Validity check for Crytography

Function sign_object(msg,private_key):

```
serialized_object = serialize the message
hex_signed_object = sign serialized_object using private_key and hex-enccode the result
return hex_signed_object
```

Function verifying_message(signed_msg_object):

```
decoded_signature = decode the signed_msg from hex to bytes
verifying_key =  sender_public_key
if verify(decoded_signature,verifying_key)
        return True
else:
        return false
```

Syncing up of replicas that got behind:

BlockTree:

pending_blocks: dictionary of block ids and votes corresponding to blockid
Candidate_block_to_commit: dictionary of block ids and block
Procedure execute_and_insert(block):
if current_round + 1 < block.round:
start_sync(block.id)
if current_round <= max{highest_vote_round,qc.vote_info.round} then:
// for a replica lagging behind its highest_vote_round will be less that the qc_round
// And qc_round will be greater than block round
send sync event to every other replica

Procedure start_sync(current_round,block.round):
last_commited_block = Ledger.get_last_commited_block()
broadcast SyncMsgRequest((last_commited_block,sender=u))

Main: EventLoop

Procedure start_event_processing(M)
if M is a **SyncMsgRequest process_sync**(M)
if M is a **BlockSyncMessage process_block**(M)

Main:
Procedure process_sync(M):
if u!=M.sender:
next_block = Ledger.get_next_block(M.last_committed.id):
last_committed_block = Ledger.get_last_commited()
BlockSyncMessage(next_block,last_committed_block,sender=M.sender)

Procedure process_block(M):
if u==M.sender:
pending_blocks[M.next_block.id]++
candidate_block_to_commit[M.next_block.id]=M.next_block
BlockTree.pending_block_tree.add(M.next_block.vote_info.id, M.next_block.id,M.next_block)
if **pending_blocks**[M.next_block.id] == 2*f + 1
// wait for a 2*f+1 blocks before we commit it to the ledger
Ledger.commit(M.next_block.id)
Pacemaker.round = M.next_block.round // to sync the round number so that it can participate in the election for proposal phase
if M.last_commited_block.id != M.next_block.id
SyncMsgRequest((Ledger.get_last_commited,sender=u))

Client requests: de-duplication; include appropriate requests in proposals

Client:

responses: dict(list) // maps transaction with replica response for client transaction (author,txn, id)
transcationMap: dict() // maps transaction with f+1 consistent authors and block id

MainLoop:**Client_EventLoop**
loop: wait for next event M ; Main.**start_event_processing**(M)

Procedure start_event_processing(M)

if M is "**transcation_commited**" then process_transcation(M)
if M is **timeout** then on_timeout(M)
If M is **validation_block** then process_valiation_block(M)

Procedure broadcast(message):

get replica process
start_timer(message.txn) // start timer for a transcation
send "request" message to these replicas

Procedure on_timeout(M):

broadcast(M.message)
start_timer(M.message)

Procedure process_transcation(M):

responses[txn] <- responses[txn] U (M.block_id,M.block_author,M.txn)
check_block_state(M.txn)

Procedure check_block_state(txn):

txn_count = dic()
Check of f+1 consistent hash for txn:

txn_map[txn] = {txn.id,txn.authors} // storing all the author who are part of f+1 , this will allow us to get transaction from anyone of them

stop_timer(txn)

send event "**validate**" for block_id to one of the txn.authors

Procedure validation(block):

If block.payload.signature is same as signature(block.payload, pvt_key):

Block.payload verified

Main:

Procedure start_event_processing(M)
    If M is **validate** process_validate(M)

Procedure proces_validate(M):
    // M is block_id
    block = ledger.commited_block(M)
    Send "**validation_block**" for block to source

MemPool:
    transcation_queue: queue()
    state:set() // used for storing current transaction processing state
    locator:dict // to hold client info for transcation

    Procedure insert_command(command):
        if command not in locator and not in state:
            transcation_queue.enque(command)
            locator[command]=source // clientid

    Function get_transcations():
        If transaction_queue is not empty:
            command <- transaction_qeue.pop
            If command in locator and command not in state:
                state.add(command)
                return command
            else:
                return get_transcations()
        else:
            return ⊥