Abhishek Joshi

111331268

Olympus Code:

//Messages and Data Strucutres used

Class Wedged\_statement:

replica\_id

checkpoint\_proof //latest checkpoint proof

replica\_history

Class Wedged\_request:

replica\_id

/\*

//Olympus's view of each replica, olympus uses

this when it receives a wedged response

to keep track of things

\*/

Class Olymp\_replica:

replica\_id

is\_valid

history\_len

checkpoint\_proofs

order\_proof

operations

//we think of each replica history as a sequence of

slot, vote(or operation) pairs. Essentially, each consistent replica

has a vote/operation for each slot. It may be empty too.

Class Replica\_vote\_list:

slot\_id

operation

Class Catchup\_shuttle:

start\_slot

op\_list;

op\_len;

Class Caughtup\_shuttle:

replica\_id

replica\_state\_hash

Class Get\_running\_response:

replica\_id

rep\_runnuing\_state

Class Get\_running\_request:

replica\_id

Class Inithist\_statement:

next\_state

Class Config:

config\_seq\_num

replica\_list //list of replica ids in this confgiuration

Class Config\_history:

Config curr\_config //pointer to current configuration

config\_hist; list of all configs

Class Checkpoint\_statement:

start\_slot

replica\_id

checkpoint\_hash //hash of history up to 100 slots

end\_slot

Class Checkpoint\_shuttle:

is\_result

replica\_id

checkpoint\_proof //list of checkpoint hashes

//Olympus fn:

Olympus responds to each \*\_event. We assume each of the msgs is mapped to an event and the decoding

of each msg has been done.

We just state the messages to be sent to the relevant replicas and have not shown how they are constructed,

the requisite information is assumed to be present.

On receiving a reconfig request, Olympus sends wedged requests to all replicas to which they respond with

their histories and checkpoint proofs. Olympus checks each of these messages for consistency and builds a list

of eligible replicas in get\_valid\_voters. After this, it tries to find a new quorum in find\_qorum function

After it has found the quorum, it finds the replica with the longest hostory, lh\_replica . It then issues

catch-up requests to all other replicas in the quorum asking them to execute the operations which they

do not share with the longest history. The replicas receive this and reply with the hash of result state after

executing those operations.

On receiving a response, we check if all hashed state values are same. If not, we need to find a new quorum.

If yes, we now send a get\_running\_state request to any one replica in the quorum, and it responds with its state.

If the hash of this state matches with what we previously computed, this state can be used as the state of the next

configuration.

If not, we have to find a new quorum again.

After receiving a correct state, we issue the inithist statements with the above state as the initial state.

def Olympus\_fn:

curr\_config = config\_history.curr\_config

Replica\_vote\_list vote\_list[] //all the replicas which can vote

Olymp\_replica olymp\_replicas=[] //olympus private list for each replica

if (event\_received\_getconfig\_request): //client wants the current config

send\_to\_client(config\_history.curr\_config)

if (event\_received\_reconfig\_request): //One of the replicas sent us a reconfig request

if (verified\_sender()): //verify it is sent by a replica we know/have the key

foreach replica in curr\_config.replica\_list:

send\_wedged\_request(replica\_id)

if (event\_received\_wedged\_statement) //a replica sent us a wedged response

get\_valid\_voters(wedged\_statements)): //defined below

find\_qorum(vote\_list, quorum) //defined below

//find replica with longest history

lh\_replica = get\_replica\_with\_max\_hist\_len(quorum)

//Now, the goal is to "catch-up" the other replicas.

foreach replica in quorum:

catchup\_shuttle.op\_len = lh\_replica.history\_len - replica.history\_len

//send all operations that the dest does not have

catchup\_shuttle.op\_list = lh\_replica.operations((replica.history\_len+1)

: lh\_replica.history\_len)

send\_catchup\_request(replica\_id, catchup\_shuttle)

if (event\_received\_caughtup\_statement):

for each replica in caughtup\_statments:

validate\_caughtup\_statment()

// do all caughtup statments have same hash.

if!(check\_consistency(caughtup\_statments))

find\_qorum(vote\_list, quorum)//if not, find a different quorum

else:

replica\_id = get\_next\_replica\_from\_quorum(quorum)

get\_running\_state(replica\_id)

if (event\_received\_get\_running\_response): //we received a running state

ch = caughtup\_statments[0].replica\_state\_hash //hash of any replica will do

//the hash of state of replica in running response statement and ch must match

if (ch == compute\_hash(get\_running\_response.rep\_runnuing\_state))

//defined below

initiate\_new\_config(get\_running\_response.rep\_runnuing\_state, quorum)

else:

//the hashes did not match, we need to find a new replica's running state

replica\_id = get\_next\_replica\_from\_quorum(quorum)

get\_running\_state(replica\_id)

/\*

We will verify for each replica, whether its history is consistent and it so,

we consider it eligible to vote , we also note the history\_len here.

We can rule out replicas which do not have a consistent history from the vote because

all honest replicas will have consistent history

\*/

def get\_valid\_voters(wedged\_statements):

foreach statement in wedged\_statements: //each replica that sent us a wedged response

if(verified\_sender); //verify this replica sent this statment

//verify that both the checkpoint proof and history proofs are consistent

if !(check\_consistency(replica\_history) || check\_consistency(checkpoint\_proof)):

olymp\_replica[replica\_id].is\_valid = FALSE

else:

add\_replica\_to\_vote\_list() //this is eligible to vote

olymp\_replica.history\_len = history\_len

else:

olymp\_replica[replica\_id].is\_valid = FALSE //we cannot consider this replica as a voting candidate

def find\_qorum(vote\_list, quorum):

//we have constructed a list of eligible voter replicas

//each replica's vote\_list is a list of <vote, slot> pairs

//vote = operation this replica believes for this slot

//We try to find t+1 replicas which agree on all slots that they have a vote for

def initiate\_new\_config(next\_state, quorum):

update\_config\_history(quorum)

for each replica in quorum:

send\_terminate\_message()

generate\_new\_keys()

send\_inithist\_message(next\_state, replica)

Client Code

Class Replica\_result:

result\_proof

replica\_hash

Class Replica\_err\_msg:

replica\_id

Class Client\_request:

uniq\_request\_id

operation

client\_signature

Client includes a uniq\_request\_id in all its requests. On timeout, it retries.

wait\_for\_response fn returns timeout status and received result if it receives one

before timing out.

It also checks the consistency of a result to detect conflicting results. If it sees

a conflict, it retransmits

def client\_fn:

Replica\_result replica\_result

Client\_request client\_request

Replica\_err\_msg replica\_err\_msg

if(init):

config = get\_curr\_config(olympus\_id)

uniq\_request\_id = get\_new\_request\_id()

client\_request = create\_signed\_operation\_statement(uniq\_request\_id, operation)

send\_operation(head, client\_request)

wait\_for\_response(timeout, replica\_result, has\_timed\_out, event\_recvd\_res)

if (has\_timed\_out):

client\_request = create\_signed\_operation\_statement(uniq\_request\_id, operation)

foreach replica in config: //retransmit to all replicas

client\_request.is\_retransmit=1

send\_operation(replica, client\_request)

wait\_for\_response(timeout, replica\_result, has\_timed\_out, event\_recvd\_res)

if(!has\_timed\_out)

if(received\_result):

err = validate\_proof(replica\_result.result\_proof)

if (err)

get\_curr\_config(olympus\_id)

else:

config = get\_curr\_config(olympus\_id)

//we got an error from some replica, perhaps they became immutable

if (event\_recvd\_err\_msg)

validate\_error\_msg(replica\_err\_msg);

config = get\_curr\_config(olympus\_id)

//waits on event "event" until timeout

def wait\_for\_response(timeout, received\_result, has\_timed\_out, event):

start\_timer(timeout)

while(!timeout):

if(event):// we received a response while waiting

if !(check\_consistency(received\_result))

has\_timed\_out=1

else:

has\_timed\_out=0

stop\_timer;

nop;

if (timeout) //client timed out waiting for result

has\_timed\_out=1

Replicas: head and others

//Messages and Data

Class Client\_request:

uniq\_request\_id

is\_retransmit

operation

client\_signature

Class Replica\_order\_statement:

replica\_id

order

slot\_id

operation

Class Replica\_order\_proof:

slot\_id

operation

replica-id

replica\_order\_statements //list of type Replica\_order\_statements

Class Replica\_result\_proof:

replica\_id

operation

result\_hash

Class Replica\_shuttle:

uniq\_request\_id

is\_result\_shuttle

Class Replica\_order\_proof replica\_order\_proof

Class Replica\_result\_proof replica\_result\_proof

Class Reconfig\_request:

replica\_id

Class Wedged\_statement:

replica\_id

checkpoint\_proof //latest checkpoint proof

replica\_history

Class Wedged\_request:

replica\_id

Class Catchup\_shuttle:

start\_slot

op\_list;

op\_len;

Class Caughtup\_shuttle:

replica\_id

replica\_state\_hash

Class Get\_running\_response:

replica\_id

rep\_runnuing\_state

Class Get\_running\_request:

replica\_id

Class Checkpoint\_statement:

start\_slot

replica\_id

checkpoint\_hash //hash of history up to 100 slots

end\_slot

Class Checkpoint\_shuttle:

is\_result

replica\_id

checkpoint\_proof //litt of checkpoint hashes

Class Replica\_history:

order\_proofs //list of objects of type Replica\_order\_proof

curr\_slot\_id //next slot\_id we expect to fill, note that we expect to fill sequentially

Class Replica:

replica\_id

Replica\_history history

mode //ACTIVE, IMMUTABLE, PENDING

//This is the fn for a non-head replica

acceptable proof of misbehavior scenarios:

1. If a replica gets a non-consistent forward result shuttle .

2. If a replica gets a non-consistent backward result shuttle.

3. If a replica gets a non-consistent checkpt. proof

Non-consistent: all replicas don't agree on the proof. Note that this implies all

replicas say something about the proof. Missing statemetns cannot be construed as

proof of misbehaviour

On each of the above events, we issue a reconfig request to Olympus.

def replica\_fn(replica\_id, curr\_config, is\_head, is\_tail):

if (event\_intihist\_msg):

transition\_becomActive

if (event\_timeout):

send\_reconfig\_request(Olympus)

if (event\_wedge\_request):

validate\_request(wedge\_request)

wedge\_statement = create\_wedge\_statement(replica.history, checkpoint\_proof)

send\_statement(wedge\_statement, Olympus)

transition\_becomeImmutable

if (event\_received\_shuttle && replica\_shuttle.is\_result): //this is a result shuttle

err = check\_consistency(replica\_shuttle);//check the result and order proof in the shuttle

if (!err)

// we will cache this using its req\_id to use in case of retransmission

cached\_results.append(replica\_shuttle, replica\_shuttle.uniq\_request\_id)

else:

//misbehavior, result shuttle is inconsistent

send\_reconfig\_request(Olympus)

if (event\_client\_request): //client has retransmitted the operation

validate\_request(client\_request)

if (replica.mode == IMMUTABLE)

reply\_client(error);

else if (client\_request\_id in cached\_results): //is this result cached

reply\_client(replica\_shuttle.result\_proof)

elif:

forward\_request(get\_head(curr\_config)) //forward to head

wait\_for\_response(timeout, received\_result, has\_timed\_out, event\_received\_shuttle)

if (!has\_timed\_out ):

check\_consistency(replica\_shuttle.result\_proof);

reply\_client(replica\_shuttle.result\_proof)

else:

send\_reconfig\_request(Olympus) //reconfig on timeout

if(event\_received\_shuttle && !replica\_shuttle.is\_result):

err = check\_consistency(replica\_shuttle) //check the result and order proof in the shuttle

if (!err):

transition\_orderCommand(replica\_shuttle, replica, curr\_config, operation)

if(is\_tail)

send\_to\_client(replica\_shuttle)

send\_to\_correct\_shuttle(cuur\_config, replica\_shuttle)

else:

send\_reconfig\_request(Olympus) //misbehavior, result shuttle is inconsistent

if (event\_checkpoint\_shuttle)://head wants us to create a checkpoint

if !(checkpoint\_shuttle.check\_consistency):

err = check\_consistency()

if (!err)

create\_checkpoint\_statement(replica.history, checkpoint\_shuttle.start\_slot,

checkpoint\_shuttle.end\_slot)

checkpoint\_shuttle.checkpoint\_proof.append(checkpoint\_statement)

//backwards if this is the tail, forward if not.

send\_to\_correct\_shuttle(cuur\_config, checkpoint\_shuttle)

else if (checkpoint\_shuttle.is\_result):

//do usual checks

truncate\_history(replica.history, checkpoint\_shuttle.start\_slot, checkpoint\_shuttle.end\_slot)

if (event\_get\_running\_request):

send\_signed\_running\_state(curr\_state, Olympus)

if (event\_catchup\_request): //we got a catchup request from Olympus

foreach operation in catchup\_shuttle.op\_list:

apply\_operation(operation)

compute\_hash(curr\_state)

send\_caughtup\_response(Olympus)

def transition\_orderCommand(replica\_shuttle, replica, curr\_config, operation):

rep\_hist = replica.history

rep\_order\_proof = rep\_hist.order\_proofs[rep\_hist.curr\_slot\_id]

if (rep\_hist.curr\_slot\_id + 1) != replica\_shuttle.slot\_id: //there is a hole or conflicting slot operation

return 1;

if (replica.replica\_id in curr\_config && replica.mode == ACTIVE):

/do all replicas before us agree on this proof and have all of them issued this same statement

if (check\_consistency(replica\_shuttle.order\_proof):

order\_statement = create\_order\_statement()

result = apply\_operation(cur\_state, operation)

//this is a signed statement with result\_hash containing the hash of result

result\_statement = create\_result\_statement(operation, result)

replica\_shuttle.replica\_order\_proof.append(order\_statement)

replica\_shuttle.replica\_result\_proof.append(result\_statement)

replica.history.order\_proofs[slot\_id].append(replica\_shuttle)

replica.history.curr\_slot\_id++

return 0

def transition\_becomeImmutable(replica):

if replica.mode == ACTIVE:

replica.mode == IMMUTABLE

def transition\_becomActive:

if replica.mode == PENDING:

replica.mode = ACTIVE

//backwards if this is a tail replica, forward if not.

def send\_to\_correct\_replica(cuur\_config, shuttle):

if (next\_replica(cuur\_config)):

send\_shuttle(next\_replica(curr\_config), replica\_shuttle)

if (replica.is\_tail)

replica\_shuttle.is\_result = TRUE;

send\_shuttle(prev\_replica(curr\_config))

Head Replica:

//done by head

Only the head functionality different from the above replica function is shown, like

receiving requests from client. The rest is the same as above

Head sends issue\_checkpoint\_requests after every it receives n (say, 100) result proofs

def head\_replica\_fn(replica\_id, curr\_config, is\_head, is\_tail):

//head functionality for all other events is same as other replicas

if (event\_client\_request):

validate\_request(client\_request)

if (replica.mode == IMMUTABLE)

reply\_client(error);

if client\_request.is\_retransmit:

if (client\_request.uniq\_request\_id in cached\_results): //is this result cached

reply\_client(replica\_shuttle.result\_proof)

elif (replica.history.order\_proof[slot\_id]): //have we already ordered this operation

wait\_for\_response(timeout, received\_result,

has\_timed\_out, event\_received\_shuttle)

if (!has\_timed\_out):

//got a result\_proof while waiting

reply\_client(received\_result.result\_proof)

else:

send\_reconfig\_request(Olympus)

else: //head has not seen this operation before

//create new request and forward to next repica

replica\_shuttle= create\_new\_shuttle(get\_next\_slot\_id, client\_request)

//backwards if this is the tail, forward if not.

send\_to\_correct\_replica(curr\_config, replica\_shuttle))

wait\_for\_response(timeout, received\_result,

has\_timed\_out, event\_received\_shuttle)

if (!has\_timed\_out):

reply\_client(received\_result.result\_proof)

else:

send\_reconfig\_request(Olympus)

else: //this is a fresh, not retransmitted request

replica\_shuttle= create\_new\_shuttle(get\_next\_slot\_id, client\_request)

send\_to\_correct\_replica(curr\_config, replica\_shuttle))

def issue\_checkpoint\_request(replica, curr\_config):

create\_checkpoint\_statement(replica.history, checkpoint\_shuttle.start\_slot, checkpoint\_shuttle.end\_slot)

checkpoint\_shuttle.checkpoint\_proof.append(checkpoint\_statement)

send\_to\_correct\_shuttle(cuur\_config, checkpoint\_shuttle) //backwards if this is the tail, forward if not