Byzantine Chain Replication

Pratik Sushil Zambani

pzambani@cs.stonybrook.edu, 111500248

Keshav Gupta

kegupta@cs.stonybrook.edu, 111464733

on receiving …, on … are handlers for specific events.
def func() are local functions.

Client

```
pending_requests = [] # tracks requests whose responses have not been
                      # received
replicas_public_keys = [] # to authenticate msgs from replicas
timer = Timer() # contains timer related functions like start, stop
```

---

on send request to head

```
request = client gets request from application
# sends a request to olympus for current configuration
config = olympus.get_config()
send (“request”, request, request_id, retransmitted=0) to config.head
pending_requests.append(request_id)
# timer for the above request
timer.start()
```

---

on timer expire

```
# resend the request
config = olympus.get_config()
for each replica in config:
      send (“request”, request, request_id, retransmitted=1) to replica
# timer for the above request
timer.start()
```

---

on receiving *error* statement*:*

```
# case when replica is immutable, fetch new config
config = olympus.get_config()
send (“request”, request, request_id, retransmitted=1) to config.head
# timer for the above request
timer.start()
```

---

on receiving (result_struct)

```
if validate(result_struct.result_proof):
      timer.stop()
      pending_requests.remove(result_struct.result_proof.request_id)
      return result_struct.result to application
else:
      # send reconfiguration request to olympus with result_proof as proof of
      # misbehaviour
      send(“reconfiguration_request”, client, result_proof) to olympus
```

```
# runs periodically to check if configuration changed
on configuration check triggered (done periodically)
if config is not (updated_config = olympus.get_config()):
      config = updated_config
      for each request in pending_requests:
            execute sending request to head for request
```

```
on_receiving("replicas_public_keys", replicas_public_keys) from Olympus
replicas_public_keys = replicas_public_keys
```

```
def validate(result, result_proof):
      # crypto_hash(arg) is a crytographic hash function
      H = crypto_hash(result)
      count=0
      for replica_result_proof in result_proof:
            # check_signature validates signature
            if check_signature(replica_result_proof) and
                  replica_result_proof.hash == H:
                        count +=1
      if count < t+1
            return False
      return True
```

Replica

```
state_object = None
# mode is initlally PENDING
mode = PENDING                  # in { ACTIVE, PENDING, IMMUTABLE }
history = []                    # sequence of order proofs
result_cache = []
# cache to store result and result proof; limited size; uses FIFO strategy
timer = Timer(callback=send_reconfiguration_request)
crypto_keys = ()                # pair of public and private keys for self
replicas_public_keys = []       # public keys for all replicas
position = -1 # to distinguish between head, tail and internal replicas
last_checkpoint_proof = None    # last checkpoint proof
slot = 0                       # points to current slot in history
```

<u>on_receiving ("inithist", hist, running_state) from Olympus</u>

```
history = hist
state_object = running_state
mode = ACTIVE
```

<u>on_receiving ("key_pair", public_key, private_key) from Olympus</u>

```
crypto_keys = (public_key, private_key)
```

<u>on_receiving("replicas_public_keys", replicas_public_keys) from Olympus</u>

```
replicas_public_keys = replicas_public_keys
```

```
# only called for head replica
```

<u>on_receiving ("request", request, request_id, retransmitted=0)</u>

```
slot = slot+1
# sign is a function to sign using the private key
if signed order_statements for all lower numbered slots present and
  mode == ACTIVE and history[slot] is None:
      order_statement = sign(<order, slot, request>, crypto_keys[1])
else:
      return error

result = state_object.evaluate_request(request)
# C is the configuration which the replica obtains from olympus
order_proof = (slot,request_id,request,replica,C,[order_statement])
history.append(order_proof)
# crypto_hash is a crytographic hash function
result_proof = [sign(<"result", request_id, request, crypto_hash(result)>,
                crypto_keys[1])]
shuttle = (order_proof , result_proof)
send (shuttle) to next replica
```

on receiving (shuttle) from predecessor position-1

```
# check for preconditions
if not is_order_proof_valid(shuttle.order_proof):
      # misbehaviour - different operations at a particular slot
      send("reconfiguration_request", replica) to olympus

# sign is a function to sign using the private key to send message to
# successor
if signed order_statements for all lower numbered slots present:
      order_statement = sign(<"order", slot, request>, crypto_keys[1])
else:
      send("reconfiguration_request", replica) to olympus
# We trigger a reconfiguration_request because if we ignore the request,
# client will retransmit, head has already ordered the operation and hence it #
will start a timer and wait, once timer expires reconfiguration will be
# triggered. So instead of delaying it, we trigger it right away.

request_id = shuttle.order_proof.request_id
result = state_object.evaluate_request(request)
shuttle.order_proof.order_stmt_list.append(order_statement)
shuttle.result_proof.append(sign(<"result", request_id, operation,
       crypto_hash(result)>), , crypto_keys[1])
history.append(shuttle.order_proof)

if position != 2t: # not tail
      send (shuttle) to next replica
else:
      send (result, shuttle.result_proof) to client
      result_cache[request_id] =
             (result, shuttle.result_proof)
      # send result shuttle back in the chain
      send (shuttle, result) to previous replica
```

on receiving (shuttle, result) from successor position+1

```
request_id = shuttle.result_proof.request_id
result_cache[request_id] = (result, shuttle.result_proof)
send (shuttle) to previous replica
# if timer is running for the request_id stop the timer and send the request
# to client
if timer(request_id):
      timer.stop()
      send (result_cache[request_id]) to client
```

```
on receiving (“request”, request, request_id, retransmitted=1)
if is_correct(replica) and result_cache.contains(request_id):
      send (result_cache[request_id]) to client
else if mode == IMMUTABLE:
      return error
else if position != 0: # not head
      send (“request”, request, request_id, retransmitted=1) to head
      timer.start(callback=send_reconfiguration_request)
else :
      # Head has seen the request and waiting for result proof from replicas
      if is_request_pending(request_id):
            timer.start(callback=send_reconfiguration_request)
      # treat it as a new request
      else:
            timer.start(callback=send_reconfiguration_request)
            execute on receiving (“request”, request, request_id,
                     retransmitted=0)
```

```
on_receiving (“wedge_request”) from Olympus
mode = IMMUTABLE
# send history along with last checkpoint proof to olympus
send(“wedged_statement”, sign(<history, last_checkpoint_proof>,
                          crypto_keys[1]) to olympus
```

```
on receiving (“catch_up”, missing_operations) from Olympus
for each operation in missing_operations:
      state_object.evaluate_request(operation)
send(“caught_up”, crypto_hash(state_object)) to olympus
```

```
on receiving (“get_running_state”) from Olympus
send (“running_state”, state_object) to olympus
```

```
# Checkpointing executed at head
on checkpointing triggered (done periodically after N new slots in history)
# state_objectis the running state
checkpoint_statement = sign(<”checkpoint”, crypto_hash(state_object)>,
                         crypto_keys[1])
checkpoint_proof_shuttle = [checkpoint_statement]
send (checkpoint_proof_shuttle, num_slots) to next replica
```

---

on receiving (checkpoint_proof_shuttle, num_slots) from predecessor

```
checkpoint_statement = sign(<"checkpoint", crypto_hash(state_object)>,
            crypto_keys[1])
checkpoint_proof_shuttle.append(checkpoint_statement)

# at tail, checkpoint proof is completed and sent back in the chain
if position == 2t:
      completed_checkpoint_proof = checkpoint_proof_shuttle
      if are_signatures_valid(completed_checkpoint_proof):
            # truncates oldest num_slots from history
            history.truncate(num_slots)
            # save last_checkpoint_proof
            last_checkpoint_proof = completed_checkpoint_proof

      send (completed_checkpoint_proof, num_slots) to predecessor replica

# if not tail, send checkpoint shuttle forward in the chain
else:
      send (checkpoint_proof_shuttle, num_slots) to next replica
```

---

on receiving (completed_checkpoint_proof, num_slots) from successor

```
# everybody in chain has received checkpoint msg, start truncating
if are_signatures_valid(completed_checkpoint_proof):
      # truncates oldest num_slots from history
      history.truncate(num_slots)
      # save last_checkpoint_proof
      last_checkpoint_proof = completed_checkpoint_proof

if pos != 0: # not head
      send (checkpoint_proof_shuttle, num_slots) to predecessor replica
```

```
# callback for timer expire
def send_reconfiguration_request():
      send ("reconfiguration_request", replica) to olympus
```

```
def are_signatures_valid(proof):
      for each rho, order_statement in proof:
            key = replicas_public_keys[rho]
            if not check_signature(order_statement, key)
                  return False

      return True
```

```
# Checks for valid signatures and operation in order proof
def is_order_proof_valid(order_proof):
	operation = order_proof.operation
	for each rho, order_statement in order_proof:
		key = replicas_public_keys[rho]
		if not check_signature(order_statement, key) or
			order_statement.operation is not operation:
			return False
	return True
```

```
def is_correct(replica):
	return replica.mode in (PENDING, ACTIVE, IMMUTABLE) and
		is_history_valid(replica.history)
```

```
def is_history_valid(hist):
	for each slot in hist:
		if not all(hist.order_proofs.operation) == hist.operation:
			return False
	return True

# checks if request is received and result is not available yet to be sent to
# client because result shuttle hasn’t been received
```

```
def is_request_pending(request_id):
	if request_id in any(history.order_proof) and
		request_id not in result_cache:
			return True
	return False
```

Olympus

```
# C is current configuration, C.head used in pseudocode code is C[0],
# C.tail is C[-1]...
C = [replica identifiers ...]
clients = [client identifiers ...] # list of clients
keys = [] #list of tuples that stores public and private keys for each replica
```

---

on receiving (“reconfiguration_request”, sender, proof_of_misbehaviour=None)

```
# if sender is not in current configuration and the proof of misbehaviour is
# invalid, ignore
if sender not in C and (proof_of_misbehaviour == None or not
  check_proof_of_misbehaviour(proof_of_misbehaviour)):
      return  # ignore request

for each replica in C:
      send(“wedge_request”) to replica

# receiving wedged statements
await atleast t+1 consistent wedged statements from replicas and store in
wedged_statements

valid_quorum = False
while not valid_quorum:

      quorum_candidate = select any t+1 wedged_statements

      # Q is quorum
      # condition for consistency: for every pair of wedged statements,
      # w1, w2 for each slot operation should be same in w1.history and
      # w2.history. Signature of the messages should be valid. Also
      # wedged_statement.checkpoint should be valid for quorum of replicas.
      if is_history_consistent(quorum_candidate):
            Q = quorum_candidate
      else:
            continue

      # longest history - most number of order proofs
      LH = select longest history order proof from Q
      for each replica in Q:
            send(“catch_up”, LH-replica.wedged_statement.history) to replica

      await |Q| “caught_up” messages from replicas in caught_up_messages
      valid_quorum = True
```

```
      # check if hash in caught_up_messages are same
      hash_val = any(caught_up_msgs.hash)
      for each msg in caught_up_msgs:
            if not msg.hash == hash_val:
                  valid_quorum = False
                  break

for each replica in Q:
      send("get_running state") to replica
      await "running_state" message from replica into running_state
      if(crypto_hash(running_state) == hash_val):
            break

# start 2t+1 new replicas in new configuration C'
for i in xrange(2t+1):
      # replica_i is identifier for replica
      replica = Replica(replica_i)
      replica.start()
      C'.append(replica)

for each replica in C':
      # the new history on the replica will be empty
      new_hist = []
      send ("inithist", new_hist, running_state) to replica
      # generate_keys generates public key cryptography keys
      (public_key, private_key) = generate_keys(replica)
      send ("key_pair", public_key, private_key) to replica
      keys[replica.replica_id] = (public_key, private_key)

# send all public keys of others to all replicas and clients
replicas_public_keys = get all public_keys from keys
for each replica in C':
      send ("replicas_public_keys", replicas_public_keys) to replica
for each client in clients:
      send ("replicas_public_keys", replicas_public_keys) to client
C = C'
```

```
def check_proof_of_misbehaviour(result_proof):
      H = result_proof[0].hash
      # if signature is not valid or hash of any result proof mismatach it
      # indicates a proof of misbehaviour
      for replica_result_proof in result_proof:
            if not (check_signature(replica_result_proof) and
                  replica_result_proof.hash == H):
                  return False

      return True
```

```
def is_history_consistent(wedged_statements):
      # check consistency between every pair of wedged statements
      for every w1 in wedged_statements:
            for every w2 in wedged_statements:
                  if w1 == w2:
                        continue
                  for slot in range(0, min(w1.history.length,
                    w2.history.length)):
                        if w1.history[slot].request !=
                          w2.history[slot].request:
                              return False

      # check validity of checkpoint proof
      hash_val = wedged_statements[0].last_checkpoint_proof.hash
      for each last_checkpoint_proof in wedged_statements:
            if last_checkpoint_proof.hash != hash_val
                  return False

      return True
```

```
def get_config():
      return C
```
