The Posquatters
Andrew Burford
Ethan DeTurk
Jiawei Qian

Phase 3

Text Generator Design

The test generator makes extensive use of Python's "generator" pattern because it is very convenient for enumeration. For deterministic test generation, the algorithm can efficiently write a large number of tests to the test file without having to allocate too much memory. For randomized tests, the configurations are collected into a list in memory and random.choice() is used to pick one.

For intra-partition message drops, we chose to include explicit events in the test cases rather than using different network partitions as suggested in the phase 3 document. We took this path because it makes it easier to drop particular message types and it also simplifies the partition triggers.

For liveness testing, we guarantee that there is a super-majority quorum in later rounds. We pad each test case with seven rounds of full network partitions with no message drops to ensure that Property 4 of DiemBFT can be detected.

Inputs:
Section 5.2 "hooks." P and L are explicitly mentioned in the paper, C is the number of leader-partition configuration assignments to rounds, and E controls the number of intra-partition message drops
R (int) - the number of rounds
P (int) - the limit of the total number of generated partitions
K (int) - the number of subsets within each partition
C (int) - round configurations
L (int) - leader choices
E (int) - intra partition drops

N (int) - number of nodes, excluding twin duplicates
F (int) - number of faulty nodes that have twins

These are mentioned in section 5.2 and are pretty self explanatory.
random_configurations controls the determinism of per-round configuration selection.
random_partitions (boolean)
random_leaders (boolean)

random_configurations (boolean)

allow_nonfaulty_leaders (boolean) - Controls whether non-faulty leaders can be selected
allow_quorumless_partitions (boolean) - Controls whether partitions with no quorum can be generated

Outputs:
The output file is in JSON format. The first line is a list of replica identifiers with no duplicates. The second line is a list of twin identifiers. The third line is a list of bugs. All the remaining lines are test cases, which are represented as a list of rounds. A round is represented as a list of three elements: the identifier of the leader, the partition, and the list of exceptions. A partition is represented by a list of “buckets,” each of which is a list of identifiers of the replicas in the bucket. An exception is represented by a list containing the identifier of the sender, the identifier of the recipient, and the message type.

Text Generator Pseudocode

Since the “generator” pattern is pretty specific to Python and central to the test generator, our pseudocode is just Python.

```
import os
import importlib
import random
import json
import sys
from src.testconfig import *

tests = None
R = None # rounds
P = None # partitions
K = None # buckets
C = None # round configurations
L = None # leader choices
E = None # inter partition drops

N = None # nodes
F = None # faulty nodes

random_partitions = False # random partitions
random_leaders = False # random leaders
random_configurations = False # random per-round configurations

allow_non_faulty_leaders = True
allow_quorumless_partitions = False

all_partitions = None
all_configurations = None

originals = None
twins = None
replicas = None
eligible_leaders = None

# Take the first n things from the generator
def take(generator, n):
    i = 0
    for (i, thing) in enumerate(generator):
        if i >= n:
            break
        yield thing

def twin(replica):
    if '\'' in replica:
        return replica[:-1]
    return replica + '\''

# Enumerates all possible intra-partition message drops
def excepts(endpoints):
    msg_types = [
        MsgType.Proposal,
        MsgType.TimeOut,
        MsgType.Vote,
        MsgType.Wildcard
    ]
```

```
    for src in endpoints:
        for dst in endpoints:
            for msg_type in msg_types:
                yield Except(src, dst, msg_type)

# Generates all samples of k things from the generator
def replacement_samples(gen, gen_args, k):
    if k == 0:
        yield []
    else:
        for sample in replacement_samples(gen, gen_args, k - 1):
            for thing in gen(*gen_args):
                yield [thing] + sample

# Generate all combinations of message drops
def except_samples(bucket):
    i = 0
    while True:
        for sample in replacement_samples(excepts, [bucket], i):
            yield sample
        i += 1

# Pick a random partition
def random_partition_gen():
    global all_partitions
    if all_partitions is None:
        all_partitions = list(partition_gen_quorum())

    while True:
        yield random.choice(all_partitions)

# Generate partitions deterministically
def partition_gen(n, k):
    if n == 0:
        yield []
    if k == 0 or n < k:
        return
        # Recursively generate partitions for the n - 1 other replicas
    for partition in partition_gen(n - 1, k - 1):
        partition.append([replicas[n - 1]])
        yield partition
    for partition in partition_gen(n - 1, k):
        for i, bucket in enumerate(partition):
            new_partition = partition.copy()
            new_partition[i] = bucket.copy()
            new_partition[i].append(replicas[n - 1])
            yield new_partition

def partition_gen_quorum():
    for partition in partition_gen(len(replicas), K):
        if max([len(set([replica[0] for replica in bucket])) for bucket in partition]) >= 2 * F + 1 \
                or allow_quorumless_partitions:
            yield partition

# Randomly or deterministically generates partitions
def partitions():
    if random_partitions:
        return random_partition_gen()
    return partition_gen_quorum()

# Randomly or deterministically generates leaders for a bucket. "Bucket" is explained in the
```

```
# next comment
def leaders():
    if random_leaders:
        while True:
            yield random.choice(eligible_leaders)
    else:
        for leader in eligible_leaders:
            yield leader

# Generates leaders and exceptions for a set within a partition. "Set within a partition"
# is referred to as a "bucket" in the pseudocode. The authors of the Twins paper overload
# the word "partition" to refer to both the partition and the elements within it
def partition_except_sets(partition):
    if len(partition) == 0:
        yield []
    else:
        bucket = list(partition[0])
        for partition_except_set in partition_except_sets(partition[1:]):
            for except_sample in take(except_samples(bucket), E):
                yield except_sample + partition_except_set

# Deterministically generate the round configurations. Partitions and leaders might still
# be random if random_partitions or random_leaders is true
def round_gen():
    i = 0
    for leader in take(leaders(), L):
        for partition in take(partitions(), P):
            for partition_except_set in partition_except_sets(partition):
                i += 1
                if i >= C:
                    return
                yield Round(leader, partition, partition_except_set)

# Generates a round configuration at random. If the partitions are deterministic, it
# samples from the first P partitions. If the leaders are deterministic, it samples from
# the first L leaders
def random_round_gen():
    global all_configurations
    if all_configurations is None:
        all_configurations = list(round_gen())
    while True:
        yield random.choice(all_configurations)

# Full-partition round simulating GST (no message drops)
def GST_round():
    return Round(replicas[0], [originals + twins], [])

def test_generator():
    if random_configurations:
        generator = random_round_gen
    else:
        generator = round_gen
    for test in take(replacement_samples(generator, (), R), tests):
        # Pad with seven full-parition rounds for property 4
        yield test + [GST_round()] * 7

def deserialize(test_case):
    rounds = []
    for rnd in json.loads(test_case):
        leader = rnd[0]
        partition = rnd[1]
```

```
        exp = rnd[2]
        for j, ex in enumerate(exp):
            src = ex[0]
            dst = ex[1]
            msg_type = MsgType(ex[2])
            exp[j] = Except(src, dst, msg_type)
        rounds.append(Round(leader, partition, exp))
    return rounds

#pylint: disable=invalid-name,global-statement
def main():
    global tests
    global R
    global P
    global K
    global C
    global L
    global E
    global N
    global F
    global random_partitions
    global random_leaders
    global random_configurations
    global allow_non_faulty_leaders
    global allow_quorumless_partitions
    global all_partitions
    global all_configurations
    global originals
    global twins
    global replicas
    global eligible_leaders
    global originals
    global twins
    global replicas
    global eligible_leaders

    if len(sys.argv) != 2:
        print('Usage: testgenerator.py CONFIG_FILE')
        sys.exit(1)
    fname = sys.argv[1]
    config_name = os.path.basename(fname).split('.')[0]
    if fname[-3:] == '.py' or fname[-3:] == '.da':
        modname = os.path.dirname(fname).replace('/', '.') + '.' + config_name
        mod = importlib.import_module(modname)
        (tests, R, P, K, C, L, E, N, F, random_partitions, random_leaders, random_configurations,
            allow_non_faulty_leaders, allow_quorumless_partitions, out_file) = mod.test_case
        all_partitions = None
        all_configurations = None
        originals = [chr(ord('a')+j) for j in range(N)]
        twins = [twin(replica) for replica in take(originals, F)]
        replicas = originals + twins
        if not os.path.isdir("generated_tests"):
            os.mkdir("generated_tests")
        if allow_non_faulty_leaders:
            eligible_leaders = originals
        else:
            eligible_leaders = list(take(originals, F))
        with open("generated_tests/" + out_file, 'w') as out_file:
            json.dump(originals, out_file)
            out_file.write('\n')
            json.dump(twins, out_file)
```

```
            out_file.write('\n')
            # no bugs
            json.dump([], out_file)
            out_file.write('\n')
            for test in test_generator():
                test_str = json.dumps(test)
                # test_str = deserialize(test_str)
                # test_str = f'{str(test_str)}'
                out_file.write(test_str+"\n")
    else:
        print('we only support .py and .da configuration files')

if __name__  == '__main__':
    main()
```

Text Executor Design

We implement the test executor by creating a TestCaseExec process which acts as a central message hub through which all messages are sent. There is also a MainExec process which reads in test case configurations and runs a TestCaseExec process for each test case configuration. These two classes replace the Runner class implemented in Phase 2 since it also reads in configurations for runs, initializes Replica processes, and calls the setup and start methods on those processes. We then create a ReplicaTestExec class which, similar to ReplicaFI, is a subclass of Replica which overrides the send method and also the receive method. Replicas are given a list of replica identification strings (ex: ['a', 'b', 'c', 'd']) with no information about twin processes. We modify the Replica code so that whenever the send method is called, the to= parameter is an identifier string for another replica. ReplicaTestExec redirects this message to the TextCaseExec process. TestCaseExec then forwards the message to the corresponding replica or replicas if the destination replica is a twin. The receive method is also overridden in order to change the sender back to an identifier string rather than the TestCaseExec process. This way, Replicas deal only with replica identifier strings and not process objects. Another design choice could have been to perform message dropping in the ReplicaTestExec class similar to the ReplicaFI implementation. We chose not to do it this way because the safety and liveness properties require gathering information from all of the replicas so some type of centralized process is required anyway. Including the message dropping in a TestCaseExec class allows most of the test executor to be implemented in one place.

The test generator saves test cases to a file which are read in and converted to TestConfig named tuples. This tuple contains some of the same information from the test case configuration objects used in Phase 2: the transmission delay bound, RNG seed, test case name, and total timeout time. In addition, TestConfig specifies a list of replicas and which of those are twin replicas. The test executor uses this information to create two dictionaries: one that maps a process object to its identifier string (for twins this could be either a or a') and one that maps an identifier string to a list of processes (list of length 1 for honest replicas and length 2 for twins). Note that the keys in the second dictionary do not differentiate between honest and twin replicas i.e. a is a key but a' is not. Finally, the TestConfig tuple specifies a list of rounds, where each round has a leader, partition, and exception rules to allow for intrapartition message drops. This information allows the TestCaseExec process to multiplex messages to and from twin replicas. Round partitions read in from TestConfig are converted to lists of processes in each partition bucket. When a message is received, we forward it to every process that matches the recipient string identifier and is in the same partition as the sender for that round.

In order to determine round numbers, the TestCaseExec performs some rudimentary deep packet inspection. Upon receiving a message to forward, it checks

the type of message that is being sent and then retrieves the round number from this message. TestCaseExec uses this round number to check whether the sender and receiver are in the same partition specified in the TestConfig. If they are not then it checks the exception list and if there is no exception then it forwards the message. Otherwise, it drops the message. Round numbers are determined on a message by message basis so no centralized partition changes occur. The partitions for every round are kept in memory throughout the duration of execution.

Timeout messages are treated specially because intrapartition exceptions are ignored after two timeout messages have already been sent by a replica in the same round. This allows progress as long as there is a bucket with a quorum in each partition. Sync request and sync response messages were modified to include a round number which is set to the round number of the message that triggered the sync request. This allows sync messages to be filtered based on the round that the replica is syncing up to.

In order to force specific leaders to be elected in each round, we create a new module for leader election which completely replaces the current LeaderElection module. TestCaseExec initializes this module with the list of leaders from the TestConfig. We modify the Replica class so that it imports this module instead of the diem leader election module. This module has the same method signatures and return types as LeaderElection, but uses the leaders provided in the TestConfig. This is the approach used in the Twins paper and requires the least modification to the Replica code since you only need to change the name of the module being imported.

In order for twin replicas to have equivocating proposals, they must generate different proposals in the same round. We modify the Mempool module so that instead of waiting to receive transactions from a set of clients and adding these to the list of pending transactions, we initialize the pending transactions list during setup to the same list in every replica except in a randomly shuffled order. This is similar to the implementation mentioned in Twins, where every process has a different input to generate transactions blocks. In our implementation, each transaction is a randomly generated string and the number of transactions is based on the maximum number of transactions per block and the number of rounds specified in the test configuration. Since transactions are iteratively chosen from the pending list to create blocks, it is highly unlikely that twins pick the same transactions from their randomly shuffled lists. We chose this approach because this is the way the Twins paper chose to implement this. This more strongly guarantees that proposals will equivocate than if we continued using clients since we cannot control the order of the delivery of client requests.

Since we no longer need client processes, the Replica class is modified so that instead of waiting for clients to say that all of their transactions were committed, TestCaseExec wait for replicas to reach the number of rounds specified in the TestConfig and then tells all of the replicas to terminate. This is checked using the same method to drop packets. Once a replica completes the final round, we stop delivering its

messages. Once we have received at least one message in the final round from all replicas, we terminate. For the special case where there exists a quorumless partition, we only wait for all replicas to reach the round with the quorumless partition. We then wait transmission delay bound * 4 * 7 to ensure no further progress is made and then terminate. Termination is achieved by sending a done message to all the replicas just as was done in Phase 2.

Safety and liveness are checked after each execution using the history of sent and received messages in the TestCaseExec process. Two liveness properties from the paper require checking when blocks are locally committed by honest replicas. Since this cannot easily be inferred by inspecting messages, we modify the code in the Replica class to send a message to TestCaseExec when a block is locally committed. This message is essentially ignored by the TestCaseExec class, we only use it at the end by querying the received message history. Once execution has finished, we check all of the safety and liveness properties using the message history of received messages in the TestCaseExec class.

Property 1. If a block is certified in a round, no other block can gather f + 1 non-Byzantine votes in the same round. Hence, at most one block is certified in each round.

We use distalgo message history queries, aggregators, and quantifiers to check this property. We first iterate through rounds and check that each round satisfies the property. To check if a round to satisfies the property, we check that for each certified block from this round, there does not exist another block that gathered f + 1 votes from non-twin replicas. We get certified blocks by iterating through all blocks which received 2f + 1 votes.

Property 2. For every two globally direct-committed blocks B, B0 , either B ←−∗ B0 or B0 ←−∗ B.

We check this property by aggregating all globally direct committed blocks. We also aggregate all block proposed blocks along with their parent block. This should form a tree of blocks. We then iterate through every combination of globally direct committed blocks and ensure that either the first block extends the second xor the second block extends the first. We determine whether a block is globally direct committed by iterating through all proposed blocks and checking that there were f + 1 votes for the QC of this block in the round after it was proposed.

Property 3. Let r be a round after GST. Every honest validator eventually locally commits some block B by calling Ledger.commit(B.id) for B.round > r.

This property is only semi-decidable since we run the algorithm for a finite number of rounds after GST, and this property only says that eventually a block will be committed after GST with no bounds on when. Nevertheless, we query the history of committed messages sent by the Replica class when a replica locally commits a block. The committed message includes the id of the block, the round that it was proposed in, and the time when it was committed. We use a distalgo quantifier to ensure that each honest replica has committed some block in a round after GST.

Property 4. Suppose all honest validators agree on honest leaders of rounds r, r + 1, r + 2 and r occurs after GST. Then, every honest validator locally direct-commits the honest block proposed in round r within 7Δ time of when the first honest validator entered round r

We collect some data online while the algorithm is executing in order to check this property. TestExec maintains a dictionary mapping round numbers to datetime objects representing the time when an honest replica first entered that round. This is computed by checking the round numbers of all received messages and saving the time whenever a message for a new round is received. Once the algorithm terminates, we iterate through the rounds with three consecutive honest leaders and then iterate through the honest replicas for each round. We query the message history of committed messages to ensure that each honest replica locally committed the block for that round within 7Δ time of when the first replica entered the round.

We chose to check the safety properties offline rather than online because checking them online adds a slight overhead during execution which can interfere with timeouts so it seemed less intrusive to perform this analysis after execution has finished. Liveness properties cannot be checked until certain replicas have reached certain round numbers and keeping track of this during execution seemed unnecessary since the properties can be checked just as easily offline once all nodes have completed all rounds.

TestExec Pseudocode

```
from replica import Replica
import mempool
from replicatestexec import ReplicaTestExec
from testconfig import *
```

```
from client import Client
import leaderelection
import random
from uuid import uuid4
from nacl.signing import SigningKey
import sys
from config import *
from logger import createLogger
import os
import importlib
from datetime import datetime
from testgenerator import deserialize
import json
import testgenerator

class TestCaseExec(process):
    def setup(exec_config_name, main_exec, test_config):
        # Stoller recommends doing this for macOS where UDP max datagram size is too small so this forces TCP
        # TODO log all of these configurations
        formatter ='TestCaseExec'
        self.test_case = test_config.name
        self.logger = createLogger(__name__, 'log/' + exec_config_name + '/%s_executor.log' % test_case,
formatter)
        self.num_replicas = len(test_config.replicas)
        self.num_faulty = len(test_config.twins)
        self.f = num_faulty
        self.twins = [ident[:-1] for ident in test_config.twins]
        self.seed = test_config.seed
        self.transmission_delay_bound = test_config.transmission_delay_bound
        self.nrounds = len(test_config.rounds)
        self.leaders = {i: r.leader for i, r in enumerate(test_config.rounds)}
        self.timeouts = {}
        self.gst = test_config.gst
        logger.info('Test case %s has %d replicas (%d faulty) and %d rounds' % (self.test_case, num_replicas,
f, nrounds))
        # repeat leader of last round because vote messages
        # in the last round need to be send to the leader
        # of the next round
        leaders[nrounds] = leaders[nrounds - 1]
        self.replicas = list(new(ReplicaTestExec, num=(num_replicas + num_faulty)))
        # used to verify property 4
        self.first_to_enter = {}
        random.seed(seed)
        private_keys = {
            r: SigningKey(random.getrandbits(256).to_bytes(32, 'big'))
            for r in test_config.replicas
        }
        public_keys = {
            r: private_keys[r].verify_key
            for r in test_config.replicas
        }
        self.process_to_id = {}
        self.id_to_process = {}
        for r, ident in zip(replicas, (test_config.replicas + test_config.twins)):
            process_to_id[r] = ident
            ident = ident[0]
            id_to_process.setdefault(ident, []).append(r)
        self.partitions = {}
        self.stopper_round = self.nrounds
        for rnd_num, rnd in enumerate(test_config.rounds):
            parts = {}
```

```
            stopper = True
            for i, p in enumerate(rnd.partition):
                if len(set([r[0] for r in p])) >= 2 * f + 1:
                    stopper = False
                for r in p:
                    # convert d' to d twin process
                    processes = id_to_process[r[0]]
                    proc = processes[len(r)-1]
                    parts[proc] = i
            self.partitions[rnd_num] = parts
            if stopper:
                self.stopper_round = min(self.stopper_round, rnd_num)
        if nrounds == stopper_round:
            logger.info('No quorumless rounds in this test case')
        else:
            logger.info('Round %d will have no quorum' % stopper_round)
        for replica in replicas:
            logger.debug(f'[setting up replicatestexec {replica}]')
            ident = self.process_to_id[replica][0]
            setup(
                replica, (
                    self,
                    test_config.replicas,
                    transmission_delay_bound,
                    public_keys,
                    private_keys[ident],
                    ident,
                    seed,
                    random.random(),
                    num_faulty,
                    leaders,
                    len(test_config.rounds),
                    test_config.name,
                    exec_config_name,
                    test_config.bugs
                )
            )
        self.t = test_config.timeout
    # msg = (src/dst, (msg_type, content))
    # return block ids
    def blocks():
        return setof(msg[1][1].block.id, received(msg), msg[1][0] == 'proposal' and get_round(msg[1]) <
nrounds)

    # return (block_id, round, parent_id)
    def blocklinks():
        return setof((msg[1][1].block.id, msg[1][1].block.round, msg[1][1].block.qc and
msg[1][1].block.qc.vote_info.id), received(msg), msg[1][0] == 'proposal' and get_round(msg[1]) < nrounds)

    # return replica ids of all votes for specific block in a round
    # twins have same id
    def voters(block, rnd):
        return setof(msg[1][1].sender, received(msg), msg[1][0] == 'vote' and msg[1][1].vote_info.id == block
and msg[1][1].vote_info.round == rnd)

    def cert_blocks(rnd):
        qsize = 2*f if 'small_quorum' in test_config.bugs else 2*f + 1
        return setof(block, block in blocks(), lenof(v, v in voters(block, rnd)) >= qsize)

    def weak_cert_blocks(rnd):
        return setof(block, block in blocks(), lenof(v, v in voters(block, rnd), v not in twins) >= f + 1)
```

```
    def global_commit_voters(block, rnd):
        return setof(msg[1][1].sender, received(msg), msg[1][0] == 'vote' and msg[1][1].vote_info.parent_id ==
block and msg[1][1].vote_info.round == rnd)

    def is_global_commit(block, rnd):
        return lenof(v, v in global_commit_voters(block, rnd + 1), v not in twins) >= f + 1
    # (block, parent) where parent is the one globally direct committed
    def get_global_commits():
        return listof(link[2], link in blocklinks(), is_global_commit(link[0], link[1]))

    def safe_round(rnd):
        violations = setof((cert_block, weak_cert_block), cert_block in cert_blocks(rnd), weak_cert_block in
weak_cert_blocks(rnd), cert_block != weak_cert_block)
        if len(violations):
            for cert_block, weak_cert_block in violations:
                logger.info('Block %s was certified in round %d but block %s also received f + 1 non-byzantine
votes in the same round' % (cert_block, rnd, weak_cert_block))
        return len(violations) == 0

    # check that b1 -->* b2
    def extends(b1, b2, graph):
        children = set(graph[b1])
        if b2 in children:
            return True
        # BFS
        while len(children):
            c = children.pop()
            if c not in graph:
                continue
            if b2 in graph[c]:
                return True
            children.update(graph[c])
        return False

    def check_safety():
        # property 1
        if not each(rnd in range(nrounds), has=safe_round(rnd)):
            logger.info('Property 1 violated')
        else:
            logger.info('Property 1 upheld')
        # property 2
        global_commits = get_global_commits()
        links = setof((link[0], link[2]), link in blocklinks())
        graph = {}
        for block, parent in links:
            if parent not in graph:
                graph[parent] = [block]
            else:
                graph[parent].append(block)
        violated = False
        for i in range(len(global_commits) - 1):
            for j in range(i+1, len(global_commits)):
                b1, b2 = global_commits[i], global_commits[j]
                b1_first = extends(b1, b2, graph)
                b2_first = extends(b2, b1, graph)
                if not (b1_first ^ b2_first):
                    violated = True
                    break
            if violated:
                break
```

```
        if violated:
            logger.info('Property 2 violated')
        else:
            logger.info('Property 2 upheld')
    def check_liveness():
        # property 3
        # technically the wording of property 3 means this hold true for all rounds after gst, not just the one
immediately after
        honest = {procs[0] for ident, procs in id_to_process.items() if len(procs) == 1}
        if not each(r in honest, has=some(received(('executor', ('committed', (_, rnd, _))), from_=_r), has=
rnd > gst)):
            logger.info('Property 3 could be violated')
        else:
            logger.info('Property 3 upheld')
        # property 4
        violated = False
        for rnd in range(gst, nrounds - 2):
            honest_leaders = True
            for i in range(3):
                if leaders[rnd + i] in twins:
                    honest_leaders = False
                    break
            if honest_leaders:
                time = first_to_enter[rnd]
                blocks = setof(msg[1][1].block.id, received(msg, from_=r), r in honest and msg[1][0] ==
'proposal' and msg[1][1].block.round == rnd)
                if len(blocks) > 1:
                    logger.info('honest validator sent equivocating proposals (should never happen)')
                    violated = True
                    break
                elif len(blocks) == 0:
                    logger.info('honest validator didnt send proposal in round after GST (should never
happen)')
                    violated = True
                    break
                block_id = blocks.pop()
                for ident in test_config.replicas:
                    if ident in twins:
                        continue
                    proc = id_to_process[ident][0]
                    # should always have length of 1 anyways
                    log_times = sorted(listof(time, received(('executor', ('committed', (_block_id, _rnd,
time))), from_=_proc)))
                    if not len(log_times):
                        logger.info('Block %s proposed during GST never committed by %s' % (block_id, ident))
                        violated = True
                        break
                    log_time = log_times[0]
                    if (log_time - time).total_seconds() > 7 * transmission_delay_bound:
                        violated = True
                        break
            if violated:
                break
        if violated:
            logger.info('Property 4 violated')
        else:
            logger.info('Property 4 upheld')

    def run():
        logger.debug(f'Starting run with {num_replicas} replicas,{num_faulty} faulty replicas')
        start(replicas)
```

```
        # wait until every replica has sent a message in the last round
        done_msg = (('done',), 'executor')
        if await(each(r in replicas, has=some(received(m, from_=_r), has=type(get_round(m[1])) == int and
get_round(m[1]) >= min(nrounds, stopper_round + 1) - 1))):
            if stopper_round < nrounds:
                logger.info('Reached quorumless round')
                logger.info('Waiting for further progression')
                if await(some(received(m), has=get_round(m[1]) and type(get_round(m[1])) == int and
get_round(m[1]) > stopper_round)):
                    logger.info('Quorumless progress property violated: a replica progressed past a round with
no quorum')
                elif timeout(transmission_delay_bound * 4 * 7):
                    logger.info('Quorumless progress property upheld: no replicas progressed passed quorumless
round')
                super().send(done_msg, replicas)
                check_safety()
                logger.info('Not checking liveness due to quorumless round')
            else:
                super().send(done_msg, replicas)
                logger.info('All rounds completed')
                check_safety()
                check_liveness()
        elif timeout(t):
            logger.info('Entire test case timed out')
            super().send(done_msg, replicas)
        send(('done',), to=main_exec)

    # TODO is it okay to block sync messages between partitions?
    #   probably but that could stress the sync manager code
    def can_send(src, dst, r, override=False):
        if r == 'stable':
            return True
        part = partitions[r]
        if part[src] != part[dst]:
            return False
        if override:
            return True
        for e in test_config.rounds[r].exceptions:
            if process_to_id[src] == e.src and process_to_id[dst] == e.dst:
                return False
        return True

    # DPI
    def get_round(msg):
        msg_type = msg[0]
        if msg_type == 'proposal':
            return msg[1].block.round
        elif msg_type == 'vote':
            return msg[1].vote_info.round
        elif msg_type == 'timeout':
            return msg[1].tmo_info.round
        # never block sync requests or sync responses
        # otherwise replicas are stuck until they are sent a TC
        # from a future round and advance rounds
        elif msg_type[:5] == 'sync_':
            return msg[1].rnd

    def receive(msg=M, from_=sender):
        #logger.info('receive', M, sender)
        dst, msg = M
        if type(dst) == list:
```

```
            for d in dst:
                receive_handler((d, msg), sender)
        else:
            receive_handler(M, sender)

    def receive_handler(M, sender):
        dst, msg = M
        if dst == 'executor':
            # this is a committed message which just needs to be stored in the message history
            # for liveness checking at the end
            return
        rnd = get_round(msg)
        src = process_to_id[sender][0]
        if src not in twins:
            first_to_enter[rnd] = min(first_to_enter.get(rnd, datetime.max), datetime.now())
        if rnd == nrounds:
            return
        if rnd == nrounds - 1 and msg[0] == 'vote':
            # prevent leader in round past GST from committing block
            return
        timeout_override = False
        if msg[0] == 'timeout':
            k = (src, dst, rnd)
            timeouts[k] = timeouts.get(k, 0) + 1
            # let through timeout messages on the third try
            if timeouts[k] > 2:
                timeout_override = True
        receivers = id_to_process[dst]
        for receiver in receivers:
            if can_send(sender, receiver, rnd, timeout_override):
                super().send((msg, src), receiver)
            else:
                logger.info('dropping %s message from %s to %s in round %d' % (msg[0], src,
process_to_id[receiver], rnd))

class MainExec(process):
    def setup(fname):
        self.test_cases = []
        self.exec_config_name = os.path.basename(fname).split('.')[0]
        logdir = 'log/' +  exec_config_name
        if not os.path.isdir(logdir):
            os.mkdir(logdir)
        ledgerdir = 'ledger/' + exec_config_name
        if not os.path.isdir(ledgerdir):
            os.mkdir(ledgerdir)
        self.logger = createLogger(__name__, logdir + '/main.log', 'MainExec')
        logger.info('Writing logs in directory %s' % logdir)
        logger.info('Writing ledgers in directory %s' % ledgerdir)

        parse_tests()

        logger.info('Parsed %d test cases from %s' % (len(test_cases), exec_config_name))

    def parse_tests():
        if fname[-3:] == '.py' or fname[-3:] == '.da':
            modname = os.path.dirname(fname).replace('/', '.') + '.' + exec_config_name
            mod = importlib.import_module(modname)
            test_cases = mod.test_cases
        elif fname[-4:] == 'json':
            with open(fname, 'r') as file:
                originals = json.loads(file.readline())
```

```
                twins = json.loads(file.readline())
                bugs = json.loads(file.readline())
                for idx,test_case in enumerate(file):
                    rounds = deserialize(test_case)
                    transmission_delay_bound = 0.8
                    seed = 0
                    name = f'test{idx}'
                    timeout = 30
                    gst = len(rounds) - 7
                    new_test = TestConfig(originals, twins,
rounds,transmission_delay_bound,seed,name,timeout,gst,bugs)          # produce 1 testconfig() namedtuple for
each test case
                    test_cases.append(new_test)
        else:
            logger.info('we only support .py and .json configuration files')

    def run():
        i = 0
        for i, test_config in enumerate(test_cases):
            # DEBUGGING:
            #if i == 2:
            #    break
            r = new(TestCaseExec, (exec_config_name, self, test_config))
            logger.info('Starting test case %s' % test_config.name)
            start(r)
            await(countof(m, received(m), m == ('done',)) == i + 1)
            logger.info('Finished test case %s' % test_config.name)
        logger.info('Finished running all test cases from %s' % exec_config_name)

def main():
    config(channel = 'reliable')

    if not(os.path.isdir("log")):
        os.mkdir("log")
    if len(sys.argv) != 2:
        print('Usage: testexec.da INPUT_FILE')
        sys.exit(1)
    testexec = new(MainExec, (sys.argv[1],))
    start(testexec)
```

TestConfig specification

```
from collections import namedtuple
from enum import Enum

# TestConfig.replicas = ['a', 'b', 'c', 'd']
# TestConfig.twins = ["d'"]
# use d and d' in the partitions
# TestConfig.rounds[0].partitions = [["a", "b", "d"], ["c", "d'"]]
GenConfig = namedtuple('GenConfig', ['tests', 'R', 'P', 'K', 'C', 'L', 'E', 'N', 'F', 'random_partitions',
'random_leaders', 'random_configurations', 'allow_non_faulty_leaders', 'allow_quorumless_partitions',
'out_file'])
TestConfig = namedtuple('TestConfig', ['replicas', 'twins', 'rounds', 'transmission_delay_bound', 'seed',
'name', 'timeout', 'gst', 'bugs'], defaults=(None))
Round = namedtuple('Round', ['leader', 'partition', 'exceptions'])
Except = namedtuple('Except', ['src', 'dst', 'msg_type'])
class MsgType(str,Enum):
```

```
        Proposal = 'proposal'
        TimeOut = 'timeout'
        Vote = 'vote'
        Wildcard = '*' # matches all message types
```

ReplicaTestExec Pseudocode

```
from replica import Replica
import pacemaker
import leaderelectiontestexec as leaderelection
import random
import safety
from config import *
from time import sleep
import mempool

class ReplicaTestExec(process, Replica):
    def setup(
        executor,
        replica_ids,
        transmission_delay_bound,
        public_keys,
        private_key,
        ident,
        seed,
        personal_seed,
        num_faulty,
        leaders,
        nrounds,
        testcase_name,
        exec_config_name,
        bugs
    ):
        super().setup(
            executor,
            replica_ids,
            transmission_delay_bound,
            public_keys,
            private_key,
            ident,
            seed,
            num_faulty,
            testcase_name,
            exec_config_name,
            bugs
        )
        random.seed(seed)
        txns = []
        alphabet = [chr(x) for x in range(ord('a'), ord('z') + 1)]
        for _ in range(nrounds * mempool.BLOCK_SIZE * 2):
            txn = [random.choice(alphabet) for _ in range(6)]
            uuid = '%xu' % random.getrandbits(16*8)
            txns.append((uuid, ''.join(txn)))
        random.seed(personal_seed)
        random.shuffle(txns)
        random.seed(seed)
        for uuid, txn in txns:
```

```
            mempool.requests[uuid] = ((txn, 'client0', uuid), None)
        leaderelection.reputation_leaders = leaders
        self.u = ident

    def receive(msg=M, from_=sender):
        super().handle_msg(*M)

    # to is always a identifier string (not a process)
    def send(m, to):
        super().send((to, m), executor)
```

LeaderElectionTestExec Pseudocode

```
# this variable is set within the ReplicaTestExec setup method
reputation_leaders = {}

# defined here to minimize changes to the Replica code
def update_leaders(qc):
    pass

def get_leader(round):
    return reputation_leaders[round]
```

Summary of Replica changes

```
Modify any code that uses the send method so that we call it with string identifiers instead of process objects
-import leaderelection
+import leaderelectiontestexec as leaderelection

 class Replica(process):
     def setup(
+        executor, # only passed in here so we can send committed log messages
+        txns, # to be given to the mempool module
         ):
+            for uuid, txn in txns:
+                mempool.requests[uuid] = txn
 def process_certificate_qc(qc):
     if qc != None:
         committed_payload, commit_state_id = blocktree.process_qc(qc)
         if committed_payload != None:
+             # tell TestExec that we have locally committed this block
+             block_id = qc.vote_info.parent_id
+             rnd = qc.vote_info.parent_round
+             if not sent(('committed', (_block_id, _, _))):
+                 send(('committed', (block_id, rnd, datetime.now())), to=executor)
```

Contributions

Andrew Burford: test executor pseudocode and design write up
Jiawei Qian: safety and liveness property checking
Ethan DeTurk: test generator pseudocode and design