CSE535: Distributed Systems (Fall 2021)

Scott D. Stoller, Stony Brook University

Phase 3 : pseudo-code

Team : PreeManiSai : Manikanta Sathwik Yeluri, Preetham Kumar Reddy Katta, Sai Bhavana Ambati

Test Generator

Design :

i) the generator receives the following inputs :

n_replicas : Total number of actual replicas

n_twins : Total number of twins

n_rounds : Total number of rounds

n_partitions : Total number of partitions in each round

is_leader_faulty : If the leader of the round can be one of the twins or not

partition_num_limit : Maximum number of partition combinations

n_test_cases : Total number of test cases

leader_partitions_num_limit : Maximum number of leader-partition combinations

random_seed : seed for generation of random numbers

ii) The generator first generates a list of all possible combinations of partitions. The partition generation algorithm can also generate combinations in which replica is present in multiple combinations.

ii) It then creates two lists : One that contains all partition combinations in which most of the partitions have more than 2n_twins + 1 replicas (to get a quorum with high probability) and another list in which one or none of the partitions have more than 2n_twins + 1 replicas (to get a quorum with low probability). The total size of the 2 lists is limited to partition_num_limit.

Example of each element of the lists :

['replica_1', 'replica_2','replica_3'],['replica_4', 'replica_5'],['replica_1f', 'replica_2f']

replica_1 and replica_1f are twins. replica_2 and replica_2f are twins.

iii) Then leader and partition combinations are generated for each of the above two lists and are stored in two separate lists respectively. The twin leader is present or absent based on is_leader_faulty.

Example of each element of the lists :

['replica_1', [['replica_1','replica_2','replica_3'],[replica_4, replica_5],[replica_1f replica_2f']]] where replica_1 is the leader

iv) To generate every test case, a random number number between 0 and n_rounds - 3 is generated and assigned to num_non_quorum_rounds. This decides that number of partition-leader combinations to select from list with high probability of getting a quorum and low probability of getting a quorum. The test case contains num_non_quorum_rounds elements from the list with low probability of getting a quorum and n_round - num_non_quorum_rounds elements from the list with high probability of getting a quorum. Then, random pairs of replicas are generated for message drops/delays for various message types. This is stored in a dictionary with key as the source and value as the list of destinations. Each test case is checked for liveness by determining if all replicas can get a quorum in atleast 3 consecutive rounds. If it is a valid test case, it is added to a file, if not, it is discarded.

v) The file is given to the test executor.

Pseudo-code :

```
Function test_generator(n_replicas, n_twins, n_rounds, n_partitions,
is_leader_faulty, partition_num_limit, n_test_cases, leader_partitions_num_limit, random_seed):
  all_replicas = list('replica_1', 'replica_2', .. 'replica_n', 'replica_1f',
  'replica_2f'.. )
   initialize global partitions_list
  // replica_x and replica_xf are twins ..

  partition_generation_algorithm (n_replicas + n_twins, n_partitions)
  adjustPartitions(n_replicas, n_twins, n_partitions)
  // Sample output of partition_generation_algorithm
   // [1,2,3][4,5][6,7]
  // 1,2,3 belong to 1 partition and 4,5 belong to another
  limit partitions_list to size partition_enum_limit using random sampling
  (or by taking first partition_enum_limit elements)

  // Sample output of the loop below
  // "replica_1 replica_2 replica_3 | replica_4 replica_5 | replica_1f replica_2f "
  // 1,2,3 belong to 1 partition and 4,5 belong to another
  // replica_1f and replica_2f are faulty replicas (n_replicas = 5 and n_twins = 2)

  for each number in each partition combination:
    replace number n with 'replica_{n}' if n <= n_replicas
    replace number n with 'replica_{n}f' if otherwise

  // The loop below creates a list 'low_partition_list' that contains partition combinations
  in which a quorum cannot be obtained

  initialize low_partition_list
  for each partition in each partition combination :
    if count of replicas in partition < 2*n_twins + 1:
        delete partition combination from partitions_list
        add partition combination to  low_partition_list

  if partitions_list is empty :
    for each partition combination in low_partition_list :
        if only one partition has less more than 2*n_twins + 1 processes  :
            delete partition combination from low_partitions_list
            add partition combination to partition_list

  limit low_partitions_list +  partitions_list to size partition_enum_limit using random
  sampling (or by taking the first partition_enum_limit elements)


  // Sample output of the loop below
  // ['replica_1' , 'replica_1','replica_2','replica_3' ][' replica_4',' replica_5'] ,
  ['replica_6', 'replica_7']]
  replica_1 is the leader
  initialize partition_leader_combination
```

```
for i ranging from (n_twins + 1) to n_replicas :
  for j ranging from 1 to partitions_list.size :
      partition_leader_combination.add('replica_{i} : partitions_list[j]')


if is_leader_faulty == True:
     for i ranging from 1 to n_twins :
          for j ranging from 1 to partitions_list.size :
              partition_leader_combination.add('replica_{i}f : partitions_list[j]')
              partition_leader_combination.add('replica_{i} : partitions_list[j]')

initialize low_partition_leader_combination
for i ranging from (n_twins + 1) to n_replicas :
  for j ranging from 1 to partitions_list.size :
      low_partition_leader_combination.add('replica_{i} : low_partitions_list[j]')


if is_leader_faulty == True:
     for i ranging from 1 to n_twins :
          for j ranging from 1 to low_partitions_list.size :
              low_partition_leader_combination.add('replica_{i}f : low_partitions_list[j]')
              low_partition_leader_combination.add('replica_{i} : low_partitions_list[j]')


limit low_partition_leader_combination +  partition_leader_combination to size
leader_partitions_num_limit using random sampling  (or by taking the first
leader_partitions_num_limit elements)
i = 0
while i < n_test_cases :

  // refer note_1
  initialize test_case_data
  choose any value between 0 to n_rounds/2
  non_quorum_rounds = random.choice(0,n_rounds/2)

  low_current_round_combination = choose any non_quorum_rounds values
  from low_partition_leader_combination

  current_round_combination = choose any n_rounds - non_quorum_rounds values
  from partition_leader_combination

  // Sample output of the loop below
  /*

  "rounds":
      "0":
          "leader": "replica_1",
          "partitions": [
              [
                  "replica_0", "replica_1","replica_2","replica_0f"
              ],
              [
```

```
                "replica_3", "replica_4"
            ]
        ],
        "messageType": 2,
        "failType": 2,
        "src_to_dest":
            "replica_2": [
                "replica_0f"
            ],
            "replica_0f": [
                "replica_0f"
            ]

    ,
    "1":
        "leader": "replica_1",
        "partitions": [
            [
                "replica_0", "replica_4", "replica_0f"
            ],
            [
                "replica_1", "replica_2", "replica_3"
            ]
        ],
        "messageType": 3,
        "failType": 1,
        "src_to_dest":
            "replica_1": [
                "replica_2"
            ],
            "replica_2": [
                "replica_1"
            ]

    ,


"n_replicas": 5,
"n_twins": 1,
"n_rounds": 5

*/
i = 0
while i < n_test_cases :
        test_case_data = {}
        non_quorum_rounds = random number in range (0,n_rounds - 3)
        quorum_rounds =   n_rounds - non_quorum_rounds
        low_current_round_combination = choose non_quorum_rounds vals from
        low_partition_leader_combination
        current_round_combination =  choose quorum_rounds vals from
        partition_leader_combination
        test_case_data['rounds'] =
```

```
        for j in range(0,non_quorum_rounds) :
            test_case_data['rounds'][j + 1] =
            test_case_data['rounds'][j + 1]['leader'] =
            low_current_round_combination[j][0]
            test_case_data['rounds'][j + 1]['partitions'] =
            low_current_round_combination[j][1]
            test_case_data['rounds'][j + 1]['messageType'] = random number in range (0,2)
            test_case_data['rounds'][j + 1]['failType'] = random number in range (0,2)
            test_case_data['rounds'][j + 1]['src_to_dest'] =
            get_src_dest_combs(low_current_round_combination[j])

        for j in range(non_quorum_rounds, n_rounds) :
            test_case_data['rounds'][j + 1] = {}
            test_case_data['rounds'][j + 1]['leader'] =
            current_round_combination[j - non_quorum_rounds][0]
            test_case_data['rounds'][j + 1]['partitions'] =
            current_round_combination[j - non_quorum_rounds][1]
            test_case_data['rounds'][j + 1]['messageType'] = random number in range (0,2)
            test_case_data['rounds'][j + 1]['failType'] = random number in range (0,2)
            test_case_data['rounds'][j + 1]['src_to_dest'] =
            get_src_dest_combs(current_round_combination[j - non_quorum_rounds])


        is_valid_test_case = is_valid_test(test_case_data, n_replicas,n_twins, n_rounds )
        if is_valid_test_case :
            print("live")
            live = live + 1
            test_case_data['n_replicas'] = n_replicas
            test_case_data['n_twins'] = n_twins
            test_case_data['n_rounds'] = n_rounds
            with open('tests/test_case_' + str(i) + '.json', 'w', encoding='utf-8') as f:
                json.dump(test_case_data, f, ensure_ascii=False, indent=4)
            i = i + 1
        else :
            print("not live")
            no_live = no_live + 1
```

```
Function powerset( s):
        x = s.size
        initialize subset_list
        for i in range(1 << x):
            subset_list.add([s[j] for j in range(x) if (i & (1 << j))])

// To add cases where replicas are in multiple lists
Function adjustPartitions(n_replicas, n_twins, n_partitions) :
        replica_list = list('replica_1', 'replica_2', .. 'replica_n', 'replica_1f',
```

```
        'replica_2f'.. )
        subset_list = powerset(replica_list) // get all subsets
        visited = '{partition_combination_1}' : False, '{partition_combination_1}' : False  ...
        for i in range(global partitions_list.size) :
            for j in range(subset_list.size) :
                new_partition_list =  self.partitions_list[i]
                new_partition_list[0].extend(subset_list[j])
                new_partition_list[0] =  unique(new_partition_list[0]) // any replica
                occurs only once
                if new_partition_list[0].size  > partitions_list[i][0].size and
                new_partition_list[0] not in visited  :
                        partitions_list.append(new_partition_list)
                        visited[new_partition_list[0].toString()] = True

//get various source to destination combinations
Function get_src_dest_combs(self, partition_combination) :
        leader = partition_combination[0]
        initialize partition
        for i in range(partition_combination[01].size) :
            found = False
            for j in range(partition_combination[1][i].size) :
                if leader == partition_combination[1][i][j] :
                    partition = partition_combination[1][i]
                    found = True
                    break
            if found :
                break

        srcs = randomly choose partition 2*partition.size elements  from partition
        initialize dictionary src_to_dests
        dests = randomly choose partition 2*partition.size elements  from partition
        for i in range(len(srcs)) :
            choice = choose random number between 0 to 2
            if choice == 0 or choice == 1 :
                src_to_dests[srcs[i]].append(dests[i])
                src_to_dests[srcs[i]] = list(set(src_to_dests[srcs[i]]))
        return src_to_dests



Function is_valid_test(test_data) :
    is_quorum = {}
    set is_quorum[replica_id] to false for all replica_ids
    set quorum_rounds[replica_id] to [] for all replica_ids
    for round in  test_data :
        leader = get leader of the round
        partition_list = list of partitions

        if partition containing leader has replicas > 2*n_twins + 1 :
                        failType = test_data['rounds'][round]['failType']
                        src_to_dest  = test_data['rounds'][round]['src_to_dest']
                        for each src, dest in src_to_dest :
```

```
                        if src, dest in partition :
                            reduce length of partition
                    if new length of partition  > 2*n_twins + 1  :
                        add round_no to quorum_rounds[replica_id] for all replicas
                        in that partition


    for replica_id in quorum_rounds :
        if quorum_rounds[replica_id] contains any 3 consecutive rounds :
            is_quorum[replica_id] = True

    if all replica_ids are true in is_quorum :
        return True
    else :
        return False



Function partition_generation_algorithm(n, k) :
    create 2D array results[k][n]
    create list of strings answer
    solution(1, n, k, 0, results, answer)
    // answer contains list of all possible combinations of partitions
    // each row is a new combination
    // in each combination, "|" separates each partitions
    return answer


Function solution(i, n, k, nums, results, answer) :
    if i > n :
        if nums == k :
            answer_str = ""
            for set in result :
                answer_str = answer_str + set.toString + " | "
            answer.add(answer_str)
        return
    for j ranging from 0 to results.size :
        results[j].add(i)
        if results[j].size > 0 :
            solution(i + 1, n, k, nums, ans)
            ans[j].removeLastElem()
        else :
            solution(i + 1, n, k, nums + 1, ans)
            ans[j].removeLastElem()
            break
```

note_1 : Algorithm for choosing n_values from list :
referring to link : https://stackoverflow.com/questions/12548312/find-all-subsets-of-length-k-in-an-array

How is liveness ensured ?

First creating to lists of *leader_partition_combinations* and *low_partition_leader_combination*. In

low_leader_partition_combinations, atleast some of the replicas contain can get a quorum whereas in *leader_partition_combinations* most of the replicas contain can get a quorum.

Every time a test case is generated, a function called *is_valid_test_case* is called that checks if all replicas can get a quorum atleast once. If it is not possible, the test case is discarded, else, it is written to the file

Test Executor Pseudo-code

Design :

a) For each test case, create the required number of process : $n\_replica + n\_twins$.

b) Ensure that twins have the same public and private keys

c) create a global list mapping each process_id with each replica_id and also the other way round. Here, replica_id is of the form 'replica_{x}' or 'replica_{x}f' and process_id is the ID assigned to the replicas by the distalgo library while creating the process

d) create a new process called "Playground" . It has all the information about test case, and it can differentiate twins (replicas can't). Every replica interacts with another only via the "Playground". For this, the send and receive functions of the replicas are modified to only send messages to the hub. If a replica wants to broadcast, it only sends the message to the playground, whereas, if it wants to send a message to only one other replica, it sends the id of intended destination to the playground, so the playground can decide whether it wants to forward it or not. The leader election algorithm of the replicas is also modified by getting the leader from the playground instead. It gives it's own round_no to the playground, so the playground can decide and return the id of the leader.

e) The playground class does the following things :

i) It continuously listens to all messages it receives, for every message, it checks if it is a broadcast message or uni-cast message. It does this by checking if the current message is a Vote messages or not. If it is a vote message, it passes the control to a handler to decide if the message should to dropped or forwarded to the intended receiver. This is determined by checking the partition in which the sender belongs for the current round. if any of the partitions that the sender belongs to has the receiver too, it checks if the message should be dropped or delayed by referring to the 'src_to_tests' attribute . It also checks if the twin of the process is in the partition (if the original is not in the partition).If the twin is present, it forwards the message to the twin instead. If the playground receives a broadcast message, it gets the list of all replicas from all partitions containing the sender. It removes replicas that are in the destination of the sender in 'src_to_tests' attribute. It forwards the message to all remaining replicas and delays messages to replicas in the destination of the sender in 'src_to_tests' attribute . The data structures containing mappings between each process_id and each replica_id are used for this purpose.

ii) The playground also keeps track of the current leader, round and partition list. It updates the values when receiving the first message with a new round. We are assuming that the replicas are synced and hence will be in the same round most of the time.

iii) Twins are two processes with same names (*replica_id*), private and public keys. With broadcast messages we send the message to all the replicas in the sender partition including the twins if any. If vote messages are sent to a replica which has twin, we send that message to the twin which exists in the same partition as sender and if both the twins are in same partition we check the block id in vote message and send that vote message to twin which proposed that block. We store the information of a replica and the id of block it proposed to differentiate the twins for vote messages.

iv) **Property checking :** i) **Safety :** At the end of each test case, it checks if safety is violated by checking if every transaction is there in at least $2 * f + 1$ replicas and the order of transaction is same among all the ledgers. ii) **Liveness :** At the end of each test case, the check for liveness violation is performed by checking if atleast 2f+1 quorum of ledgers has at least 1 commit i.e are live.

iii) **Online vs Offline :** For online, the replicas should be polled to check for violations which helps in detecting issues at an earlier stage but this takes more resources for computing and is less efficient . For offline, the logs can be checked at the end to look for errors. This uses less resources, but issues are detected at a late, so the the fixes occur at a later stage.

Pseudo-code :

```
Function test_executor_main(n_replicas, n_twins, n_rounds, n_partitions):
  all_replicas = list('replica_1', 'replica_2', .. 'replica_n', 'replica_1f',
  'replica_2f'.. )
  #creates all_replicas set of process with ids as input

  replica_processes  = {}
  replica_ids = {}
 // replica_x and replica_xf are twins ..

  for i,test_case_data in enumurate(file) : // file generated by test generator :
        create new folder  test_i for test case logs
        initialize replica_pub_keys
        initialize replica_private_keys
        initialize twin_replica_pub_keys
        initialize twin_replica_private_keys
        for replica in all_replicas :
            'replica' is the process ID and replica_id is of
            the form 'replica_{x}' or 'replica_{x}f'
            replica_processes[replica.replica_id] = replica
            replica_ids[replica] = replica.replica_id
            if replica is of the form 'replica_{x}f' and not 'replica_{x}' :
                actual_replica = 'replica_{x}f' - 'f'
                twin_replica_pub_keys[replica] = replica_pub_keys[actual_replica]
                twin_replica_private_keys[replica] = replica_private_keys[actual_replica]
                replica.replica_id = actual_replica
            else :
                generate private_key, public_key
                replica_pub_keys[replica] = public_key
                replica_private_keys[replica] = private_key


      create n_replicas  + n_twins instances of 'replica' class
      for each replica :
        set data with replica_pub_keys, private_key of replica, replica_id etc
        //same as in existing code

      create list rounds where rounds[i] = [(leader, partitions in round 1), ..
      (leader, partitions in round n_rounds)]

      create 1 instance playground of 'Playground' class
      set data with 'test_case_data', replica_processes as parameter
```

```
      wait until liveness time bound is reached
      safe = is_safe(ledger_folder_path, n_twins, n_replicas)
      live = is_live(ledger_folder_path, n_twins, n_replicas)

      log safety and lives  for the test case


      Kill all processes


Function is_safe(directory_path, n_twins, n_replicas) :
    transaction_dictionary = defaultdict(lambda : 0)
    for i in range(0,n_twins) :
        filename = "validator_" + str(i) + ".ledger"
        fp = open file  (directory_path + "/" + filename, 'r')
        for line in file :
            txn = line.strip()
            transaction_dictionary[txn] =
            transaction_dictionary[txn] + 1
        close file
        filename = "validator_" + str(i) + "_twin.ledger"
        fp = open file  (directory_path + "/" + filename, 'r')
        for line in file :
            txn = line.strip()
            transaction_dictionary[txn] =
            transaction_dictionary[txn] +  1
        close file
    for i in range(n_twins, n_replicas) :
        filename = "validator_" + str(i) + ".ledger"
        fp = open file (directory_path + "/" + filename, 'r')
        for line in file :
            txn = line.strip()
            transaction_dictionary[txn] = transaction_dictionary[txn] +  1
        close file
    for transaction in transaction_dictionary :
        if transaction_dictionary[transaction] < 2 * n_twins  + 1 :
            return False
    return True

Function is_live(directory_path, n_twins, n_replicas) :
    validator_dict = {}
    for i in range(0,n_twins) :
        filename = "validator_" + i + ".ledger"
        open file (directory_path + "/" + filename, 'r')
        line_count = 0
        for line in file :
            if '-' in line : line_count = line_count + 1
        close file
        validator_dict[filename] = line_count
        filename = "validator_" + str(i) + "_twin.ledger"
        open file (directory_path + "/" + filename, 'r')
```

```
        line_count = 0
        for line in file :
            if '-' in line : line_count = line_count + 1
        close file
        validator_dict[filename] = line_count
    for i in range(n_twins, n_replicas) :
        filename = "validator_" + str(i) + ".ledger"
        open file (directory_path + "/" + filename, 'r')
        line_count = 0
        for line in file :
            if '-' in line : line_count = line_count + 1
        close file
        validator_dict[filename] = line_count

    validator_cnt = 0
    for validator in validator_dict :
        if validator_dict[validator] == 0  :
            validator_cnt = validator_cnt + 1
    if n_twins  + n_replicas - validator_cnt  > 2 * n_twins + 1 : return True
    return False



class Playground(test_data, replica_processes) :  // test_data is the same as rounds
    self.test_data = test_data

    function get_current_leader(round_no) :
        leader = test_data[round_no - 1][0]
        if leader of the form replica_{x}f, replace leader with replica_x
        return leader
    self.replica_blocks = {}
    function main() :
        loop: wait for next event M ; Main.start event processing(M)
        Procedure start event processing(M)
            if M is a proposal message:
                replica_blocks[M.sender].append[M.block.id]
                process_broadcast_message(M)
            if M is a vote message then process_single_msg(M)
            else : process_broadcast_message(M)

    function get_round_(M):

        round = based on M.msg_type
        return round


    function partition_list(validator, round):
        initialize set to_list
        partitions = test_case[str(round)]["partitions"]
        initialize id  = validator
        initialize value = False
        for partition in partitions:
```

```
        for replica in partition:
            replica = replica[8:]
            if id == replica:
                value = True

        if value:
            for replica in partition:
               to_list.add(replica)
            value = False

    return to_list

function dropping_messages(validator, round, dest, message_type):
    if test_case[str(round)]["failType"] == 0 and
    test_case[str(round)]["messageType"] == message_type :
        dest = dest - faulty_nodes(validator, round)
    return dest

function  faulty_nodes(validator, round):
    initialize set dest
    for replica in list(test_case[str(round)]["src_to_dest"].keys()):
        # replica = list(test_case[str(round)]["src_to_dest"].keys())[0]
         initialize id  = validator
        for dest_rep in test_case[str(round)]["src_to_dest"][replica]:
                dest.add(dest_rep)

    return dest


function process_broadcast_message(M) :
    sender = M.sender
    sender_process_id = replica_ids[sender]
    round = get_round(M)
    dests = partition_list(sender, round)
    dests = dropping_messages(sender, round, dests, 0)
    dests = get process_ids of replicas in dests using replica_processes

    initialise delay_set =  set()
    if test_case[round]["failType"] == 1 and
    test_case[round]["messageType"] == M.messageType :
        delay_set = faulty_nodes(sender, round)

    send(M, to=dests - delay_set )
    wait for delay
    send(M, to=delay_set )

 function process_single_msg() :
    sender = M.sender
    sender_process_id = replica_ids[sender]
    round = get_round(M)
    receiver_id = M.receiver_id
    dests = partition_list(sender, round)
```

```
        dests = dropping_messages(sender, round, dests, 0)
        dests = get process_ids of replicas in dest using replica_processes
                    in partition for round
        receiver = replica_ids[receiver_id]

        initialise delay_set =  set()
        if test_case[round]["failType"] == 1 and
        test_case[round]["messageType"] == M.messageType :
            delay_set = faulty_nodes(sender, round)

        send(M, to=dests - delay_set )
        wait for delay
        send(M, to=delay_set )


class replica :
    process_certificate_qc(qc):
        //LeaderElection_update_leaders(qc)

      // replace get_leader function
    function LeaderElection_get_leader(round):
        return playground.get_current_leader(round)


    function send_msg_replica(to_replica_id, msg_type, m):
        ....
        // 'self' contains the the process_id of the sender
        send((msg_type, self, replica_id, sign_record(
            sign_msg, private_key), m, to_replica_id ), to=playground)

    // replica_id is id of the source replica
    function  broadcast_msg(msg_type, m):
        ....
        // 'self' contains the the process_id of the sender
        send((msg_type, self, replica_id, sign_record(
            sign_msg, private_key), m, all_replicas_id), to=playground)
```