Final Report for Replicated Concurrency Control and Recovery Project

Team Members:

- Shika Rao (sr7463)
- Khushboo . (kx2252)

Programming Language: Python

1. High-Level Architecture

The project simulates a replicated database system with snapshot isolation, available-copies replication, and first-committer-wins (FCW) conflict handling. The code is organized into three main modules:

- data_structure.py : domain data types (Version, VariableHistory, Site, Transaction).
- repcrec.py : core RepCRec engine implementing the concurrency-control and replication logic.
- runner.py : command-line driver: reads test scripts, creates a fresh engine per test, and invokes RepCRec methods.

```
runner.py
   ↓  (drives)
repcrec.py (RepCRec engine)
   ↓ uses
data_structure.py
   ↓ defines
{ Transaction, Site, VariableHistory, Version}
```

2. Component Overview

Data Structures used

Version

```
@dataclass
class Version:
```

```
    value: int
    commit_time: int
    writer: Optional[str]
    sites: Set[int]
```

- Represents a committed version of a variable.
- `commit_time` is a global logical time used to enforce snapshot isolation.
- `sites` lists the sites that have this version (for replicated variables).

VariableHistory

```
@dataclass
class VariableHistory:
    name: str
    versions: List[Version] = field(default_factory=list)

    def latest_before(self, ts: int) -> Optional[Version]:
        ...
```

- Maintains the history of committed versions of a variable (`x1` … `x20`).
- `latest_before(ts)` is the core snapshot helper: it picks the most recent committed version ≤ `ts`, and is used during reads to implement snapshot isolation.

Site

```
@dataclass
class Site:
    id: int
    is_up: bool = True
    data: Dict[str, int] = field(default_factory=dict)
    can_read: Dict[str, bool] = field(default_factory=dict)
    failure_times: List[int] = field(default_factory=list)
    recovery_times: List[int] = field(default_factory=list)

    def fail(self, time: int) -> None: ...
    def recover(self, time: int, is_replicated) -> None: ...
```

- Represent a physical site (1–10).

- Track whether the site is up or down.
- Track committed data visible at that site in `data[var]`.
- Track readability of replicated variables after recovery via `can_read[var]`:
    - For replicated vars, `can_read[var]` is `False` after recovery until a new commit touches that var at the site.
    - For non-replicated vars, `can_read[var]` is `True` immediately after recovery.
- Maintain `failure_times` and `recovery_times`, which the engine uses to check "site up continuously between time A and time B."

Transaction

```
@dataclass
class Transaction:
    tid: str
    start_time: int
    status: str = "active"  # active | committed | aborted | waiting
(conceptually)
    read_vars: Set[str] = field(default_factory=set)
    write_buffer: Dict[str, int] = field(default_factory=dict)
    write_sites: Dict[str, Set[int]] = field(default_factory=dict)
    site_write_times: Dict[int, int] = field(default_factory=dict)
    commit_time: Optional[int] = None
```

Responsibilities:

- Represent a logical transaction `Ti`:
    - `start_time` is the snapshot time.
    - `write_buffer` holds uncommitted writes.
    - `write_sites[var]` stores which sites should receive that variable at commit.
    - `site_write_times` tracks the earliest write time to each site, needed for the available-copies failure rule.
- `status` is updated by the engine as commands execute.

Replicated Concurrency Control Engine

```
class RepCRec:
    def __init__(self):
        self.time = 0
        self.sites: Dict[int, Site] = {}
```

```
self.vars: Dict[str, VariableHistory] = {}
self.txns: Dict[str, Transaction] = {}
self.var_last_writer: Dict[str, Tuple[str, int]] = {}
self._init_sites_and_vars()
```

Global State

- `time`: global logical clock incremented once per input command line.
- `sites`: all `Site` objects.
- `vars`: all `VariableHistory` objects for `x1`...`x20`.
- `txns`: active/finished `Transaction` objects.
- `var_last_writer[var] = (tid, commit_time)`: used for **first-committer-wins**.

`begin(tid)`

- `execute_line()` parses a line of the test script and routes it
- Advances `self.time` by 1 for each real command.
  Creates a new `Transaction` with current `self.time` as `start_time`.
- Stores it in `self.txns`.

`read(tid, var)`

Implements snapshot isolation and available copies:

1. Fetch the transaction; ignore reads for aborted/committed transactions.
2. Use `VariableHistory.latest_before(txn.start_time)` to get the snapshot version.
3. If variable is non-replicated:
   - Read from its home site if it is up and the snapshot version exists there.
4. If variable is replicated:
   - Try to find a site s such that:
     s is up, `var` is readable at s (`can_read[var]` is `True` if replicated), s is in `snapshot.sites`, and `site_up_continuously(s, snapshot.commit_time, txn.start_time)` holds.
   - If none exists:

     Either abort immediately (for some cases), or conceptually mark the transaction as “waiting for a snapshot”.

`write(tid, var, value)`

1. Determine sites that currently hold the variable and are up:
    - Replicated: all up sites.
    - Non-replicated: only home site, if up.
2. If no such sites: transaction is aborted (project spec says it should “wait”; we implement an eager abort as a simplification).
3. Buffer the write in `txn.write_buffer[var] = value`.
4. Record `txn.write_sites[var]` and update `txn.site_write_times[site]` for available-copies failure rules.
5. Print that the write is buffered (no visible state change yet).

`end(tid)`

1. Available copies abort rule:
   For each site `s` that this transaction wrote to:
    - If `s` failed at any time `f` with `write_time < f ≤ current_time`, abort the transaction.

2. First-committer-wins (FCW):
   For each variable `var` in `write_buffer`:
    - If `var_last_writer[var] = (other_tid, other_ctime)` with `other_ctime > txn.start_time` and `other_tid != tid`, abort.

3. SSI / cycle detection: RW-cycle detection component:
    - The engine tracks read-write relationships between transactions and aborts a transaction if it would close a dangerous cycle. This is implemented conservatively to avoid over-aborting: FCW + snapshot isolation already enforces most constraints.

4. If none of the above cause an abort, commit:
    - Set `txn.commit_time = current_time`.
    - For each `(var, value)` in `write_buffer`:
        - Apply the update to each site in `txn.write_sites[var]` that is up.
        - For replicated variables, mark `site.can_read[var] = True` on those sites.
        - Append a new `Version` to `vars[var].versions`.
        - Update `var_last_writer[var] = (tid, commit_time)`.
    - Set `txn.status = "committed"` and print “Ti commits”.

Main Runner file

- Input test file with all the test cases. (test.txt contains all 25 sample test cases)
- Command: python3 runner.py test.txt

3. Reprozip replay in new directory:

```
reprounzip directory setup repcrec.rpz repro_test_run
reprounzip directory run repro_test_run
```