CSE535 Phase 3: Design and Pseudocode for Twins

Dhaval Bagal, Sanket Goutam

December 2, 2021

Contents

Chapter 1

Twins Generator

1.1 High Level Overview

Generator uses the following functions:

1. `partition_in_k_subsets()`
2. `add_twins()`
3. `project_num_test_cases()`
4. `generate_test_cases()`
5. `write_test_cases()`

`partition_in_k_subsets()`

This function is used to enumerate all possible partitions of a list of elements into k subsets.

`add_twins()`

This function is used to add twins to the generated partitions.

`project_num_test_cases()`

Used to calculate the number of test cases that will be generated to make provisions in case this number explodes.

`generate_test_cases()`

This function first enumerates all possible partitions of the list of nodes into k subsets.

Each of the leader candidates derived from the configuration is mapped to every possible partition generated in the previous step. This is termed as scenario generation.

After generating all possible scenarios, the function permutes these scenarios for the configured number of rounds.

In addition, the function also tracks the number of test cases that will be generated with the given configuration before actually generating them. Based on this number, the algorithm decides the type of pruning if required.

`write_test_cases()`

As the name suggests, this function processes the raw test cases and writes it to a json file

`liveness_assured_filtering()`

This function ensures that the generated test cases do not result in failure of liveness testing. If generated test cases result in alternating TC and QC, then liveness of the system is affected. To avoid this, we ensure that in every test case there are at least 2 consecutive rounds with 2f+1 nodes to reach quorum.

1.2 Some Aspects

1.2.1 Operation mode

The system supports **offline** mode only. The generator writes all the generated test cases to a json file which is then read by the executor and executed sequentially.

1.2.2 Selection of leaders

The system has a property in the configuration named **elect_fault_leaders_only** which decides the leader candidates.

1.2.3 Enumeration limits

The parameter **num_test_cases_upper_bound** in the configuration controls the number of items produced by the generator in each of the three steps.

1.2.4 Enumeration order

The system supports both **deterministic** and **randomised** enumeration order.

With **randomised enumeration**, the cross-product is taken over a subset of the scenarios during each round. This subset of scenarios is chosen randomly.

With **deterministic enumeration**, the algorithm continuously reduces the number of rounds and permutes the scenarios over the reduced number of rounds. The scenarios are repeated in a round robin fashion to fill the remaining rounds.

1.2.5 Liveness testing

To ensure liveness, we need to generate test cases in which at least 2 consecutive rounds should have 2f+1 nodes to achieve a quorum.

Chapter 2

Twins Executor

2.1 High Level Overview

Executor uses the following major functions:

```
1.  run()
2.  intercept_rounds()
3.  check_safety_violation()
4.  check_liveness_violation()
```

`run()`

This function simply executes every test case that is passed to it. Based on the partitions in the test case it instructs the playground to generate appropriate network topology. It also instructs the playground to set the leader for the BFT protocol based on the rounds as mentioned in the test case.

The protocol starts executing on the playground and after executing the protocol for configured number of rounds, it checks for safety and liveness violations

`intercept_rounds()`

This function anticipates the round and updates the network partition accordingly.

We configure the number of nodes defining a round in the configuration. Whenever the executor intercepts these many messages, it updates the partition to the one provided in the test case for that particular round.

`check_safety_violation()`

This function checks if 2f+1 nodes in the system have the same ledger hash. It implies that the correct nodes have committed the blocks successfully and correctly.

`check_liveness_violation()`

We put a liveness bound specifying that every block should get committed within 3 rounds. This function checks if the liveness bound is obeyed by the system

2.2 Some Aspects

2.2.1 Partition change trigger

The executor starts a thread which intercepts every message and records its round number. For a particular round, if enough number of messages (as configured) are received from different validators, the system is assumed to be in that particular round and the partition for that round as mentioned in the test case is realised by the playground.

2.2.2 Message drops

There are two types of message drops viz inter-partition and intra-partition.

The inter-partition drops are handled automatically based on the partitions mentioned in the test cases.

Intra-partition message drops are taken care of by **round_based_msg_filtering_policies** which is given as input to the playground. The playground looks at this message delivery schedule and drops messages accordingly.

2.2.3 Property checking

The system supports safety and liveness property checks once it is executed for the configured number of rounds, by looking at the persistent ledger for each validator.

2.2.4 Process identifiers

As mentioned in the paper, the network playground is designed to be run on a single host where the validators are threads rather than individual processes. Thus, we follow the design principle of using threads for validators instead of processes, in accordance with the paper.

Chapter 3

Pseudocodes

3.1 Generator

DEFINE CLASS TwinsGenerator:

```
    DEFINE FUNCTION partition_in_k_subsets(self, set, k)::
        """
        @references:
        - https://github.com/asonnino/twins-generator/blob/master/generator.py
        """
        DEFINE FUNCTION stirling2(n, k):
                assert n > 0 and k > 0
                IF k EQUALS 1:
                    RETURN [
                        [[x FOR x IN range(n)]]
                    ]
                ELSEIF k EQUALS n:
                    RETURN [
                        [[x] FOR x IN range(n)]
                    ]
                ELSE:
                    SET s_n1_k1 TO stirling2(n-1, k-1)
                    SET tmp TO stirling2(n-1, k)
                    FOR i IN range(len(s_n1_k1)):
                        s_n1_k1[i].append([n-1])
                    SET k_s_n1_k TO []
                    FOR _ IN range(k):
                        k_s_n1_k += deepcopy(tmp)
                    FOR i IN range(len(tmp)*k):
```

```
                    k_s_n1_k[i][i // len(tmp)] += [n-1]
                SET partitions TO s_n1_k1 + k_s_n1_k
                RETURN partitions
        SET partitions TO stirling2(len(set), k)
        SET partitions TO [
                [
                    [set[idx] FOR idx IN subset]
                    FOR subset IN partition
                ]
                FOR partition IN partitions
            ]
        RETURN partitions
```

DEFINE FUNCTION **add_twins(self, partitions):**

```
    """
    - generates twin nodes with 'twin_' prefix
    - adds the twins to every subset in every partition based on the
    'allow_twins_in_same_partition' policy
    """
    twins = dict()
    FOR node IN compromised_nodes:
        twin_i = (i+1)%len(nodes)
        twins[node] = nodes[twin_i]
    IF allow_twins_in_same_partition IS NOT allowed THEN:
        drop all network partitions containing both twins in the same partition
    ELSE:
        filtered_partitions = partitions
    RETURN twins, filtered_partitions
```

DEFINE FUNCTION **project_num_test_cases(self, num_parts, num_leaders, num_rounds):**

```
    """
    - calculate the number of test cases that will be generated
    with the given input
    """
    SET num_leader_partition_comb TO num_leaders * num_parts
    SET num_round_scenario_mappings TO num_leader_partition_comb**num_rounds
    RETURN num_round_scenario_mappings
```

DEFINE FUNCTION **liveness_assured_filtering(self, test_cases, num_byzantine):**

```
    """
    @function:
```

```
- IF there is a sequence of alternating qc and tc, then liveness is blocked
- to avoid this, there should be atleast 2 consecutive rounds with a 2f+1 quorum to
    ensure 2 consecutive qcs are formed and the block is committed
"""

DEFINE FUNCTION liveness_assured(test_case, quorum_size):
    SET partitions TO [case[1] FOR case IN test_case]
    SET prev_size TO len(max(partitions[0], key=len))
    SET consecutive_quorums_found TO False
    FOR i IN range(1, len(partitions)):
        SET partition TO partitions[i]
        SET max_subset_size TO len(max(partition, key=len))
        IF max_subset_size EQUALS prev_size==quorum_size:
            SET consecutive_quorums_found TO True
            break
        SET prev_size TO max_subset_size
    RETURN consecutive_quorums_found
```

DEFINE FUNCTION generate_test_cases(self):

```
# determine the minimum size of the partition set to ensure liveliness
# a partition with all sets containing less than 2f+1 nodes will block
liveliness

SET num_byzantine TO (len(self.nodes)-1)//3
SET num_partition_sets TO self.config.num_partition_sets
SET elect_faulty_leaders_only TO self.config.elect_faulty_leaders_only
SET num_partition_sets TO self.config.num_partition_sets
SET num_test_cases_upper_bound TO self.config.num_test_cases_upper_bound
SET enumeration_order TO self.config.enumeration_order
SET num_rounds TO self.config.num_rounds

SET partitions TO self.partition_in_k_subsets(
                self.nodes,
                k=num_partition_sets
            )
SET twins, partitions TO self.add_twins(partitions)

IF elect_faulty_leaders_only EQUALS True:
    SET leader_candidates TO self.compromised_nodes
ELSE:
    SET leader_candidates TO self.nodes

SET scenarios TO list(itertools.product(leader_candidates, partitions))
SET num_test_cases_projected TO self.project_num_test_cases(
                                SET num_parts TO len(partitions),
                                SET num_leaders TO len(leader_candidates),
                                SET num_rounds TO num_rounds
                            )
DEFINE FUNCTION repeat_scenarios(test_case, target_num_rounds):
    SET extended_test_case TO [None,]*target_num_rounds
    SET j TO 0
    FOR i IN range(target_num_rounds):
        SET extended_test_case[i] TO test_case[j]
        SET j TO (j+1)%len(test_case)
    RETURN extended_test_case

test_cases=[]

IF num_test_cases_projected > num_test_cases_upper_bound:
    IF enumeration_order EQUALS "deterministic":
        # reduce the number of rounds,
        # take cross-product of the scenarios with themselves
        FOR the reduced number of rounds
        SET r TO num_rounds-1
        WHILE True:
            SET num_test_cases_projected TO self.project_num_test_cases(
                                SET num_parts TO len(partitions),
                                SET num_leaders TO len(leader_candidates),
                                SET num_rounds TO r
                            )
            IF num_test_cases_projected > num_test_cases_upper_bound:
                r-=1
            ELSE:
                break
        SET test_cases_for_reduced_rounds TO list(itertools.product
                                                    (scenarios, repeat=r))
        # repeat the scenarios IN a round robin fashion to
        fill up the test-case FOR 'num_rounds' rounds
        FOR test_case IN test_cases_for_reduced_rounds:
            test_cases += [repeat_scenarios(test_case, num_rounds)]
    ELSEIF enumeration_order EQUALS "randomized":
        SET num_scenarios_to_consider_per_round TO []
        SET num_test_cases TO 1
        SET num_scenarios TO len(scenarios)

        # taking cross-product of the scenarios with themselves
        FOR 'num_rounds' times gives us the test_cases
        FOR i IN range(num_rounds):
            SET running_num TO num_test_cases*num_scenarios
```

```
                IF running_num>=num_test_cases_upper_bound:
                    WHILE running_num>num_test_cases_upper_bound:
                        num_scenarios -= 1
                        SET running_num TO num_test_cases*num_scenarios
                SET num_test_cases TO running_num
                num_scenarios_to_consider_per_round += [num_scenarios]
            SET scenario_list TO []
            FOR n IN num_scenarios_to_consider_per_round:
                random.shuffle(scenarios)
                scenario_list += [scenarios[0:n]]
            SET test_cases TO itertools.product(*scenario_list)
    ELSE:
        SET test_cases TO list(itertools.product(scenarios, repeat=num_rounds))
```

DEFINE FUNCTION intra_partition_loss_schedule(self, num_rounds):

```
    DEFINE FUNCTION random_subset(_set):
        SET out TO set()
        FOR elem IN _set:
            IF random.randint(0, 1) EQUALS 0:
                out.add(elem)

        IF len(out)==0:
            SET i TO random.randint(0, 1)
            out.add(list(_set)[i])
        RETURN out
    RETURN [
        list(random_subset(self.intra_partition_msg_loss_candidates))
        FOR _ IN range(num_rounds)
    ]
```

3.2 Executor

DEFINE CLASS TwinsExecutor:

DEFINE FUNCTION track_rounds(self, num_rounds, test_case_timer):

```
    """
    - keep track of the global notion of the round
    - stop the thread after num_rounds number of global rounds are done
    - a global round is assumed to be entered by all processes, once leader enters the round
    """
    SET highest_round TO self.global_round
    SET replicas TO r:i FOR i,r IN enumerate(self.system_config.replica_id_set)
    WHILE True:
        SET i TO replicas[self.leaders[self.global_round]]
        SET r TO self.replica_rounds[i].value
        IF r>highest_round:
            SET self.global_round TO r
            SET highest_round TO r

        FOR i IN range(self.system_config.num_validators):
            SET self.replica_rounds[i].value TO self.global_round

        IF self.global_round >= (num_rounds-1) or
            len(self.requests) EQUALS self.num_cmds or self.time_up:
            test_case_timer.cancel()
            break
```

DEFINE FUNCTION execute_testcase(self, testcase, num):

```
    """
    - emulate BFT protocol according to the testcase and check
    for safety and liveness violations
    """
    SET self.time_up TO False
    DEFINE FUNCTION timeout_testcase():
        SET self.time_up TO True
    SET delta TO self.system_config.transmission_time_delta
    SET self.num_rounds TO testcase["num-rounds"]
    # start a timer FOR the testcase after which it should force exit
    SET test_case_timer TO threading.Timer(
        SET interval TO int(delta*4*self.num_rounds),
        function=timeout_testcase
    )
    test_case_timer.start()
    SET self.global_round TO 0

    # maintain a shared-variable between executor
    and every process which maintains the local round the process currently is IN
    SET self.replica_rounds TO [multiprocessing.Value('i', 0)
            FOR _ IN range(self.system_config.num_validators)]
    SET faulty_nodes  TO set(testcase["compromised-nodes"])
    SET twin TO testcase["twins"]
    twin.update(v:k FOR k,v IN testcase["twins"].items())
    SET replica_id_set TO self.system_config.replica_id_set
    SET client_id_set TO self.client_config.client_id_set
```

```
# get the list of leaders from the testcase to
update the leader at every validator FOR every global round
SET self.leaders TO []
FOR i IN range(self.num_rounds):
    SET scenario TO testcase[str(i)]
    self.leaders += [scenario["leader"]]
SET replicas TO []
SET manager TO multiprocessing.Manager()

# this queue is used to send a stop signal to all
replicas once all client responses have been received
SET self.replica_notify_queue TO manager.Queue()

# whenever client gets f+1 responses for a request, it notifies the executor
# when all requests for all clients are responded to,
executor shuts down the entire system

SET self.client_notify_queue TO manager.Queue()
FOR i IN range(self.system_config.num_validators):
    SET rid TO replica_id_set[i]
    # give faulty node the same private key as its twin, so that it can equivocate
    IF rid IN faulty_nodes:
        SET private_key TO self.system_config.replica_keys[twin[rid]][0]
    ELSE:
        SET private_key TO self.system_config.replica_keys[rid][0]
    SET keys TO {
        "private-key":private_key,
        "public-keys":{
            {
                r:k[1] FOR r,k IN self.system_config.replica_keys.items()
            },
            {
                r:k[1] FOR r,k IN self.client_config.client_keys.items()
            }
        }
    }
    # twins playground at each replica intercepts all msgs
    and drops/processes them according to the policies
    SET playground TO twins_playground.TwinsPlayground(
        node_id=rid,
        testcase=testcase,
        system_config=self.system_config,
        client_config= self.client_config,
        SET round_var TO self.replica_rounds[i],
        SET notify_queue TO self.replica_notify_queue
    )
    SET r TO multiprocessing.Process(target=self.spawn_replicas,
                    args=(rid, keys, playground))
    replicas += [r]
    r.start()

# maintain client request-response metadata as
and when clients receive responses to their requests
SET self.requests TO []
SET self.num_cmds TO self.client_config.num_clients *
                    self.client_config.num_commands_to_be_sent

SET track_rounds_thread TO threading.Thread(target=self.track_rounds,
                args=((testcase["num-rounds"], test_case_timer)) )
track_rounds_thread.start()
time.sleep(2)
SET clients TO []
FOR i IN range(self.client_config.num_clients):
    SET cid TO client_id_set[i]
    SET keys TO
        "private-key":self.client_config.client_keys[cid][0],
        "public-keys":
            r:k[1] FOR r,k IN self.system_config.replica_keys.items()



    # client playground intercepts all msgs to and from the client
    and notifies executor about the responses that the
    client has received for the requests sent

    SET playground TO client_playground.ClientPlayground(
        node_id=cid,
        client_config= self.client_config,
        notify_queue=self.client_notify_queue
    )
    SET c TO multiprocessing.Process(target=self.spawn_clients,
                    args=(cid, keys, playground))
    clients += [c]
    c.start()
# stop the BFT system IF either all requests are responsded to or
# IF number of rounds exceed the configured number of rounds IN the testcase
WHILE True:
    self.requests += [self.client_notify_queue.get(timeout=1)]

    IF len(self.requests) EQUALS self.num_cmds or self.time_up or
        self.global_round >= (testcase["num-rounds"]-1):
        test_case_timer.cancel()
```

```
            # send a stop signal to all replicas
             FOR i IN range(self.system_config.num_validators):
                 self.replica_notify_queue.put(
                     "type":"stop-signal"
                 )
             break
     FOR node IN replicas+clients:
         node.terminate()
     self.check_liveness()
     self.check_safety()
```

DEFINE FUNCTION check_safety(self):

```
    """
    - checks the number of consistent ledgers by comparing their hashes
    """
    IF len(ledger_files) > 0:
        SET ledger_hashes TO defaultdict(int)
        SET ledgers TO dict()
        FOR fname IN ledger_files:
            SET path TO os.path.join(ledger_folder, fname)
            with open(path, "r") as fp:
                SET ledger TO fp.read()
            SET ledger_hash TO crypto.Crypto.hash(ledger)
            ledger_hashes[ledger_hash] += 1
            SET ledgers[ledger_hash] TO ledger

        # safety property is satisfied only IF there are 2f+1 consistent ledgers
        SET is_safe TO False
        SET num_consistent_ledgers TO max(ledger_hashes.values())
        SET consistent_ledger_idx TO max(ledger_hashes, key=ledger_hashes.get)
        IF num_consistent_ledgers >= 2*self.system_config.num_byzantine+1:
            SET ledger TO ledgers[consistent_ledger_idx]
            SET ledger TO [l FOR l IN ledger.split("") IF len(l)>0]
            IF len(ledger)==len(set(ledger)):
                SET is_safe TO True
```

DEFINE FUNCTION check_liveness(self):

```
    """
    - FOR each response received at the client and
    forwarded to the executor (via notify_queue), find the response time
    - IF the response time FOR a request is below 7*delta,
    then liveness is ensured, else not
    - not all requests will have liveness ensured
```

```
- this is because we are running the system only for a configured number of rounds,
    so some requests which arrive late may not get committed before
    the configured number of rounds
- while determining liveness, the timestamp of the original request sent is
considered and not that of the retransmitted request
"""

IF len(self.requests)==0:
    SET num_requests TO self.client_config.num_commands_to_be_sent *
                        self.client_config.num_clients
    RETURN
SET liveness_time_bound TO 7*self.system_config.transmission_time_delta
FOR req IN self.requests:
    SET sent_ts TO datetime.datetime.strptime(req["ts"],
                    "%m-%d-%Y::%H:%M:%S.%f")
    SET rcvd_ts TO datetime.datetime.strptime(req["response-ts"],
                    "%m-%d-%Y::%H:%M:%S.%f")
    SET t_delta TO (rcvd_ts - sent_ts).total_seconds()
    SET req["response-time"] TO str(t_delta) + " secs"
    SET req["liveness-maintained"] TO (t_delta <= liveness_time_bound)
```