Byzantine chain replication (BCR)

Pseudocode

Byzantine- Chain Replication (BCR) is a generic approach to make applications tolerate arbitrary faults beyond crash failures in an asynchronous environment. BCR algorithm tolerates a maximum of "t" failures of worst kind called byzantine failures. It constitutes a client performing operations at random, Olympus is the configuration service and chain of replicas which perform same set of of operations and maintain similar running states.

Pseudocode defined below has a Client requesting operations to be executed at the chain of replicas and the configuration of replicas is real time managed by Olympus.

Symbols Used:

$\mathbf{p_i}$: replica at i^{th} position in the configuration

$\mathbf{C_i}$: current configuration (chain of replicas)

S : Slot number

o : operation

delta : cryptographic hash

Shuttle: tuple of order_proof (s,o)p_i and result_proof (result,o,delta(r))p

1. Client: *#operations that take place at client side*

1. request_sequence = **getRequestSequenceFromConfig()** *#returns a sequence of (**operation o,requestId)***
2. set client.timer = **Timer.start()**$_{\mathbf{requestId}}$ *# Timer corresponding to request Id is initiated, maintains bound on turn around time.*
3. **client** sends configuration request to **Olympus**

   (replica_sequence, replica_keys) = **getCurrentConfigFromOlympus**(); *# returns a current replica configuration if there exists one else returns a new configuration.*

#After execution of section 1 involving client operations, olympus ***continued below.***

2. Olympus: On receiving **getCurrentConfigFromOlympus()**

1. **If**(configuration does not exists):
   a. Generate a configuration Ci with p replicas
   b. **for** all replica **in** Configuration Ci

      inihist(); #inithist is the valid history of operation per slot | running state of replica. Empty initially.

      replica_key = generate_Key(replica):

      *#replica_key.**public** = public_key of replica*

      *#replica_key.**private** = private_key of replica*

   c. Broadcast corresponding public keys to all the replicas.
2. **else return** (Ci(configuration)) to the client.

#After execution of section 2 involving Olympus,Client operations continued below.

3.Client : On receiving (Ci(configuration)):

1. From the Ci(p1,p2...pn), Fetch the **head_replica**. *# where Ci is the current configuration*

   received_result (**order_proof, result_proof**) = **send_operations_to_replica**(operation, *head)

   *# for **send_operations_to_replica** goto replica section 4.*

   *# *head is the pointer to the head replica*

   Where:

   order_proof** is signed order statement **(s,o)**p_i for all p_i belongs to **C.

   ***result_proof** is signed tuple of **(result,o,delta(r))**p_i**). delta(r)),** delta(r) is encrypted replica result.*

Case: fault-free

i. **if** (**hash**(result(r)) **==delta(r)**) *# hash is applied by client on result to verify if delta(result) from replica matches*

   "successfully executed operation"

   else # (**hash**(result(r)) **!=delta(r)**) *hash result did not match the result returned by replica,this constitutes as **proof of misbehaviour, hence** reconfiguration request sent to Olympus*

   reconfigure_replicas(proof_of_misbehaviour) ***#declared in the end***

Case: failure.

if (client.timer == expired): *# timer was initiated in step 2.*

a. received_result = **broadcast_operations**(operation(o), replica_Sequence)

*#**broadcast_operations** retransmits request to all the replicas. Function declared below under replica(faulty case)**:***

*#received_result captures the result of all replicas, once client **retransmit** operation to all replicas in current config.*

i. **if (**received_result == **ERROR):** *# this means that the replica is immutable*

- Client requests latest configuration from Olympus.

  replica_sequence = **getCurrentConfigFromOlympus**();

  # returns a current replica configuration if there exists one else returns a new configuration.

- Client executes request protocol from scratch. *#from section 3*

ii. **elseif** (received_result == result(r))

if (**hash**(result(r)) **==delta(r)**) *# replica has the result cached, client receives it and verifies*

"successfully executed operation"

else if (**hash**(result(r)) **!=delta(r)):** *# hash result did not match the result returned by replica,this constitutes as **proof of misbehaviour, hence** reconfiguration request sent to Olympus*

reconfigure_replicas(proof_of_misbehaviour) ***#declared in end***

iii. elseif(received_result = Error and result(r)) *# some replica returned **Error**, some returned **result(r)***

if (**hash**(result(r)) **==delta(r)**):

"successfully executed operation"

else if (**hash**(result(r)) **!=delta(r)):**

reconfigure_replicas(proof_of_misbehaviour)

4. Replica : *#on receiving **send_operations_to_replica**(operation, *head) from client*

1. p_{head} orders the operations and assign slot(s) number.
2. **for** p_i in (Ci(configuration)): *# iterating from **head to tail** request protocol goes through every replica in configuration, executes the operation and updates the **order_proof**, **result_proof***

   if(p_i == **head**):

   a. create **shuttle** containing **((s,o)p_i, (result,o,delta(r))p_i)** # head replica creates a **shuttle** containing **order_proof (s,o)p_i** and **result_proof (result,o,delta(r))p_i.**

   b. Increment **checkpointing_counter:** *#counter to implement checkpointing*

   if(**checkpointing_counter** mod **N** == 0): *#when checkpointing counter is a multiple of N checkpoint shuttle is along the chain.*

   Checkpointing(); *# the function handles checkpoint algorithm and is declared towards the end.*

   else:

   for all **p_i** in replica_sequence: *#starting from head to tail*

   i. **for** all **$p_j < p_i$** in replica_sequence**:**

   a. **Validate order_proof (<order,s,o>)p_j** *# for all predecessor replica **p_j**, **(s,o)p_j** should belong to **(<order,s,o>)p_i***

   b. **execute** operation **o** and obtain **result r**.

   ii. add its **(s,o)p_i** to the **p_i.history** which is a list of **order_proof**.

   iii. add its result **(result,o,delta(r))p_i** to its **result_proof**.

   iv. if (**p_i** == **tail**): *# if shuttle reaches the tail replica*

   return **result_proof** (result,o,delta(r))pi to client.

3. **for** p_i in (Ci(configuration)): *#**iterating from tail to head** replica.*

   if (p_i == **tail**):

   a. Tail forward the **result shuttle ((s,o)p_i, (result,o,delta(r))p_i)** back to predecessor replicas.

   b. **if**(p_i.timer$_{\textbf{requestId}}$ is active): *#timer against requestId was started*

 # this is the case where timeout occurs at client, operation is broadcasted to replicas, if operation is not recognized *protocol is executed from scratch. Cancel the timer in case result_shuttle is received before timeout. Otherwise send a reconfiguration request to olympus.*

if p_i receives the **result shuttle**: *#in case of retransmission and timer against the received request ID is active*

- **cancel the timer**
- **cache** *result_proof* = ***(result,o,delta(r))*** p_i
- **return** *result_proof* *#to the client*

for any p_i **in replica_sequence :**

- **if** (p_i timer ==expire):

  reconfigure_replicas() #send **reconfiguration** request to olympus. **Function declared** towards the end.

c. **cache**(key=**requestId**, value=**result_shuttle**) *# cache the result replica with key as requestId and value as the result_replica*

else: *# replicas other than tail*

cache(key=**requestId**, value=**result_shuttle**) *#cache the result*

4. Replica : in case of timeout failure **: broadcast_operations** received from **client**

 # case when timeout happens at client

*# **broadcast_operations** called from section 3*

broadcast_operations(operation(o), replica_sequence): *# where o is operation*

for all p_i in **replica_sequence**:

if (p_i.**mode** = **Immutable**)

return "**Error**" statement.

if (p_i.**mode** = **Active**)

if (**cached_result**(requestId) **exists**) # cached result is present against the requestId

return *result_proof* = ***(result,o,delta(r))*** p_i *# returns cached result*

else *# if **no cached result** found*

- p_i.timer = Timer.start()$_{o.requestId}$. # starting timer corresponding to requestid of o at replica meanwhile request forwarded to head
- pass the request to p_{head}.
- p_{head} upon receiving the request:
  1. **if** (**cached_result**(requestId) == exists):

     return ***(result,o,delta(r))*** p_{head} # this is the **result_proof**
  2. **elseif** (**operation(o)** is processing and **result_shuttle** yet to receive))

P_{head}.timer = Timer.start()$_{requestId}$ #starting timer at head replica corresponding to the requestId of the operation.

if(result **shuttle** is received **before** timer p_{head}.timer expires):

i. p_i.timer = Timer.cancel() #cancel timer at replica

ii. **cache** the result_proof

iii. **return** ***(result,o,delta(r))***p_{head} # this is the result_proof

else: (P_{head}.timer == expired):

reconfigure_replicas(proof_of_misbehaviour)

3. **elseif (operation o** is not recognized):

send_operations_to_replica(operation, *head)()* # starts the protocol from scratch**, goto section 4 Replica**

<u>4. Replica</u>:

Case : when order proof not validated | peer faulty | timeout at client

1. If (**p_i** finds peer faulty || **order_proof** not validated)

   i. Send wedge statement**(<wedge, h>p_i**) to **Olympus** *#h denotes the history which contains order proofs*

   ii. **P_i.mode=Immutable** # *The replica changes it's state to immutable, i.e. it can't issue new order statements*

Olympus side:

On receiving wedge statement**(<wedge, h>p_i)**

Broadcasts **wedge** request to all replicas *#broadcast wedge request to all replicas requesting their history*

Replica side:

On Receiving wedge request from Olympus

for all replica **p in** (Ci(configuration)):

Send wedge statement and order proof ***(<wedge, h>p_i*** & ***<order,s,r>p_i***) to **Olympus**

Olympus Side:

On receiving ***(<wedge, h>p_i & <order,s,r>p_i) statement from replicas***

#the replicas in the current config will have at least 1 honest replica among max T failures

for all **replicas** check

if(combination of (**<wedge, h>p_i**) wedge statement and **<order,s,r> p_i statement** does not includes a order_proof for (s,o) *) # this concludes as proof of misbehaviour and reconfiguration function is called*

reconfigure_replicas(proof_of_misbehaviour) ***#declared in the end***

<u>**reconfigure_replicas():**</u> ***#reconfiguration algorithm***

Olympus: On receiving **(<wedge, h>p_i)$_{Ci}$** from all the replicas

1. let Q be quorum of replica in **C_i** with valid histories
2. for every pair **p_i and p_j in Q**
   a. **if** there exists a **<slot,operation1>** in **$p_{i.history}$** and **<slot,operation2`>** in **$p_{j.history}$**
      if (operation1 == operation2)**:**
         Then **slot history** is **consistent**
      else: Choose different Quorum and **goto step-1** #(repeat ***reconfiguration algorithm)***
   b. **LH = longest** of the **<slot, operation> pairs** from all replica corresponding to a slot number #LH is longest history
3. **for** all replica(p) in Q:
   a. (catch _up)**p_i** = (**LH - p_i .history**) #suffix of LH that **p_i** has not executed yet
   b. send (catch _up)**p_i** to replica in Q #olympus sends catch up message to replica

On Replica Side:

4. for all replica **p_i** on receiving (catch _up)**p_i** message
   a. **p_i** executes operations in (catch_up)**p_i**
   b. **ch** = **delta**(running_state)**$p_{i,}$** **#ch** is the **cryptographic hash** of **$p_{i's}$** running state S
   c. send(caught_up(ch)) to **Olympus**

On Olympus side:

a. after receiving "caught_up" from all replicas in Q
   for replica in **quorum Q**:
      if(replica "caught_up" != **ch**) # checking if any replica "caught_up" value is not **ch**
         choose different Quorum and **goto Step-1** #(repeat ***reconfiguration algorithm)***

else #all replicas have "caught_up" value = **Ch**

i. Send **"get_running_State"** message to any replica in Q

\# **replica** returns its **running state S**

if(**delta**(S)!= ch)

Request running_State from another replica in Q

else:

i. Generate a configuration Cj with p replicas

ii. **for** all replica **in** Configuration Cj

inithist(S); **# S** is the current running state of the previous configuration

*On receiving **Checkpointing()** from replica in section 4.*

<u>Checkpointing():</u> ***#(at Nth operation)***

while(checkpointing_counter mod N == 0): *# function is called after every N operations*

1. **for** all p_i in **replica_sequence**: *#iterating from **head replica to tail** replica.*
    a. **if** (p_i == head):

        initiate a shuttle containing the **checkpoint and a running state.**

        (checkpoint, delta(running_state))p_i *# checkpoint_proof tuple*

    b. **else**:

        add a **(checkpoint,delta(state))**p_i to the **checkpoint_proof**.
2. **for** all p_i in **replica_sequence**: *#iterating **tail replica to head** replica.*
    - truncate **prefix from** p_i**.history**
    - **add a checkpoint in history** corresponding to deleted history.

        p_i***.history*** becomes ((s, o, p_i, C, (checkpoint,order_proof)p_i)

        # once checkpoint_shuttle is received remove the prefix from history until the checkpoint.
    - **return the shuttle** to the next replica in sequence towards head.

<u>Bibliography</u>

1. Byzantine Chain Replication research paper by *Van Renesse, Chi Ho, and Nicolas Schiper.*
2. Project.txt document provided by *Prof. Scott Stoller*