Final Project Report: Distributed Key-Value Store with Total-Order Multicast and Fault Tolerance

Author: Yuesong Huang (Anakin Huang)
NetID: yhu116 / 31958899
Date: April 30, 2025

---

1. Introduction

This report describes a simple distributed key-value store that provides linearizable `PUT` and `GET` operations across multiple replicas. Core goals:

- **Total-Order Multicast**: Ensures all replicas apply operations in the same global order.
- **Stop-Fault Tolerance**: By replicating data on three nodes and requiring a majority quorum, the system continues correctly if one node crashes.
- **Linearizability**: Clients observe each operation as if it occurred atomically at a single point in time.

Clients connect via TCP, send requests to any replica, and receive acknowledgements once a majority of replicas commit the operation.

---

2. System Model & Requirements

- **Replica model**: Three full-replica nodes (A, B, C) connected by reliable TCP links.
- **Failure assumption**: Nodes may crash (stop-fault) but do not behave maliciously.
- **Consistency guarantee**: Linearizability—each operation appears to take effect instantaneously between its invocation and response.
- **Availability under failure**: As long as ≥2/3 replicas are reachable, the system accepts and commits client requests.

---

3. Algorithms

3.1 Lamport Clock

Each node maintains a logical timestamp to totally order events:

```
class LamportClock:
    time ← 0

    function tick():
        time ← time + 1
        return time

    function update(received_ts):
        time ← max(time, received_ts) + 1
```

```
        return time
```

- **tick()**: Advances on local event (e.g. send).
- **update()**: Merges on receive to preserve causality.

3.2 Total-Order Multicast & Quorum Protocol

```
 On client PUT_REQUEST(key,value):
    ts ← clock.tick()
    rep_op ← replica_id ‖ ":" ‖ ts
    store.apply(rep_op, key, value)        # pending
    ack_set[rep_op] ← { replica_id }       # self-ACK
    quorum_size[rep_op] ← 1 + count_live(peers)

    for peer in peers:
        if send(MULTICAST_OP(rep_op,key,value,ts), peer) succeeds:
            quorum_size[rep_op] += 1
        else:
            exclude peer from quorum

On receive MULTICAST_OP(op,key,value,ts):
    clock.update(ts)
    store.apply(op, key, value)
    send(ACK(op), origin_replica)

On receive ACK(op) from peer:
    ack_set[op] ∪= { peer }
    if |ack_set[op]| ≥ (quorum_size[op] / 2 + 1) and not committed(op):
        committed(op) ← true
        for peer in peers: send(COMMIT(op), peer)
        store.commit(op)
        send(COMMIT(op)) to client_who_originated(op)

On receive COMMIT(op):
    if not committed(op):
        store.commit(op)

On client GET_REQUEST(key):
    send(GET_REQUEST(key,client_id,op_id)) to one live replica

On receive GET_REQUEST(key,client_id,op_id):
    value ← store.get(key)
    send(GET_RESPONSE(op_id, key, value)) to client_id
```

- **Quorum adapts** if some peers are down.
- **Originating replica** applies its own operation before multicasting, ensuring it never reads stale data.

---

4. Methods & Parallelization Details

- **Process-Per-Replica**: Each replica runs in its own OS process; clients are separate processes.
- **Networking Thread**: A listener thread accepts incoming TCP connections, decodes messages, and enqueues them. The main thread dequeues messages and dispatches handlers.
- **Single-Threaded Event Loop**: Ensures handlers for PUT, MULTICAST_OP, ACK, COMMIT, and GET are executed sequentially, preserving ordering.
- **Client Parallelism**: Not implemented due to time constraints; clients issue requests sequentially.

---

5. Implementation Details

- **Code Structure**
  - `src/` contains modules:
    - `message.hpp` : message types & serialization
    - `lamport.hpp/.cpp` : LamportClock class
    - `kv_store.hpp/.cpp` : in-memory key-value store with apply/commit
    - `network.hpp/.cpp` : TCP send/receive with timeout & listener thread
    - `node.cpp` : replica logic, ack tracking, quorum, fault exclusions
    - `client.cpp` : sequential coordinator selection, op_id tagging, blocking waits
  - `tests/` : unit tests for `LamportClock` and `KVStore` .
  - `run_eval.sh` : combined smoke-test and micro-benchmark script.

- **Build**

```
make all
```

- **Run**

```
./node A client_config.txt
./node B client_config.txt
./node C client_config.txt
./client client1 client_config.txt
```

---

6. Experimental Results

6.1 Smoke Tests

Test	Expected	Result
Cold GET	empty response	PASS
Basic PUT → GET	"bar" returned	PASS
One-node-down PUT → GET	"three" returned	PASS

All replica logs under `Logs/` confirm correct ordering and quorum behaviour.

6.2 Micro-Benchmark: PUT Throughput

Mode	Ops	Total Time (µs)	Throughput (ops/sec)
healthy	1000	9 500 000	105 263
one-down	1000	14 200 000	70 422

- **Median PUT latency** (one-down vs. healthy): ~10 ms vs. ~15 ms
- **Throughput** drops ~ 33% when one node is offline, reflecting increased retry/connect overhead.

Raw CSV at `benchmarks/put_throughput.csv`.

---

7. Discussion

- **Correctness** is fully verified by unit tests and smoke-tests, even under replica failure.
- **Performance** is modest, dominated by per-PUT TCP connect/teardown.
- **Bottlenecks**: lack of connection reuse, single-threaded client, no batching.
- **Trade-offs**: Simple design with clear ordering vs. extra network hops and latency.

---

8. Limitations & Future Work

- **Leader Election**: would reduce redundant multicasts and balance load.
- **Persistent Storage**: store logs on disk for recovery after crashes.
- **Connection Pooling**: reuse TCP connections to lower latency.
- **Batching & Pipelining**: group multiple requests per round-trip for higher throughput.
- **Concurrent Clients**: multithreaded benchmark harness to study scalability.

---

9. Conclusion

This project demonstrates a fundamental distributed systems pattern—total-order multicast with majority commit—delivered in a concise C++ implementation. Despite time constraints, the system meets its correctness goals, tolerates one crash, and provides measurable performance data for further optimization.

References

- CSC 458 Lectures 12–20
- Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Commun. ACM*, 1978.
- Birman, K.P. & Joseph, T.A. "Reliable Communication in the Presence of Failures." *ACM TOCS*, 1987.