# Distributed Key-Value Store Technical Report

## 1. System Architecture

### 1.1 Core Components

1. **Storage Layer** (storage.py): In-memory key-value store with JSON persistence
2. **Node Manager** (node\_manager.py): Handles node coordination and request routing
3. **API Server** (main.py): FastAPI-based REST interface
4. **Frontend Dashboard** (dashboard/): React-based monitoring interface

### 1.2 Node Configuration (config.json)

{

"nodes": [

{"id": "node\_1", "host": "127.0.0.1", "port": 8000},

{"id": "node\_2", "host": "127.0.0.1", "port": 8001},

{"id": "node\_3", "host": "127.0.0.1", "port": 8002}

],

"replication\_factor": 2

}

## 2. Data Distribution

### 2.1 Key Distribution (Consistent Hashing)

def get\_node\_for\_key(self, key: str) -> Node:

hash\_value = int(hashlib.md5(key.encode()).hexdigest(), 16)

healthy\_nodes = [n for n in self.nodes.values() if n.is\_healthy]

node\_index = hash\_value % len(healthy\_nodes)

return healthy\_nodes[node\_index]

1. Uses MD5 hashing for key distribution
2. Keys are assigned to nodes based on hash value modulo number of healthy nodes
3. Automatically rebalances when nodes become unhealthy

### 2.2 Data Replication

1. Primary-Secondary replication model
2. Replication factor configurable in config.json
3. Replica selection:

def get\_replica\_nodes(self, primary\_node: Node) -> List[Node]:

healthy\_nodes = [n for n in self.nodes.values() if n.is\_healthy and n.id != primary\_node.id]

replicas = []

for i in range(min(self.replication\_factor - 1, len(healthy\_nodes))):

replicas.append(healthy\_nodes[i])

return replicas

## 3. Storage Implementation

### 3.1 In-Memory Storage

class Storage:

def \_\_init\_\_(self):

self.\_store: Dict[str, Any] = {}

self.\_lock = threading.Lock()

self.\_start\_time = datetime.now()

1. Thread-safe operations using locks
2. Supports any JSON-serializable value type
3. Maintains uptime metrics

### 3.2 Persistence Layer

def \_save\_to\_disk(self) -> None:

try:

with open(self.\_storage\_file, 'w') as f:

json.dump(self.\_store, f, indent=2)

except Exception as e:

logger.error(f"Error saving data to disk: {e}")

1. Each node maintains its own JSON file
2. File location: data/{node\_id}\_storage.json
3. Automatic recovery on node restart

## 4. Request Handling

### 4.1 PUT Operations

@app.put("/store/{key}")

async def put\_value(key: str, item: KeyValue):

# 1. Determine target node

target\_node = node\_manager.get\_node\_for\_key(key)

# 2. Store on primary node

if target\_node.id == node\_id:

storage.put(key, item.value)

else:

response = await node\_manager.forward\_request(...)

# 3. Replicate to backup nodes

replicas = node\_manager.get\_replica\_nodes(target\_node)

for replica in replicas:

await node\_manager.forward\_request(...)

### 4.2 GET Operations

@app.get("/store/{key}")

async def get\_value(key: str):

# 1. Find responsible node

target\_node = node\_manager.get\_node\_for\_key(key)

# 2. Try primary node

if target\_node.id == node\_id:

value = storage.get(key)

else:

response = await node\_manager.forward\_request(...)

# 3. Try replicas if primary fails

if response is None:

replicas = node\_manager.get\_replica\_nodes(target\_node)

for replica in replicas:

response = await node\_manager.forward\_request(...)

## 5. Node Health Management

### 5.1 Node Health Tracking

class Node:

def \_\_init\_\_(self, id: str, host: str, port: int):

self.failed\_attempts = 0

self.max\_failures = 3

self.last\_heartbeat = datetime.now()

@property

def is\_healthy(self):

return self.failed\_attempts < self.max\_failures

### 5.2 Request Forwarding with Retry Logic

async def forward\_request(self, node: Node, method: str, path: str, \*\*kwargs):

max\_retries = 2

retry\_delay = 1

for attempt in range(max\_retries + 1):

if not node.is\_healthy:

return None

try:

# Attempt request

async with aiohttp.ClientSession(timeout=timeout) as session:

response = await session.request(...)

node.mark\_healthy()

return response

except:

node.mark\_failed()

await asyncio.sleep(retry\_delay)

## 6. Frontend Dashboard

## 

### 6.1 Node Status Monitoring

1. Real-time health status
2. Key count per node
3. Uptime tracking
4. Request success/failure indicators

### 6.2 Key Distribution Visualization

const KeyDistributionChart = ({ nodes, nodeStatuses }) => {

// D3.js-based donut chart showing:

// - Distribution of keys across nodes

// - Node health status

// - Total key count

}

## 7. Fault Tolerance

### 7.1 Node Failure Handling

1. Automatic failover to replica nodes
2. Health check-based node exclusion
3. Configurable retry attempts

### 7.2 Data Recovery

1. Automatic data reload from JSON files on restart
2. Replication ensures data availability when nodes fail

## 8. Limitations and Considerations

1. **Consistency Model**
2. Eventually consistent
3. No strict ordering guarantees
4. Possible temporary inconsistencies during replication
5. **Scalability**
6. Limited by in-memory storage
7. JSON persistence may become bottleneck
8. No automatic resharding
9. **Recovery**
10. Basic file-based recovery
11. No transaction log
12. Potential data loss during crashes
13. **Network Partition Handling**
14. Basic timeout-based detection
15. No complex partition resolution
16. Possible split-brain scenarios

## 9. Startup and Operation

### 9.1 Node Startup (run\_nodes.sh)

start\_node() {

local node\_id=$1

local port=$2

NODE\_ID=$node\_id /usr/local/bin/python3 main.py $node\_id &

# Health check

curl -s "http://127.0.0.1:$port/status"

}

### 9.2 Monitoring Endpoints

1. /status: Node health and metrics
2. /keys: List of stored keys
3. /node-info: Detailed node information