1. Pseudo-code(10PTS)

Validity checks for cryptographic values: (3pts)

- We used SHA256 for cryptographic hashes, and Ed25519 for digital signatures.

Safety Module:

#valid_signatures() uses the verify_key/public keys of the author to verify the signature for qc and last_tc.
Function valid_signatures(qc,last_tc)
If qc != None :
    try:
        for(author, signature) in qc.signatures:
            public_keys[author].verify(signature)
    except:
        return false
If last_tc!= None :
    try:
        for(author, signature) in last_tc.tmo_signatures:
            public_keys[author].verify(signature)
    except:
        return false
return true

#sign_u() signs the given payloads by converting them into bytes using pickle.dumps(), and then uses the private key to sign the payload_bytes.
Function sign_u(*payloads)
    payloads_bytes = pickle.dumps(payloads)
    return private_key.sign(payloads_bytes)

#verify_client() verifies the client's private key using the public key.
Function verify_client(author, signature):
    try:
        public_keys[author].verify(signature)
    except:
        return False
    return True

For message types that contain a signature field, we will use the safety.sign_u() function to generate signatures.

"sync up" replicas that got behind: (3pts)

Syncmanager module

```
syncing = False
syncing_high_commit_qc = None
syncing_high_qc = None
attempted_peers = set()
msg_queue = []
u = None
multicast = None
process_certificate_qc = None
handle_msg = None
server_ids = None

# return True if we need to sync up before processing this proposal
Function need_sync(remote_high_commit_qc, remote_high_qc, msg, src):
    global syncing
    global syncing_high_commit_qc
    global syncing_high_qc
    if syncing:
        msg_queue.append((msg, src))
        print('%s is already syncing' % u)
        return True
    syncing_high_commit_qc = remote_high_commit_qc
    syncing_high_qc = remote_high_qc
    if not synced():
        print('%s starting new sync up to ' % u, syncing_high_qc.vote_info.id)
        syncing = True
        msg_queue.append((msg, src))
        send_sync_request()
        return True
    return False

Function need_vote_sync(remote_high_commit_qc, vote_info, msg, src):
    if not blocktree.pending_block_tree.contains(vote_info.id):
        return True

Function process_sync_request(sync_request, src):
    if blocktree.pending_block_tree.contains(sync_request.high_qc.vote_info.id):
        process_certificate_qc(sync_request.high_qc)
    block_id = sync_request.high_commit_qc.vote_info.parent_id if sync_request.high_commit_qc else None
    chain = ledger.get_commit_chain(block_id)
    path = blocktree.path_from_commit_root()
    resp = SyncResponseMsg(chain, path)
    multicast(('sync_response', resp), src)
```

```
Function process_sync_response(sync_response, src):
    print('%s got sync response' % u)
    if not syncing:
        return False
    # first catch up high_commit_qc
    if not sync_high_commit_qc(sync_response):
        print('failed to sync high_commit_qc')
    if not sync_high_qc(sync_response):
        print('failed to sync high_qc')
    # check if we're done
    if synced():
        print('%s finished syncing up with %s' % (u, server_ids[src]))
        end_sync()
    else:
        print('%s failed to sync with response from %s' % (u, server_ids[src]))
        attempted_peers.add(server_ids[src])
        send_sync_request()

# ---------------
# private methods
# ---------------

Function high_commit_qc_synced():
    if syncing_high_commit_qc == None:
        return True
    if blocktree.high_commit_qc and syncing_high_commit_qc.vote_info.parent_round <
blocktree.high_commit_qc.vote_info.parent_round:
        return True
    return blocktree.pending_block_tree.contains(syncing_high_commit_qc.vote_info.parent_id)

Function high_qc_synced():
    # first proposal
    if syncing_high_qc == None:
        return True
    if blocktree.high_qc and syncing_high_qc.vote_info.round < blocktree.high_qc.vote_info.round:
        return True
    return blocktree.pending_block_tree.contains(syncing_high_qc.vote_info.id)

Function synced():
    return high_commit_qc_synced() and high_qc_synced()

Function send_sync_request():
    # first bring high_commit_qc up to date
    req = SyncRequestMsg(blocktree.high_commit_qc, syncing_high_qc)
    qc = syncing_high_qc if high_commit_qc_synced() else syncing_high_commit_qc
    signators = {replica for replica, signature in qc.signatures}
    # randomly pop a replica from the quorum
    # if we timeout, let the next proposal trigger a new sync
    not_tried = signators - attempted_peers
```

```
    # there must be non faulty nodes in signators
    # so if we received a response, it would end the sync
    # and clear attempted_peers
    assert len(not_tried)
    multicast(('sync_request', req), not_tried.pop())

Function timeout():
    global syncing
    if syncing:
        print('%s sync ending due to timeout' % u)
    msg_queue.clear()
    attempted_peers.clear()
    end_sync()

Function end_sync():
    global syncing
    syncing = False
    msg_queue.reverse()
    while(len(msg_queue)):
        handle_msg(*msg_queue.pop())
    attempted_peers.clear()

Function sync_high_commit_qc(sync_response):
    if not len(sync_response.commit_chain):
        return False
    if syncing_high_commit_qc and sync_response.commit_chain[-1].round <
syncing_high_commit_qc.vote_info.parent_round:
        return False
    # root of commit_chain should have qc referencing our own highest committed block
    for block in sync_response.commit_chain:
        if block.qc and not blocktree.pending_block_tree.contains(block.qc.vote_info.id):
            return False
        process_certificate_qc(block.qc)
        blocktree.execute_and_insert(block)
    return True

Function sync_high_qc(sync_response):
    if not len(sync_response.high_qc_path):
        return False
    if sync_response.high_qc_path[-1].round < syncing_high_qc.vote_info.round:
        return False
    for block in sync_response.high_qc_path:
        if not blocktree.pending_block_tree.contains(block.qc.vote_info.id):
            # we could receive a path whose prefix we have already committed
            # so continue until we have the pending parent block
            continue
        process_certificate_qc(block.qc)
        blocktree.execute_and_insert(block)
    return True
```

Replica module

```
Function process_proposal_msg(P, sender):
    if syncmanager.need_sync(P.high_commit_qc, P.block.qc, ('proposal', P), sender):
        pacemaker.advance_round_tc(P.last_round_tc)
        Return
          . . . . . .

Function process_timeout_msg(M, sender):
    if not syncmanager.need_sync(M.high_commit_qc, M.tmo_info.high_qc, ('timeout', M), sender):
        process_certificate_qc(M.tmo_info.high_qc)
        process_certificate_qc(M.high_commit_qc)
    # we can still handle the qc's in the TC even if we aren't synced up to them
    # because we don't put these qc's in blocktree until they are proposed in future rounds
    pacemaker.advance_round_tc(M.last_round_tc)
    tc = pacemaker.process_remote_timeout(M)
    if tc:
        pacemaker.advance_round_tc(tc)
        process_new_round_event(tc)
Function process_vote_msg(M, sender):
    if syncmanager.need_vote_sync(M.high_commit_qc, M.vote_info, M, sender):
        print('%s ignoring vote because not synced' % u)
        return
    qc = blocktree.process_vote(M)
    if qc:
        process_certificate_qc(qc)
        process_new_round_event(None)

Function handle_msg(M, sender):
. . . . . .
    elif msg_type == 'sync_request':
        output('%s received %s message from %s' % (to_str(), M[0], server_ids[sender]))
        syncmanager.process_sync_request(M[1], sender)
    elif msg_type == 'sync_response':
        output('%s received %s message from %s for high_qc round %d' % (to_str(), M[0], server_ids[sender],
M[1].high_qc_path[-1].round))
        syncmanager.process_sync_response(M[1], sender)
. . . . . .
```

Pacemaker module

```
Function local_timeout_round():
    timeout_info = safety.make_timeout(current_round, blocktree.high_qc, last_round_tc)
    if timeout_info:
        timeout_msg = blocktree.TimeoutMsg(timeout_info, last_round_tc, blocktree.high_commit_qc)
        broadcast(('timeout', timeout_msg))
        syncmanager.timeout()
        start_timer(current_round)
        logging.info(f"[Timeout with timeout message]\n\
            [{timeout_msg}]")
```

Client requests: deduplication; include appropriate requests in proposals: (2pts)

Replica module:

```
Function handle_msg(M, sender):
. . . . . .
    elif msg_type == 'request' and sender in clients:
        # process request from client
        txn = M[1]
        author = txn[1]
        signature = M[2]
        if not safety.verify_client(author, signature):
            # invalid signature, reject the request
            pass
        # check if duplicate (otherwise add to mempool.pending)
        commit_state_id = mempool.check_transaction(txn, sender)
        if commit_state_id:
            # duplicate request, reply to client
            multicast(('committed', txn, commit_state_id), to=sender)
. . . . . .
```

Mempool module:

```
Function check_transaction(txn, client):
    uuid = txn[2]
    # Check whether transaction has already been committed
    if uuid in requests and requests[uuid][1] != None:
        requests.move_to_end(uuid, last=False)
        return requests[uuid][1]
    if len(requests) == SIZE:
```

```
        requests.popitem()
    requests[uuid] = (txn, None)
    return False
```

Block-Tree module:

```
Function path_from_commit_root():
    path = []
    qc = high_qc
    if high_commit_qc == None:
        high_commit_qc_parent_id = None
    else:
        high_commit_qc_parent_id = high_commit_qc.vote_info.parent_id
    while qc != None and qc.vote_info.id != high_commit_qc_parent_id:
        block = pending_block_tree.id_to_block[qc.vote_info.id].block
        path.append(block)
        qc = block.qc
    # increasing round order
    path.reverse()
    return path
```

Client pseudocode: verify that a submitted command was committed to the ledger: (2pts)

Mempool module:

```
Function commit_transactions(txns, commit_state_id, clients, multicast):
    for (command, author, uuid) in txns:
        requests.move_to_end(uuid, last=False)
        requests[uuid] = ((command, author, uuid), commit_state_id)
        client = [c for c in clients if c == author]
        if not len(client):
            output('unrecognized client in txn (this should never happen)')
        client = client[0]

        multicast(('committed', (command, author, uuid), commit_state_id), to=client)
        if len(requests) == SIZE:
            committed.popitem()
```

Client module:

```
Function run():
. . . . . .
for i in range(workload.count):
   committed = False
   uuid = '%xu' % random.getrandbits(16*8)
   while not committed:
      txn = send_request(i, uuid)
      if await(len(setof(sender, received(('committed', _txn, commit_state_id), from_=sender))) >
            num_faulty):
         committed = True
      elif timeout(duration):
         duration *= 2
   duration /= 2
. . . . . .
```

READ-ME (3PTS)

Platform

- Ethan Deturk:
  - DistAlgo version: 1.1.0b15
  - Python implementation and version: CPython 3.7.3
  - Operating system name and version: Debian 10
  - Type of host: laptop
- Andrew Burford:
  - DistAlgo version: 1.1.0b15
  - Python implementation and version: CPython 3.7.10
  - Operating system name and version: macOS Big Sur 11.5.2
  - Type of host: laptop
- Jiawei Qian
  - DistAlgo version: 1.1.0b15
  - Python implementation and version: implementation = CPython 3.7.9
  - Operating system name and version: Windows 10
  - Type of host: laptop

Workload generation

- We defined the workload as a named tuple. *Workload = namedtuple('Workload', ['type', 'count', 'num_clients', 'timeout', 'delays'], defaults=(1, []))*. Where the 'type' is an enumeration member of the *WorkloadType(Enum)*.
- When we pass the configuration to the run process, we will initialize a workload instance to the workload field of the config.
- **run.da**, **config.da** contains the implementation of workload generation

Timeouts

- The timeout formulas for the client depend on the workload type of the client:
  - Workload type = retransmit:
    - The timeout formula is the exponential backoff algorithm, if a timeout occurs after a duration, we will double the duration. The duration is initialized as 0.1.
  - Workload type = flood:
    - The timeout value is workload.timeout
  - Workload type = timed:
    - The timeout value is workload.timeout
- The timeout formula for server is 4*transmisson_delay_bound since it's recommended in the paper. The timeout value is the transmission delay bound, which is initialized in the configuration.

Bugs and Limitations

- There is a nondeterministic, infrequent error that occurs when a client tries to sync up and creates a malformed blocktree with missing blocks. This results in a key error and a loud warning message in the log, followed by a timeout.
- The error described above becomes much more frequent when there is a lot of logging output, or if the tests run in quick succession

Main files

- Files containing the main code for clients:
  - consensus/src/client.da
  - consensus/src/run.da
- Files containing the main code for replicas:
  - consensus/src/replica.da
  - consensus/src/run.da

Code size

- Total LOC(non-blank non-comment lines of code): 1613
- Algorithm LOC: 1172
  - About 75% of the Algorithm LOC is the algorithm itself.
  - About 25% of the Algorithm LOC is other functionality interleaved with it.
- Other LOC: 441
  - run.da, replicafi.da, logger.da, clients.da, syncmanager.da, config.da
- I obtained the counts by using CLOC

  github.com/AlDanial/cloc v 1.90 T=0.05 s (427.4 files/s, 44469.5 lines/s)

Language	files	blank	comment	code
DAL	20	144	324	1613
SUM:	20	144	324	1613

Language feature usage

- numbers of list comprehensions: 10
- numbers of dictionary comprehensions: 10
- numbers of set comprehensions: 4
- numbers of aggregations: 0
- numbers of quantifications: 3
- numbers of await statements: 5
- numbers of receive handlers: 1

Contributions

- Ethan DeTurk: Implemented BlockTree and Ledger. Made substantial revisions to other components
- Andrew Burford: Implemented MemPool, Replica, ReplicaFI, Clients, Config, SyncManager, and the run class
- Jiawei Qian: Implemented Pacemaker, Safety, LeaderElection, Logging, and documentation