**Pseudocode**

**Verify signature:**

Using nacl.signing for digital signature verification

**Function *validate\_signatures***(signatures\_list):

//for each signature check if any of the replica public key can verify it and return True

for each signature in signatures\_list :

is\_signature\_validated ← False

for each key in replica\_pubic\_keys :

is\_signature\_validated ← is\_signature\_validated or ***verify\_signature***(signature,key)

if is\_signature\_validated:

**return** True

if not is\_signature\_validated:

**return** False

**return** True

**Function verify\_signature**(signature, key):

//Verifies signature with a public key

verify\_key = VerifyKey(key, encoder=HexEncoder)

**try**:

verify\_key.verify(signature, encoder=HexEncoder)

**except** nacl.exceptions.BadSignatureError:

**return** False

**return** True

**Syncing up replicas that got behind**

**Main : EventLoop**

**Procedure start\_event\_processing(M):**

If M is proposal message and M.high\_commit\_qc.vote\_info.round < Block-Tree.high\_commit\_qc.round-1:

send(msg\_type="sync\_request",Block-Tree.high\_commit\_qc.round,to=(all\_replicas))

If M is vote message and M.high\_commit\_qc.round < Block-Tree.high\_commit\_qc.round-1:

send(msg\_type="sync\_request",Block-Tree.high\_commit\_qc.round,to=(all\_replicas))

If M is timeout message and M.timeout\_info.high\_qc.ledger\_commit\_info.commit\_state\_id is not none and M.timeout\_info.high\_qc.vote\_info.parent\_round < Block-Tree.high\_commit\_qc.round:

send(msg\_type="sync\_request",Block-Tree.high\_commit\_qc.round,to=(all\_replicas))

**Procedure receive** (msg\_type="sync\_request",round,from = replica):

data = (Ledger.get\_txn\_after\_round(round), Block-Tree.pending\_block\_tree, Ledger.pending\_state\_tree)

send("sync\_response",data, to replica)

**Procedure receive** (msg\_type="sync\_response", M, from replica):

data\_hash = hash(data)

pending\_sync\_responces[data\_hash] ← pending\_sync\_responces[hash(data,replica)] U data.signature

if |pending\_sync\_responces[data\_hash]| == 2f+1:

(txns, pending\_block\_tree, pending\_state\_tree) = data

Ledger.commit\_txns(txns)

Ledger.pending\_state\_tree = pending\_state\_tree

Block-Tree.pending\_block\_tree = pending\_block\_tree

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

Mempool maintains request cache of received transactions from client storing information of the states (‘queue’,’processing’,’’processed) of each transaction.

**Procedure**  **Mempool\_push\_transaction**(transaction\_id): //pushes transactions in mempool

//transaction not received or in processing and not commited

If transaction\_id not in request\_cache or request\_cache.get(transaction\_id) == ‘processing’:

pending\_transactions\_set U {transaction\_id}

transaction\_queue. append (transaction \_id)

request\_cache.set(transaction\_id, “queue”)

If request\_cache.get(transaction\_id) in {“queue”,"processed"}:

//Duplicate or commited transaction

send\_response (to = client,

msg={transaction\_id at state = request\_cache.get(transaction\_id)})

**client pseudocode: verify that a submitted command was committed to the ledger**

**at client:**

**Function : is\_committed**(transaction):

send\_request(command,to=(replicas)) //send request to all replicas

await( len(results[command]) = f+1:

return True

else if timeout:

return False

**Function receive**((command,message), from = replica):

If message == processed:

results[command] U {replica}

at replica :

**Procedure**  **Mempool\_push\_transaction**(transaction\_id): //pushes transactions in mempool

………

If request\_cache.get(transaction\_id) in {"processed"}:

//commited transaction

send\_response (to = client,

msg={transaction\_id at state = request\_cache.get(transaction\_id)})