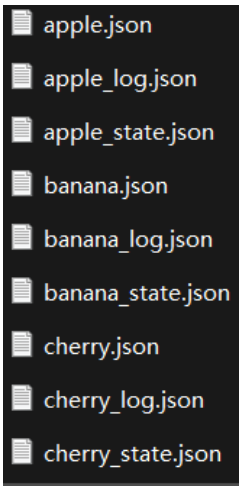
Problem 1

Firstly, reliable storage is implemented by storing data in a JSON file. The acceptor states are recorded in a file named {*site_id*}*_state.json*, including the *promised_id*, *accepted_id*, and *accepted_value* for each slot. The file {*site_id*}*_log.json* maintains a sequence of decided commands. The key-value pairs are stored in {*site_id*}*.json*. After running public test case 1, the files will be like



Secondly, when a site restarts, it checks for the existence of the persistent storage files. If the files exist, it loads the acceptor state, log, and key-value store from corresponding JSON files.

```python
# Load or initialize stable storage
log_filename = f'{site_id}_log.json'
state_filename = f'{site_id}_state.json'
kv_store_filename = f'{site_id}.json'

if os.path.exists(log_filename):
    with open(log_filename, 'r') as f:
        log = json.load(f)
else:
    log = []

if os.path.exists(state_filename):
    with open(state_filename, 'r') as f:
        state = json.load(f)
    # Ensure keys are integers
    state = {int(k): v for k, v in state.items()}
else:
    state = {}

if os.path.exists(kv_store_filename):
    with open(kv_store_filename, 'r') as f:
        kv_store = json.load(f)
else:
    kv_store = {}
```

Then, the function *fill_holes*() synchronizes the log contents to fulfill the gaps occurring while the site is down.

```
def fill_holes():
    max_slot = get_max_slot()
    for i in range(len(log), max_slot + 1):
        log.append(None)
    for i in range(len(log)):
        if log[i] is None:
            # Request the decided value for the slot
            value = request_value(i)
            if value is not None:
                learner({'slot': i, 'value': value})
            else:
                # No decided value; propose NO_OP to fill the slot
                propose_no_op(i)
```

```
def get_max_slot():
    max_slot = len(log) - 1
    for host_id, info in known_hosts.items():
        if host_id == site_id:
            continue
        addr = (info['ip_address'], info['udp_start_port'])
        request_message = {
            'type': 'max_slot_request',
            'from': site_id
        }
        send_message(request_message, addr)
    # Process own max_slot_response immediately
    message_queue.append({'type': 'max_slot_response', 'max_slot': len(log) - 1, 'from': site_id})
    start_time = time.time()
    while time.time() - start_time < TIMEOUT:
        with mutex:
            for msg in message_queue:
                if msg['type'] == 'max_slot_response':
                    if msg['max_slot'] > max_slot:
                        max_slot = msg['max_slot']
                    message_queue.remove(msg)
        time.sleep(0.01)
    return max_slot
```

Finally, the function *rebuild_kv_store*() reconstructs the key-value pairs based on recovered log.

```
def rebuild_kv_store():
    global kv_store
    kv_store = {}
    with mutex:
        for entry in log:
            if entry and entry.get('operation') != 'NO_OP':
                apply_to_kv_store(entry)
    save_kv_store()
```

Problem 2

1. **Correctly getting the max slot (*max_slot*):** The recovered site must learn the max slot number used by the most up-to-date active site, because it allows the recovered site to understand how many entries it needs to learn or recover. This is achieved by sending *max_slot_request* messages to other sites and collecting responses.
2. **Correctly fulfilling holes (*fill_holes*()):** If the recovered site cannot learn a decided value for a slot, it must initiate a Paxos instance to propose a value for that slot and ensure it reaches consensus. This ensures that the recovered site does not have gaps in its log. By proposing a *NO_OP*, the site can fill the hole without altering the state of the key-value store, maintaining consistency and allowing progress.
3. **Stable storage:** The site must persistently save log, acceptor state, key-value store to JSON files before/after recovery. This ensures that if the site fails, it can recover to the same state without losing the progress made during the previous recovery.
4. **Handling concurrent proposals:** The recovery algorithm must handle concurrent proposals for the same slots by other sites. Other sites may also be proposing values for the same slots. The recovered site must correctly participate in the Paxos protocol to handle such scenarios, ensuring that only one value is decided per slot.

Problem 3

Yes. An example is as follows.

1. Initially, there are two sites A and B having null logs and key-value pairs.
2. Site A proposes and reaches consensus on operation X (*put key1 value1*). It updates its log: $log[0] = X$. Then, site A proposes and reaches consensus on another operation Y (*put key2 value2*). It updates its log: $log[1] = Y$.
3. Site B is down and doesn't receive messages from A. It is not ware of the previous decisions made by A in 2.
4. Site A proposes and reaches consensus on operation Z (*append key1 value3*). It updates its log: $log[2] = Z$.
5. However, site B has recovered and received the decision of operation Z.
6. Based on the *learner*(*message*) function below, site B checks if $log[2]$ is *None* and updates $log[2] = Z$. At this point, B will execute Z immediately.
7. Then, B executes *fill_holes*() mentioned above, where it updates its log with X and Y, and then applies them to the key-value pair.

As a result, the order of operations at A is $X \rightarrow Y \rightarrow Z$, while the order is $Z \rightarrow X \rightarrow Y$ at B.

```
def learner(message):
    slot = message['slot']
    value = message['value']
    with mutex:
        if slot >= len(log):
            log.extend([None] * (slot - len(log) + 1))
        if log[slot] is None:
            log[slot] = value
            save_log()
            # Update KV store
            if value.get('operation') != 'NO_OP':
                apply_to_kv_store(value)
                save_kv_store()
    # Print received decide message to stderr
    print(f'received decided value {value} for slot {slot}', file=sys.stderr)
```

Problem 4

1. In the Paxon parliament, the processes are **legislators**, where proposers are those who propose decrees, acceptors are those who vote on proposals, and learners are those who learn about the decrees that have been passed. A legislator "**fails**" by becoming absent from parliamentary proceedings, possibly because of being delayed due to travel, attending to personal affairs on their estates, or any other reasons. A legislator "**recovers**" by returning to participate in parliament. After returning, they resume their duties, including proposing new decrees, voting on proposals, and catching up on decrees that were passed in their absence. They have "**stable storage**" by recording information in their notebooks, which ensure that a legislator can retain knowledge of their previous commitments. Promises are the highest numbered proposal, less than which they have agreed not to accept, and accepted proposals are details of any accepted proposals.
2. In the Paxon parliament, messaging is conducted through notes carried by messengers. Legislators send and receive notes to propose decrees, request votes, and inform others of decisions. The delivery is **asynchronous** because of the varying travel speeds, distance, messenger availability, and other delays. There is no guarantee of time of delivery. It's **unreliable** because notes can be lost due to messenger errors, accidents, or other unforeseen events. Notes can be **duplicated** through sending multiple copies for reliability or delivering the same note more than once by massagers. There is **no corruption** since the content of the notes remains intact even though they can be lost, so legislators can trust the information received is exactly what is sent.