Problem 1:

If concurrent requests are allowed, our system model will NOT ensure linearizability. Assume that two clients send *append* commands simultaneously. The primary server will respond to one client after replicating to the backup, and it is possible that the other *append* is executed before the backup is completed, which leads to a scenario where the outputs look different at two clients. The first client will not see the results of the second *append* even if that *append* is executed before the response. This implies that a *get* command may not be responded by the latest *put*. Therefore, it is not linearizable.

```
if command == 'put':
    self.kv_store[key] = [value]
    self.replicate_to_backup(msg)
    response = {'type': 'success'}
    self.sock.sendto(json.dumps(response).encode('utf-8'), addr)
elif command == 'get':
    if key in self.kv_store:
        response = {'type': 'success', 'value': self.kv_store[key]}
    else:
        response = {'type': 'success', 'value': None}
    self.sock.sendto(json.dumps(response).encode('utf-8'), addr)
elif command == 'append':
    if key in self.kv_store:
        self.kv_store[key].append(value)
    else:
        self.kv_store[key] = [value]
    self.replicate_to_backup(msg)
    response = {'type': 'success'}
    self.sock.sendto(json.dumps(response).encode('utf-8'), addr)
```

```
def replicate_to_backup(self, msg):  3 usages
    backup_id = self.current_view['backup']
    if backup_id == self.server_id or not backup_id:
        return
    if backup_id in self.known_hosts:
        replication_msg = {'type': 'replicate', 'command': msg['command'],
                           'key': msg.get('key'), 'value': msg.get('value')}
        self.sock.sendto(json.dumps(replication_msg).encode('utf-8'), (get_port_ip(self.known_hosts, back
        try:
            self.sock.settimeout(0.4)
            data, addr = self.sock.recvfrom(4096)
            response = json.loads(data.decode('utf-8'))
            if response.get('type') != 'ack':
                pass
        except socket.timeout:
            get_view = {'type': 'get_view'}
            self.sock.sendto(json.dumps(get_view).encode('utf-8'), (get_port_ip(self.known_hosts,  server_id
```

Problem 2:

Yes. Since the messaging is reliable and there is always one single primary key-value server, linearizability is guaranteed by the sequential processing of operations. Upon any input from clients or CLI, the view server will check if the primary and backup are alive. When a primary is down, the backup with identical replica will replace it. Even some client sends request at this time, a correct output will be responded by the new primary server. Hence, for any *get* operation, the system should return the latest *put* result. In the case where all key-value servers are down, the linearizability will become meaningless. Under this setting, FIFO manner ensures that all operations are consistent.

```
def update_view(self):  2 usages
    servers = [server for server in self.current_view.values() if server is not None]
    if len(servers) != 0:
        result = self.server_status.check_servers(servers)
        if False in result:
            result = self.server_status.ping_servers(self.servers)
            fserver = None if len(result) == 0 else result[0]
            sserver = None if len(result) == 1 else result[1]
            if len(servers) == 1 and self.current_view['backup'] == None:
                self.current_view['primary'] = fserver
            elif len(servers) == 2 and result == [False, True]:
                self.current_view['primary'] = fserver
            elif len(servers) == 2 and result == [True, False]:
                self.current_view['backup'] = sserver
            elif len(servers) == 2 and result == [False, False]:
                self.current_view['primary'] = fserver
                self.current_view['backup'] = sserver
```

```
def replicate_to_backup(self, msg):  3 usages
    backup_id = self.current_view['backup']
    if backup_id == self.server_id or not backup_id:
        return
    if backup_id in self.known_hosts:
        replication_msg = {'type': 'replicate', 'command': msg['command'],
                           'key': msg.get('key'), 'value': msg.get('value')}
        self.sock.sendto(json.dumps(replication_msg).encode('utf-8'), (get_port_ip(self.known_hosts, ba
        try:
            self.sock.settimeout(0.4)
            data, addr = self.sock.recvfrom(4096)
            response = json.loads(data.decode('utf-8'))
            if response.get('type') != 'ack':
                pass
        except socket.timeout:
            get_view = {'type': 'get_view'}
            self.sock.sendto(json.dumps(get_view).encode('utf-8'), (get_port_ip(self.known_hosts, serve
```

Problem 3:

No. In our implementation, the view server will verify if the primary is alive upon any input event, so if the primary is down at that time, the view server will simply set backup as primary. As all information is stored in memory, the recovered key-value server has

no idea of its role after recovery. If it can recover before the verification process, the view server will not be aware of the crash and just keep the status. Therefore, it is impossible for multiple sites to think they are the primary at the same time.

```python
class KVServer:  1 usage
    def __init__(self, server_id, known_hosts, ip, port):
        self.server_id = server_id
        self.known_hosts = known_hosts
        self.role = 'none'
        self.kv_store = {}
        self.lock = Lock()
        self.current_view = {'primary': None, 'backup': None}
        self.start(ip, port)

    def start(self, ip, port):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind((ip, port))
        self.report_to_view_server(ip, port)
        threading.Thread(target=self.listener, daemon=True).start()
        threading.Thread(target=self.handle_commands, daemon=True).start()
```

```python
def handle_commands(self):  1 usage
    while True:
        cmd = sys.stdin.readline().strip()
        if cmd == 'view':
            with self.lock:
                self.update_view()
                primary = self.current_view['primary'] if self.current_view['primary'] else 'none'
                backup = self.current_view['backup'] if self.current_view['backup'] else 'none'
            print(f'primary: {primary} backup: {backup}')
            sys.stdout.flush()
```

```python
# Create a new socket bound to the available port
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
    try:
        sock.bind((self.ip, local_port))
        sock.settimeout(0.4)
        sock.sendto(message_bytes, (ip, port))
        response, _ = sock.recvfrom(1024)
        response_data = json.loads(response.decode('utf-8'))

        if response_data.get('type') == 'pong' and response_data.get('server_id') == server_id:
            return True
    except socket.timeout:
        return False
    except (json.JSONDecodeError, KeyError):
        return False
    finally:
        # Release the port back to the port manager
        self.port_manager.push(local_port)
```

```
def __init__(self, known_hosts, ip, port, server_status):
    self.known_hosts = known_hosts
    self.server_status = server_status
    self.servers = set()
    self.current_view = {'primary': None, 'backup': None}
    self.lock = threading.Lock()
    self.start(ip, port)
```

Problem 4:

If the messages can be lost but there is no failure, we will implement a reliable transmission protocol such as TCP instead of using UDP. Clients will retransmit requests for missing messages, where an ack is not received within the timeout interval. The same protocol should be implemented on view and key-value servers as well. Moreover, the backup server should periodically check with the primary to ensure their copies are identical. These mechanisms will ensure the linearizability.

```
self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
self.sock.bind((ip, port))
```

Problem 5:

In the context of this assignment, our system could guarantee the consistency and partition-tolerance. The requests are in FIFO order, so every *get* operation will receive the most recent *put*, which ensures consistency. Partition-tolerance is guaranteed by the code specification and design of the system. However, the system does not ensure availability because it is possible that all key-value servers are down.

```
if command == 'put':
    self.kv_store[key] = [value]
    self.replicate_to_backup(msg)
    response = {'type': 'success'}
    self.sock.sendto(json.dumps(response).encode('utf-8'), addr)
elif command == 'get':
    if key in self.kv_store:
        response = {'type': 'success', 'value': self.kv_store[key]}
    else:
        response = {'type': 'success', 'value': None}
    self.sock.sendto(json.dumps(response).encode('utf-8'), addr)
elif command == 'append':
    if key in self.kv_store:
        self.kv_store[key].append(value)
    else:
        self.kv_store[key] = [value]
    self.replicate_to_backup(msg)
    response = {'type': 'success'}
    self.sock.sendto(json.dumps(response).encode('utf-8'), addr)
```

```
def update_view(self):  2 usages
    servers = [server for server in self.current_view.values() if server is not None]
    if len(servers) != 0:
        result = self.server_status.check_servers(servers)
        if False in result:
            result = self.server_status.ping_servers(self.servers)
            fserver = None if len(result) == 0 else result[0]
            sserver = None if len(result) == 1 else result[1]
            if len(servers) == 1 and self.current_view['backup'] == None:
                self.current_view['primary'] = fserver
            elif len(servers) == 2 and result == [False, True]:
                self.current_view['primary'] = fserver
            elif len(servers) == 2 and result == [True, False]:
                self.current_view['backup'] = sserver
            elif len(servers) == 2 and result == [False, False]:
                self.current_view['primary'] = fserver
                self.current_view['backup'] = sserver
```