SYSTEM ARCHITECTURE

Distributed Key-Value Store

System Design Document

Document Version: 1.0
Classification: Internal Technical Documentation
Date: August 2025
Status: Final Draft

Executive Summary

This document provides a comprehensive technical overview of the Distributed Key-Value Store (KVStore) system - a high-performance, fault-tolerant distributed database designed for modern cloud-native applications. The system leverages BadgerDB as the storage engine and implements the Raft consensus algorithm for distributed coordination.

Key Characteristics

- **High Performance:** Sub-millisecond latencies with throughput exceeding 100K operations/second
- **Strong Consistency:** ACID transactions with linearizable consistency guarantees
- **Fault Tolerance:** Automatic failover with no data loss in minority failure scenarios
- **Horizontal Scalability:** Dynamic cluster membership with intelligent load balancing
- **Cloud Native:** Kubernetes-optimized with comprehensive observability

Table of Contents

1. Architecture Overview

System Architecture Diagram

```
┌──────────────────────────────────────────────────┐
│                 Client Applications               │
└────────────────┬────────────────┬────────────────┘
                 │                │
┌────────────────▼────────────────▼────────────────┐
│                   Load Balancer                    │
│              (gRPC/HTTP Multiplexing)              │
└────────────────┬────────────────┬────────────────┘
                 │                │
┌────────────────▼────────────────▼────────────────┐
│                  KVStore Cluster                   │
│ ┌──────────┐   ┌──────────┐   ┌──────────┐       │
│ │  Node 1  │   │  Node 2  │   │  Node 3  │       │
│ │ (Leader) │◄──│(Follower)│◄──│(Follower)│       │
│ │          │   │          │   │          │       │
│ │┌────────┐│   │┌────────┐│   │┌────────┐│       │
│ ││  API   ││   ││  API   ││   ││  API   ││       │
│ ││ Server ││   ││ Server ││   ││ Server ││       │
│ │└────────┘│   │└────────┘│   │└────────┘│       │
│ │┌────────┐│   │┌────────┐│   │┌────────┐│       │
│ ││  Raft  ││   ││  Raft  ││   ││  Raft  ││       │
│ ││ Engine ││   ││ Engine ││   ││ Engine ││       │
│ │└────────┘│   │└────────┘│   │└────────┘│       │
│ │┌────────┐│   │┌────────┐│   │┌────────┐│       │
│ ││BadgerDB││   ││BadgerDB││   ││BadgerDB││       │
│ ││ Storage││   ││ Storage││   ││ Storage││       │
│ │└────────┘│   │└────────┘│   │└────────┘│       │
│ └──────────┘   └──────────┘   └──────────┘       │
└──────────────────────────────────────────────────┘
```

Key Design Principles

1. **Separation of Concerns:** Clear separation between consensus, storage, and API layers
2. **Fault Isolation:** Component failures don't cascade across system boundaries
3. **Horizontal Scalability:** Linear performance scaling with cluster size
4. **Operational Simplicity:** Self-healing with minimal manual intervention
5. **Performance First:** Optimized for high-throughput, low-latency workloads

2. Core Components

1. API Gateway Layer

Responsibilities:

- Client request routing and load balancing
- Protocol translation (REST ↔ gRPC)
- Authentication and authorization
- Rate limiting and circuit breaking
- Request/response transformation

Key Features:

- Multi-protocol support (gRPC, HTTP/REST)
- Intelligent leader routing for write operations
- Read load distribution across healthy nodes
- Automatic retry with exponential backoff
- Comprehensive request/response logging

2. Consensus Engine (Raft)

Responsibilities:

- Distributed consensus and leader election
- Log replication across cluster members
- Membership changes and configuration updates
- Split-brain prevention and network partition handling

Implementation Details:

- Modified Raft algorithm with optimizations for key-value workloads
- Batched log entries for improved throughput
- Pre-vote mechanism to reduce leader election disruption
- Configuration checkpointing for fast cluster recovery

3. Storage Engine (BadgerDB)

Responsibilities:

- Persistent data storage with ACID guarantees
- Efficient key-value operations with prefix scanning
- Automatic compaction and garbage collection
- Backup and restore capabilities

Key Features:

- LSM-tree based storage for write-optimized performance
- Memory-mapped value log for read optimization
- Configurable compression (Snappy, Zstandard)
- Encryption at rest with AES-256

3. Data Model

Key Structure

```
type Key struct { Value string // UTF-8 string, max 1KB Namespace string // Optional namespace for isolation Metadata Metadata // System and user metadata } type Metadata struct { CreatedAt time.Time UpdatedAt time.Time ExpiresAt *time.Time // Optional TTL Version uint64 // Optimistic concurrency control Tags map[string]string }
```

Value Structure

```
type Value struct { Data []byte // Binary data, max 16MB ContentType string // MIME type hint Encoding string // Compression encoding Checksum string // Data integrity verification }
```

Transaction Model

The system supports ACID transactions with the following guarantees:

- **Atomicity:** All operations in a transaction succeed or fail together
- **Consistency:** Transactions maintain data integrity constraints
- **Isolation:** Concurrent transactions don't interfere with each other
- **Durability:** Committed transactions survive system failures

Example Transaction

```
transaction: operations: - type: PUT key: "user:123:profile" value: "{\"name\": \"John Doe\", \"email\": \"john@example.com\"}" conditions: - if_version_match: 5 - type: DELETE key: "user:123:temp_data" - type: PUT key: "user:123:last_login" value: "2024-01-15T10:30:00Z" ttl_seconds: 86400
```

4. Consensus Algorithm

Raft Implementation Details

Our Raft implementation includes several optimizations for key-value store workloads:

1. Log Structure Optimization

```
type LogEntry struct { Index uint64 Term uint64 Timestamp time.Time Type EntryType Operations []Operation Checksum string } type Operation struct { Type OpType // PUT, DELETE, BATCH Key string Value []byte Conditions []Condition Metadata map[string]string }
```

2. Batching Strategy

- **Write Batching:** Multiple client operations batched into single log entries
- **Read Batching:** Follower read requests batched for improved throughput
- **Configurable Batch Size:** Adaptive batching based on load patterns

3. Leader Election Optimization

- **Pre-vote Phase:** Prevents unnecessary leader elections
- **Priority-based Elections:** Preferred leaders based on node capabilities
- **Fast Recovery:** Optimized recovery from network partitions

4. Membership Changes

```
cluster_membership: current_config: - node_id: "node-1" address: "10.0.1.10:7000" role: "voter" - node_id: "node-2" address: "10.0.1.11:7000" role: "voter" - node_id: "node-3" address: "10.0.1.12:7000" role: "voter" pending_config: operation: "add_node" target: node_id: "node-4" address: "10.0.1.13:7000" role: "voter"
```

5. Storage Engine

BadgerDB Architecture

1. LSM-Tree Structure

```
Memory Components:
┌─────────────────┐
│   Memtable      │ ← Active writes
│   (Skip List)   │
└─────────────────┘
┌─────────────────┐
│  Immutable      │ ← Sealed memtable
│   Memtable      │
└─────────────────┘

Disk Components:
┌─────────────────┐
│    Level 0      │ ← Direct memtable flushes
│  (Unsorted)     │
└─────────────────┘
┌─────────────────┐
│    Level 1      │ ← Compacted, sorted
│   (Sorted)      │
└─────────────────┘
┌─────────────────┐
│    Level N      │ ← Higher levels
│   (Sorted)      │
└─────────────────┘

Value Log:
┌─────────────────┐
│   Value Log     │ ← Large values stored separately
│  (Append-only)  │
└─────────────────┘
```

2. Key Features

- **Write Amplification:** Minimized through level-based compaction
- **Bloom Filters:** Reduce read amplification for non-existent keys
- **Block Cache:** LRU cache for frequently accessed blocks
- **Value Threshold:** Small values stored in LSM, large values in value log

3. Performance Tuning

```
badger_config: # Memory settings memtable_size: 64MB block_cache_size: 256MB index_cache_size: 128MB # Compaction settings num_memtables: 5 num_level_zero_tables: 5 level_size_multiplier: 10 # Value log settings value_threshold: 1024 value_log_file_size: 1GB # Performance settings sync_writes: false num_compactors: 4 compression: snappy
```

7. API Design

1. gRPC API

The primary API uses Protocol Buffers for type safety and performance:

```
service KVStore { // Core operations rpc Put(PutRequest) returns (PutResponse); rpc Get(GetRequest) returns (GetResponse); rpc Delete(DeleteRequest) returns (DeleteResponse); // Batch operations rpc BatchPut(BatchPutRequest) returns (BatchPutResponse); rpc BatchGet(BatchGetRequest) returns (BatchGetResponse); // Advanced operations rpc List(ListRequest) returns (ListResponse); rpc Watch(WatchRequest) returns (stream WatchResponse); rpc Transaction(TransactionRequest) returns (TransactionResponse); // Administrative operations rpc Backup(BackupRequest) returns (BackupResponse); rpc Restore(RestoreRequest) returns (RestoreResponse); rpc Status(StatusRequest) returns (StatusResponse); }
```

2. REST API

HTTP API for broader compatibility:

```
endpoints: # Core operations PUT /api/v1/keys/{key} # Create/update key GET /api/v1/keys/{key} # Retrieve key DELETE /api/v1/keys/{key} # Delete key HEAD /api/v1/keys/{key} # Check existence # Batch operations POST /api/v1/batch/put # Batch create/update POST /api/v1/batch/get # Batch retrieve POST /api/v1/batch/delete # Batch delete # Advanced operations GET /api/v1/keys # List keys (with prefix) POST /api/v1/transaction # Execute transaction GET /api/v1/watch/{prefix} # Server-sent events # Administrative POST /api/v1/admin/backup # Create backup POST /api/v1/admin/restore # Restore from backup GET /api/v1/admin/status # Cluster status GET /api/v1/admin/metrics # Performance metrics
```

8. Performance Characteristics

1. Latency Metrics

Operation Type	P50	P95	P99	P99.9
Single PUT	0.5ms	2ms	5ms	10ms
Single GET	0.3ms	1ms	3ms	8ms
Batch PUT (10)	2ms	8ms	15ms	30ms
Batch GET (10)	1ms	4ms	10ms	20ms
Transaction (5 ops)	3ms	12ms	25ms	50ms

2. Throughput Capabilities

Cluster Size	Write QPS	Read QPS	Total QPS
3 nodes	50K	200K	250K
5 nodes	80K	400K	480K
7 nodes	100K	600K	700K

3. Scalability Characteristics

- **Write Scaling:** Limited by leader capacity, typically 50-100K writes/sec
- **Read Scaling:** Linear with follower count
- **Storage:** Supports PB-scale data with appropriate hardware
- **Memory:** Configurable cache sizes, typically 10-50% of working set

9. Security Model

1. Authentication & Authorization

TLS Configuration

```
tls: enabled: true cert_file: "/etc/certs/server.crt" key_file: "/etc/certs/server.key" ca_file: "/etc/certs/ca.crt" client_auth: "require_and_verify_client_cert" min_version: "1.3" cipher_suites: - "TLS_AES_256_GCM_SHA384" - "TLS_CHACHA20_POLY1305_SHA256"
```

RBAC System

```
roles: - name: "admin" permissions: - "kvstore:*" - "cluster:*" - "backup:*" - name: "developer" permissions: - "kvstore:read" - "kvstore:write" - "kvstore:list:app:*" - name: "readonly" permissions: - "kvstore:read" - "kvstore:list" users: - username: "admin" roles: ["admin"] auth_method: "certificate" - username: "app-service" roles: ["developer"] auth_method: "jwt" namespace_prefix: "app:"
```

2. Encryption

Encryption at Rest

- **Algorithm:** AES-256-GCM
- **Key Management:** External KMS or Vault integration
- **Key Rotation:** Automatic daily rotation
- **Scope:** All stored data including logs and snapshots

Encryption in Transit

- **Protocol:** TLS 1.3 minimum
- **Cipher Suites:** AEAD ciphers only
- **Certificate Management:** Automated with cert-manager
- **mTLS:** Enforced for inter-node communication

3. Audit & Compliance

```
audit: enabled: true events: - "authentication_failure" - "authorization_failure" - "admin_operations" - "data_access" destinations: - type: "file" path: "/var/log/kvstore/audit.log" format: "json" - type: "syslog" facility: "LOG_AUTH" severity: "LOG_INFO" - type: "webhook" url: "https://siem.company.com/api/events" headers: Authorization: "Bearer ${AUDIT_TOKEN}"
```

10. Operational Concerns

1. Deployment Architecture

Kubernetes Deployment

```
apiVersion: apps/v1 kind: StatefulSet metadata: name: kvstore spec: serviceName: kvstore replicas: 3 template: spec: containers: - name: kvstore image: kvstore:latest resources: requests: cpu: 2 memory: 8Gi storage: 100Gi limits: cpu: 4 memory: 16Gi volumeMounts: - name: data mountPath: /data - name: config mountPath: /etc/kvstore volumeClaimTemplates: - metadata: name: data spec: accessModes: ["ReadWriteOnce"] storageClassName: "ssd" resources: requests: storage: 100Gi
```

2. Monitoring & Observability

Metrics Collection

```
prometheus: scrape_configs: - job_name: 'kvstore' static_configs: - targets: ['kvstore-0:2112', 'kvstore-1:2112', 'kvstore-2:2112'] scrape_interval: 15s metrics_path: /metrics
```

Key Metrics

- **Performance:** Latency percentiles, throughput, error rates
- **Consensus:** Leader elections, log replication lag, commit latency
- **Storage:** Disk usage, compaction stats, cache hit rates
- **System:** CPU, memory, network, disk I/O

Alerting Rules

```
groups: - name: kvstore rules: - alert: KVStoreHighLatency expr: histogram_quantile(0.95, kvstore_request_duration_seconds) > 0.1 for: 5m labels: severity: warning annotations: summary: "KVStore high latency detected" - alert: KVStoreLeaderElection expr: increase(kvstore_leader_elections_total[5m]) > 0 for: 1m labels: severity: critical annotations: summary: "KVStore leader election occurred"
```

3. Backup & Recovery

Backup Strategy

```
backup: schedule: "0 2 * * *" # Daily at 2 AM retention: daily: 7 weekly: 4 monthly: 12 storage: type: "s3" bucket: "kvstore-backups" encryption: "AES256" compression: "zstd" verification: enabled: true sample_rate: 0.1 # Verify 10% of keys
```

Recovery Procedures

1. **Point-in-time Recovery:** Restore from snapshot + replay logs
2. **Cross-region Recovery:** Replicate backups across regions
3. **Disaster Recovery:** Automated failover to backup region

4. Capacity Planning

Growth Projections

```
capacity_planning: metrics: - name: "storage_growth" current: "500GB" growth_rate: "10GB/month" projection_months: 12 threshold_warning: "80%" threshold_critical: "90%" - name: "request_rate_growth" current: "10K QPS" growth_rate: "15%/quarter" projection_quarters: 4 threshold_warning: "70% of capacity" threshold_critical: "85% of capacity"
```

Resource Requirements

Minimum Configuration (Development)

```
resources: cpu: 0.5 cores memory: 1GB storage: 10GB SSD network: 100 Mbps
```

Production Configuration

```
resources: cpu: 4-8 cores memory: 16-32GB storage: 1TB NVMe SSD network: 10 Gbps
```

11. Future Roadmap

Phase 1: Enhanced Features (Q1 2024)

- **Secondary Indexes:** Support for custom indexing strategies
- **Streaming API:** Real-time change streams
- **Multi-tenancy:** Namespace isolation with resource quotas
- **Compression:** Advanced compression algorithms (LZ4, Zstandard)

Phase 2: Advanced Capabilities (Q2 2024)

- **Cross-region Replication:** Active-active multi-region deployments
- **Schema Evolution:** Structured data with schema validation
- **Time-series Support:** Optimized storage for time-series data
- **Graph Queries:** Basic graph traversal capabilities

Phase 3: Enterprise Features (Q3 2024)

- **LDAP Integration:** Enterprise directory service integration
- **Advanced Analytics:** Built-in analytics and reporting
- **Automated Tuning:** ML-based performance optimization
- **Edge Computing:** Lightweight edge node deployment

Phase 4: Ecosystem Integration (Q4 2024)

- **Kafka Integration:** Change data capture to Kafka
- **Spark Connector:** Direct integration with Apache Spark
- **Kubernetes Operator:** Advanced cluster lifecycle management
- **Service Mesh:** Native Istio/Envoy integration

12. Conclusion

The Distributed Key-Value Store represents a modern approach to distributed data storage, combining proven algorithms (Raft consensus) with high-performance storage engines (BadgerDB) to deliver a robust, scalable solution for contemporary applications.

Key Strengths

1. **Performance:** Sub-millisecond latencies with high throughput
2. **Reliability:** Strong consistency with automatic failover
3. **Scalability:** Linear read scaling with efficient write handling
4. **Operability:** Comprehensive monitoring and self-healing capabilities
5. **Security:** Defense-in-depth with encryption and RBAC

Design Trade-offs

1. **Consistency vs. Availability:** Strong consistency may impact availability during network partitions
2. **Write Scalability:** Single leader limits write throughput
3. **Memory Usage:** LSM trees require significant memory for optimal performance
4. **Complexity:** Distributed consensus adds operational complexity

This system design provides a solid foundation for building reliable, high-performance distributed applications while maintaining operational simplicity and strong consistency guarantees.