Manual

DiemBFT Consensus Algorithm

Algomaniacs

Sumeet Pannu, Pratik Nagelia, Shubham Agrawal

17th October, 2021

Contents

Build and Run System

Requirements:

1. Python
2. DistAlgo
3. Pip
4. PyNacl

```
pip3 install pynacl
pip3 install pyDistAlgo
python3.7 -m pip install cffi
```

Run:

```
python3.7 -m da main_run_tests.da ./config/config.json
```

Configuration

test_name : Name of the test scenario

F : Number of faulty nodes

Nodes : Total Number of Nodes

Clients : Total number of Clients

Delta : The duration for the timeout

Client Requests : The number of transaction to be initiated per client

Example Test case Work :

```
[{
 "test_name" : "Test 1 : Client 1 Nodes 4, 4 Transactions",
 "F": 1,
 "nodes": 4,
 "clients": 2,
 "delta" : 20000,
 "client_requests" : 4
}]
```

Platform

DistAlgo Version: 1.1.0b15

Python Implementation: CPython

Python Version: 3.7.12

OS: Linux

Type of hosts: Google Cloud Platform (VM)

Pseudocode

1. **Validity Checks for Cryptographic Values**

Whenever a message transaction happens, we are using Nacl to digitally sign the message. Digital signatures allow you to publish a public key, and then you can use your private signing key to sign messages. Others who have your public key can then use it to validate that your messages are actually authentic.

[Reference: https://pynacl.readthedocs.io/en/latest/signing/]

In the DiemBFT, while performing message transactions, we sign it with the private key and verify with the public key which was generated at the time of creation of the private key.

Client Signing the message with its own private key

```
Client
client_req = ClientRequest(txn, self.private_key.sign(bytes(str(txn.client_id),
encoding='utf-8')))
send(("TXN", client_req), to=nodes)
```

Class: Node.da

```
# Replica verifying the signature with the public key sent from the client

Func valid_client_signature(signature, client_id):
   try:
       self.clientPublicKeyDict[client_id].verify(signature)
   except nacl.exceptions.BadSignatureError:
       print("[Node-{}] Signature Invalidated for client Id {} ".format(self.node_id,
client_id))
       return False
   return True


# Proposer Signing Message Proposal Message

def process_new_round_event(last_tc, transaction):
   if self.node_id == self.leader_election.get_leader(self.pacemaker.current_round):
       print("[Node-{}] Processing New Round Event as the leader".format(self.node_id))
       block = self.block_tree.generate_block(transaction, self.pacemaker.current_round)
       proposal = ProposalMessage(block, last_tc, self.block_tree.high_commit_qc,
       self.private_key.sign(bytes(str(block.block_id), encoding='utf-8')), self.node_id)
       send(("PROPOSAL", proposal, self.node_id), to=nodesDict.values())

# Timeout Message Signature Validation

def valid_timeout_message(self, tmo_msg):
   try:
       self.public_keys[tmo_msg.tmo_info.sender].verify(tmo_msg.tmo_info.signature)
   except nacl.exceptions.BadSignatureError:
       print("[Node-{}] Signature Invalidated for Timeout Message from Node- {}
".format(self.node_id, tmo_msg.tmo_info.sender))
       return False
   return True
```

Class : Safety.da

```
# Other Replicas Verifying the signature of the Proposal Message
def valid_signature(self, block, timeout_certificate, signature):
   try:
       self.public_key[block.author].verify(signature)
   except nacl.exceptions.BadSignatureError:
       print("[Safety-Node-{}] BLOCK_IDX: [{}] Signature Invalidated ".format(self.node_id,
block.block_id))
       return False
   return True


# Voter Signing Message

def sign(self, ledger_commit_info):
   return self.private_key.sign(bytes(ledger_commit_info))

# Signing Timeout Message

def signTimeoutInfo(self, round, highqc_round):
   return self.private_key.sign(bytes(str(round)+str(highqc_round)))
```

Class: BlockTree.da

```
# Proposer Validating Message
try:
   self.public_keys[vote_msg.sender].verify(vote_msg.signature)
   collected_votes.append(vote_msg)
except nacl.exceptions.BadSignatureError:
   print("[Block-Tree-Node-{}] VOTE_IDX: [{}] Signature Invalidated ".format(self.node_id,
str_vote_idx))
```

2. Deduplication

Class: MemPool.da

```
deduplication_set = set() #set which will contain the hash of incoming requests
pending_transactions = [] #list of transactions that are yet pending to be fed into the
system and committed
in_process_transactions = [] #list of transactions that are currently being served by the
system
committed_transactions = [] #list of transactions that have been committed
committed_set = ()
```

```
    procedure add_transaction(transaction)
        hash_txn = hash(transaction) #get the string contained in the trsn
        if hash_txn exists in self.deduplication_set {
          #do nothing -> current transaction already exists in the system
          return
        }
        deduplication_set.add(hash_txn)
        pending_transactions = pending_transaction U (transaction) #add transaction to the
pending transaction queue
```

```
    procedure get_transactions:
        if size(pending_transactions)=0 {
              #There isnt any pending transaction that needs to be served.
            return None
        }
        #fetch the transaction that came first
        pending_txn <- pending_transactions.pop(0)
        in_process_transactions = in_process_transactions U pending_txn #add the pending
transaction to the in_process queue
        return pending_txn
```

```
    procedure commit_transaction(transaction):
        hashC <- hash(transaction)
        in_process_transactions.remove(transaction)
        committed_transactions.add(transaction)
        committed_set.add(hashC)
```

3. **Verify that a submitted command was committed to the ledger**

Class Client.da

```
def check_commited_transaction(txn):
        client_req = ClientRequest(txn,
self.private_key.sign(bytes(str(txn.client_id), encoding='utf-8')))
        send(("CHECK_COMMITTED", client_req), to=nodes)
        output('[CLIENT] Initiated Check Commited transaction sent to all
nodes. Waiting for a reply....')
        time.sleep(1)

    def receive(msg=("TRANSACTION_COMMIT_RESPONSE", payload), from_=node):
        print("[CLIENT- {}] Received whether the submitted transaction was
committed or not: {}".format(self.client_id, payload))
```

Mempool

It contains the status for each transaction whether it is in NEW_TRANSACTION, PENDING,

IN_PROGRESS or COMMITTED

```
def update_transaction_status(self, transaction, new_status):
        if transaction is None:
            return
        self.output_to_files("[MEMPOOL-Node-{}] Update Status of TXN: {} to
{}".format(self.node_id, transaction.to_string(), new_status))
        hash_txn = hash(transaction.to_string())
        self.transaction_status[hash_txn] = new_status

    def get_transaction_status(self, transaction):
        if transaction is None:
            return "IGNORE"
        hash_txn = hash(transaction.to_string())
        if hash_txn not in self.transaction_status:
            return "NEW_TRANSACTION"
        return self.transaction_status[hash_txn]

    def check_txn_status(self, transaction):
        hash_txn = hash(transaction.to_string())
        if self.transaction_status[hash_txn] is 'COMMITTED':
            return True
        return False
```

```
    def if_committed_or_not(self, transaction):
        status = self.get_transaction_status(transaction)
        if status is "COMMITTED":
            return True
        return False
```

Node: Contains the receive handlers

```
def receive(msg=("TRANSACTION_COMMITTED_OR_NOT", client_request),
from_=client):
        payload = client_request.transaction
```

```
        if self.valid_client_signature(client_request.signature,
payload.client_id):
            string = "[Node-{}] Received {} message:".format(self.node_id,
"TRANSACTION_COMMITTED_OR_NOT")+" | Message: "+payload.to_string()
            self.output_to_files(string)
            status = self.mempool.if_committed_or_not(transaction)
            send(("TRANSACTION_COMMIT_RESPONSE", status), to=client)
```

Ledger: Committing entries to the ledger and updating the status of the transaction in Mempool

```
def commit(self, block_id):
        file = open(self.filename, 'a')
        if block_id not in self.pending_transactions:
            self.output_to_files("[Ledger-Node-{}] Already
Committed~~~~~~~~~~~~~~~~~~~~~~".format(self.node_id))
            return
        pending_txn = self.pending_transactions[block_id]
        txn_status = self.mempool.get_transaction_status(pending_txn)
        if block_id in self.pending_transactions and txn_status is not
"COMMITTED":
            pending_txn = self.pending_transactions[block_id]
            msg = str(datetime.datetime.now()) + " " +
pending_txn.to_string() + '\n'
            file.write(msg)
            file.flush()
            self.output_to_files("[Ledger-Node-{}] Committing pending
transaction for block_id: [{}]. Corresponding transaction: [{}]
**********************************************************************
*****".format(self.node_id, block_id, pending_txn.to_string()))
            self.pending_transactions.pop(block_id)
            self.mempool.update_transaction_status(pending_txn,"COMMITTED")
            self.committed_transactions[block_id] = pending_txn
            return pending_txn
        else:
            self.output_to_files("[Ledger-Node-{}] Nothing to commit at
this time".format(self.node_id))
            return None
        file.close()
```

WorkLoad Generation

Various workloads as well as test cases can be generated by editing the `***diemBFT/config/config.json***` file. The json configuration is explained above.

Contributions

Sumeet Pannu - Understanding the algorithm, completed parts of code in Node, Ledger, Leader Election, Block Tree, Safety, Pacemaker, Testing and documentation.

Pratik Nagelia - Understanding the algorithm, completed the parts of code in Node, Block tree, Ledger, Safety, Pacemaker, Clients, Testing and documentation.

Shubham Agrawal - Understanding the algorithm, completed parts of code in Node, Block Tree, Signatures, Safety, Ledger, Clients, Testing and documentation.