlotList| \geq REJECTED\_THRESH$ then
        $seq_{new} \leftarrow seq$ such that $\langle seq \rangle \in RejectedSlotList \wedge \forall \langle seq' \rangle \in RejectedSlotList, seq \leq seq'$

        $colres \leftarrow$ CreateColRes($LOCAL\_ID, seq_{old}, seq_{new}, false$)
        return $DE_a \cup \{colres\}$
    end if

    for all $\langle seq \rangle \in RejectedSlotList$ sorted by $seq$ do
        if $\exists \langle seq', Dat' \rangle \in LocalSlots$ then
            Break
        end if
        $prev \leftarrow seq$
    end for

    if $prev \neq -1$ then
        $DE_a \leftarrow DE_a \cup$ CreateColRes($LOCAL\_ID, seq_{old}, prev, false$)
    end if

    $RejectedSlotList \leftarrow \{\langle seq \rangle | \langle seq \rangle \in RejectedSlotList, seq > prev\}$

    for all $\langle seq \rangle \in RejectedSlotList$ sorted by $seq$ do
        $DE_a \leftarrow DE_a \cup$ CreateColRes($LOCAL\_ID, seq, seq, false$)
    end for

    return $DE_a$
end function
```

Arbitrate:

```
function ARBITRATE($DE_a$)
    $AllCommits \leftarrow$ GETCOMMITS
    $AllTrans \leftarrow$ GETTRANS
    $LiveCommits \leftarrow \{c | c \in AllCommits,$ CHECKCOMMITLIVE$(c)\}$
    $LiveTrans \leftarrow \{t | t \in AllTrans,$ CHECKTRANSLIVE$(t)\}$
    $KV \leftarrow \emptyset$
    $lastcomseq \leftarrow -1$
    $CurrKV \leftarrow \emptyset$
    $DKV \leftarrow \emptyset$
    $KVTmp \leftarrow \emptyset$

▷ Get all the latest commits
    for all $\langle seq'_{trans}, KV' \rangle \in LiveCommits$ do
        $CurrKV \leftarrow CurrKV \cup KV'$
    end for

    for all $\langle seq_t, id_t, KV_t, Guard_t \rangle \in LiveTrans$ ordered by $seq'$ do
        if GETARBITRATORKV$(KV_t) \neq LOCAL\_ID$ then
            Continue            ▷ Client not arbitrator for this transaction
        end if

        if $\neg$EVALUATEGUARD$(Guard_t, CurrKV)$ then
            $abortde \leftarrow$CREATEABORT$(seq_t, id_t)$
▷ No more space so we cant arbitrate any further
            if ( then$lnot$DEHASSPACE$(DE_a, abortde)$)
                return $DE_a$
            end if
            $DE_a \leftarrow DE_a \cup abortde$
        else
            $DKV \leftarrow \{\langle k, v \rangle | \langle k, v \rangle \in KV \wedge \langle k', v' \rangle \in KV_t \wedge k' = k\}$
            $KVTmp \leftarrow (KV \setminus DKV) \cup KV'$
            $DKV \leftarrow \{\langle k, v \rangle | \langle k, v \rangle \in CurrKV \wedge \langle k', v' \rangle \in KVTmp \wedge k' =$
$k\}$
            $CurrKV \leftarrow (CurrKV \setminus DKV) \cup KVTmp$
            $commitde \leftarrow$ CREATECOMMIT$(seq_t, KVTmp)$
            if $\neg$ DEHASSPACE$(DE_a, commitde)$ then
                if $lastcomseq \neq -1$ then
                    $DE_a \leftarrow DE_a \cup$ CREATECOMMIT$(lastcomseq, KV)$
                end if
                return $DE_a$
            else
                $KV \leftarrow KVTmp$
                $lastcomseq \leftarrow seq_t$
            end if
        end if
    end for
    $DE_a \leftarrow DE_a \cup$ CREATECOMMIT$(lastcomseq, KV)$
```

Create New Slot:
Create a slot and encrypt it.

```
1: function CreateNewSlot($seq_a, DE_a$)
2:     $\langle seq, SDE\rangle \leftarrow \langle seq', SDE'\rangle$ such that $\langle seq', SDE'\rangle \in LocalSlots \wedge$
   $(\forall \langle seq'', DE''\rangle \in LocalSlots, seq' \geq seq'')$
3:     $\langle seq, id, DE, hmac_p, hmac_c\rangle \leftarrow SDE$
4:
5:     $newhmac \leftarrow$ GenerateHmac($seq_a, LOCAL\_ID, DE_a, hmac_p$)
6:     $newSDE \leftarrow \langle seq, LOCAL\_ID, DE_a, hmac_c, newhmac\rangle$
7:     $encryptnewSDE \leftarrow$Encrypt(newSDE)
8:
9:     return $\langle seq_a, encryptnewSDE\rangle$
10: end function
```

Send Data to Server:
Send the data to the server. If this fails then new slots will be returned by the server.

```
1: function SendToServer($seq_a, DE_a, newsize_a$)
   ▷ Make the slot and try to send to server
2:     $newslot \leftarrow$ CreateNewSlot($seq_a, DE_a$)
3:     $\langle success, newslots\rangle \leftarrow$ PutSlot($seq_a, newslot, newsize_a$)
4:
5:     if $success$ then
6:         $RejectedSlotList \leftarrow \emptyset$
7:         return $\langle true, \{newslot\}\rangle$
8:     else
9:         if $|newslots| = 0$ then
10:            Error("Server rejected but did not send any slots")
11:        end if
12:        $RejectedSlotList \leftarrow RejectedSlotList \cup \{seq_a\}$
13:        return $\langle false, newslots\rangle$
14:    end if
15:
16: end function
```

Try Insert Transaction:
Try to insert a transaction into the block chain. Does resizing, rescues and insertion of other data entry types as needed.

```
function TryInsertTransaction($pendingtrans_a, forceresize$)
    $DE \leftarrow \emptyset$                                ▷ The data entries for this slot
    $seq \leftarrow$ GetNextSeq               ▷ Get the sequence number for this slot
    $newsize \leftarrow 0$
    $trans \leftarrow$ CreateTrans($pendingtrans_a, seq$)
    $transinserted \leftarrow false$
    $slotstoinsert \leftarrow \emptyset$

    $resize \leftarrow$ ShouldResize( )                    ▷ Check if we should resize
    $resize \leftarrow resize \lor forceresize$
    if $resize$ then
        $newsize \leftarrow$ CalcNewSize($max\_size$)
        $DE \leftarrow DE \cup \{$CreateQState($newsize$)$\}$
    end if

    if $RejectedSlotList \neq \emptyset$ then
        $DE \leftarrow$ RejectedMessages($DE$)
    end if

    $DE \leftarrow$ MandatoryRescue($DE$)                     ▷ Round 1 of rescue
    if $DE = NULL$ then
▷ Data was going to fall off the end so try again with a forced resize
        return TryInsertTransaction($trans_a, true$)
    end if

    $DE \leftarrow$Arbitrate($DE$)

    if DEHasSpace($DE, trans$) then                          ▷ transaction fits
        $DE \leftarrow DE \cup trans$
        $transinserted \leftarrow true$
    end if

▷ Rescue data to fill slot data entry section
    $DE \leftarrow$ OptionalRescue($DE$)

▷ Send to server.
    $\langle sendsuccess, newslots\rangle \leftarrow$ SendToServer($seq, DE, newsize$)

▷ Insert the slots into the local bloakc chain
    DecryptValidateInsert($newslots, true$)

    return $transinserted \land success$          ▷ Return if succeeded or not
end function
```

Try Insert New Key:
Try to insert a new key into the block chain. Does resizing, rescues and insertion of other data entry types as needed.

```
function TryInsertNewKey($k_a, id_a, forceresize$)
    $DE \leftarrow \emptyset$                                   ▷ The data entries for this slot
    $seq \leftarrow$ GetNextSeq              ▷ Get the sequence number for this slot
    $newsize \leftarrow 0$
    $newkey \leftarrow$ CreateNewKey($k_a, id_a$)
    $newkeyinserted \leftarrow false$
    $slotstoinsert \leftarrow \emptyset$

    $resize \leftarrow$ ShouldResize( )                ▷ Check if we should resize
    $resize \leftarrow resize \lor forceresize$
    if $resize$ then
        $newsize \leftarrow$ CalcNewSize($max\_size$)
        $DE \leftarrow DE \cup \{$CreateQState($newsize$)$\}$
    end if

    if $RejectedSlotList \neq \emptyset$ then
        $DE \leftarrow$ RejectedMessages($DE$)
    end if

    $DE \leftarrow$ MandatoryRescue($DE$)                  ▷ Round 1 of rescue
    if $DE = NULL$ then
▷ Data was going to fall off the end so try again with a forced resize
        return TryInsertNewKey($k_a, id_a, true$)
    end if

    $DE \leftarrow$Arbitrate($DE$)

    if DEHasSpace($DE, newkey$) then                          ▷ new key fits
        $DE \leftarrow DE \cup newkey$
        $newkeyinserted \leftarrow true$
    end if

▷ Rescue data to fill slot data entry section
    $DE \leftarrow$ OptionalRescue($DE$)

▷ Send to server.
    $\langle sendsuccess, newslots \rangle \leftarrow$ SendToServer($seq, DE, newsize$)

▷ Insert the slots into the local block chain
    DecryptValidateInsert($newslots, true$)

    return $newkeyinserted \land success$    ▷ Return if succeeded or not
end function
```

4.4 Client Interfaces

Put Key Value Pair:
Puts a key value pair into the key value pair buffer

```
1: function PutKeyValue($k, v$)
2:     $\langle seq, KV, Guard \rangle \leftarrow PendingTrans$
3:
       ▷ Check if KV already has a key value pair for the specified key
4:     $DSet \leftarrow \{\langle k_1, v_1 \rangle | \langle k_1, v_1 \rangle \in KV \wedge k_1 = k\}$
5:
6:     if $DSet \neq \emptyset$ then
7:         Error(”Value for key already in most recent update”)
8:     end if
9:
10:    $KV \leftarrow KV \cup \{\langle k, v \rangle\}$                ▷ Add key value pair
11:    $PendingTrans \leftarrow \langle seq, KV, Guard \rangle$
12:    CheckArbitrator($PendingTrans$)        ▷ Check that the
       transaction still valid
13: end function
```

Get KV Pair Speculative:
Get the value for the key while speculating.

```
1: function GetValueSpeculate($k_a$)
2:     $\langle k, v \rangle \leftarrow \langle k, v \rangle$ such that $\langle k, v \rangle \in SpeculatedKV \wedge k = k_a$
3:     return $v$
4: end function
```

Update
Sync with the server and get all the latest slots.

```
1: function Update()
2:     $\langle seq, Dat \rangle \leftarrow$ MaxSlot($LocalSlots$)
3:     $NewSlots \leftarrow$ GetSlots($seq$)
4:     DecryptValidateInsert($NewSlots, false$)
5: end function
```

Get KV Pair Committed:
Get the value for the key which have been committed.

```
1: function GetValueCommit($k_a$)
2:     $\langle k, v\rangle \leftarrow \langle k, v\rangle$ such that $\langle k, v\rangle \in Committed \wedge k = k_a$
3:     return $v$
4: end function
```

Put Guard:
Puts a guard transaction into the key value update. A guard is a key value with a logical operator (*lop*).

```
1: function PutGuard($k, v, lop$)
2:     $\langle seq, KV, Guard\rangle \leftarrow PendingTrans$
3:
4:     if $\langle k, v, lop\rangle \in Guard$ then
5:         return                    ▷ Already have guard condition in update
6:     end if
7:
8:     $Guard \leftarrow Guard \cup \{\langle k, v, lop\rangle\}$
9:     $PendingTrans \leftarrow \langle seq, KV, Guard\rangle$
10:    CheckArbitrator($PendingTrans$)          ▷ Check that the transaction still valid
11: end function
```

Transaction Start:
Starts a transaction. Clears out the key value pair update buffer.

```
1: function TransactionStart()
2:     $PendingTrans \leftarrow NULL$
3: end function
```

Transaction Commit:
Commits the transaction into the block chain. Keeps attempting to insert the transaction into the block chain until it succeeds.

```
function TRANSACTION COMMIT()
    $DKV \leftarrow \emptyset$
    $pt \leftarrow NULL$

    $PendingTransQueue$.PUSH($PendingTrans$)

    while HASCONNECTIONTOSERVER( ) $\wedge PendingTransQueue \neq \emptyset$ do
        $pt \leftarrow PendingTransQueue$.PEAK( )

        if TRYINSERTTRANSACTION($pt, false$) then
            $PendingTransQueue$.POP( )
        end if
    end while

    ▷ Go Through local pending transactions and speculate
    for all $\langle KV, Guard\rangle \in PendingTransQueue$ do
        if EVALUATEGUARD($Guard, SpeculatedKV$) then
            $DKV \leftarrow \{\langle k,v\rangle | \langle k,v\rangle \in SpeculatedKV \wedge \langle k',v'\rangle \in KV \wedge k' = k\}$
            $SpeculatedKV \leftarrow (SpeculatedKV \setminus DKV) \cup KV$
        end if
    end for
end function
```

Create New Key:
Creates a new key and specifies which machine ID is the arbitrator. If there is already a new key entry in the block chain for this key name then do not insert into the chain, another client got there first.

```
1: function TRANSACTION COMMIT($k_a, id_a$)
2:     $success \leftarrow false$
3:
4:     while $\neg success$ do
5:         if $\exists \langle k', id' \rangle \in Arbitrator, k' = k_a$ then
6:             return $false$                              ▷ Key already created
7:         end if
8:
9:         $success \leftarrow$ TRYINSERTNEWKEY($k_a, id_a, false$)
10:    end while
11:    return $true$                ▷ If got here then insertion was correct
12: end function
```