Distributed Game Server based on RAFT Consensus Core

Jialuo Hu
jih146@ucsd.edu
University of California, San Diego

Abstraction: This report introduces a Go-based framework for building distributed multiplayer game servers that delivers strong consistency and fault tolerance without burdening developers with low-level details. At its core lies a Raft consensus module that guarantees a single, totally ordered history of game events, while a simple state-machine interface lets game logic consume those events in order. By combining consensus, durable storage, and pluggable game-server components under a unified API, the framework enables reliable, low-latency gameplay and seamless recovery from failures. Future work will focus on snapshot-based fast recovery, encrypted transport, real-cluster validation, extensible state-storage support, and performance tuning under heavy concurrency.

1 Introduction

In recent years, multiplayer online gaming has evolved from turn-based encounters to fast-paced, large-scale worlds where thousands of players interact in real time. Delivering such experiences is based on two critical requirements: actions must be reflected in the game world with minimal delay, and all servers must maintain a unified view of game state, even when individual machines crash or networks fail (3). Traditional client-server designs often create a single point of failure or performance bottleneck, while purely peer-to-peer approaches struggle to enforce a global ordering of events and defend cheating under volatile network conditions.

Consensus algorithms provide a robust foundation to meet these challenges. In particular, the Raft protocol offers a clear, leader-based approach to replicating a log of state-changing commands across a cluster of servers. With Raft, servers automatically elect a leader, replicate incoming commands to followers, and gracefully handle membership changes, all while guaranteeing that once a command is committed it will never be lost or reordered.

In this project, we designed and built a high-performance game server framework in Go that embeds a standalone Raft consensus core alongside a modular game state machine on each physical server. Client requests enter a lightweight gateway in the game server layer, pass through a Raft gateway for agreement, and only then are applied to game logic once a majority of nodes confirm durable storage. For persistence, we integrated Cockroach Labs' Pebble key-value store to achieve low-latency write-ahead logging and efficient compaction, ensuring low commit latencies under heavy load. By separating consensus mechanics from gameplay code, our framework presents a simple, callback-driven API to developers, who can focus on crafting game features such as movement, combat, or team mechanics, and without worrying about failure handling or consistency.

This report begins by reviewing the Raft consensus algorithm, its safety guarantees, and its membership change procedures. We then describe our overall system architecture, detailing how the game

https://github.com/jialuohu/curlyraft

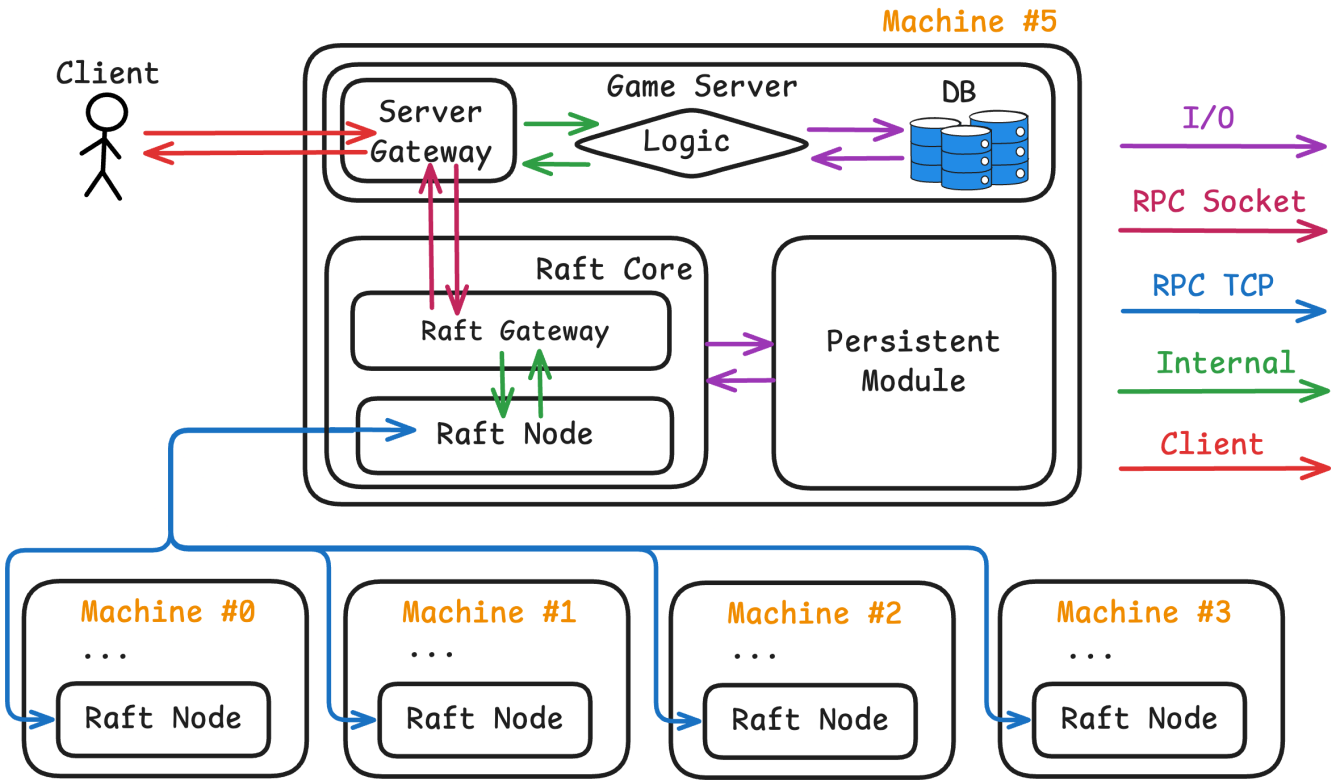

Figure 1: Architecture of the Raft-based game framework.

server and Raft core interact on each node. Next, we dive into implementation specifics, covering cluster initialization, storage integration, RPC handling, and heartbeat-driven election loops. Then we will have an experimental evaluation. Finally, we discuss planned future improvements such as snapshots, secure transport, elastic scaling, and summarize our project.

2 System Architecture

The distributed game server cluster is composed of multiple physical machines, each hosting two distinct components: the game server and the Raft consensus core. The Figure 1 shows the whole architecture. When a player issues an action, the client library opens a connection to a front-end server gateway service that exposes the public API. This "Server Gateway" is responsible for managing client connections, routing incoming messages to the appropriate back-end service, and performing any necessary authentication or rate limiting. As soon as the gateway accepts a valid game request, it hands that request off to the Raft consensus core, ensuring that every state-changing operation first obtains agreement from a majority of servers before it is applied.

Within the consensus core, the entry point is a "Raft Gateway" module that marshals the request into the log record format defined by the Raft protocol. Then the module invokes the local "Raft Node" and delivers the new record for replication. Each node in the cluster implements the Raft protocol's leader election procedure and log replication mechanisms. The node chosen as leader takes responsibility for appending the new record to its own log, then dispatches RPC messages to peer nodes on other machines in order to have them append identical log entries. The system waits until a majority of peers acknowledge durable storage of the record before considering the entry to be committed. If the

leader fails or becomes unreachable, the remaining nodes automatically begin a new election so that progress can continue without manual intervention.

Once the log record reaches committed status, the "Raft Node" notifies the game server, which actually acting as the "Replicatedd State Machine" under context of Raft terminology, to process the original client request. It interprets the request, executes the corresponding game logic to update player attributes, world state or other domain objects in memory, and then store those changes to its own database. This database is optimized for the high throughput and low latency demands of multiplayer gameplay. Separately, the Raft core also stores its internal state data such as log entries, current term number, and voting history into the "Persistent Module" - a dedicated durable store that ensures rapid recovery after crashes or planned restarts.

By separating raft consensus core from game logic, the architecture guarantees that all state transitions occur in a total order agreed upon by a majority of machines, even in the presence of network partitions or server failures. Meanwhile the system stays highly responsive because the server that commits a command also runs the game logic and handles its own database operations. The whole system will ensure that players experience consistent, reliable interactions with the game world, while benefiting from the fault tolerance and strong consistency by the Raft protocol.

3 Raft Consensus Algorithm

The consensus in distributed systems revolves around ensuring that multiple servers agree on a sequence of state-changing commands despite failures. Traditional approaches like Paxos achieve this but are difficult to understand and implement correctly. Raft was developed for clarity. It centers on a single leader, cleanly separates the steps of choosing a leader, copying log entries, and enforcing safety, and keeps the possible states of each server to a minimum. By doing so, it matches Paxos in both fault tolerance and performance while remaining much easier for developers and students to understand and implement.

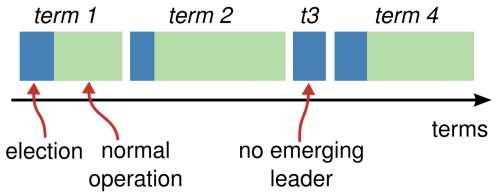

Figure 2: Raft term reproduced from Raft paper (2).

In Raft, time is divided into numbered terms acting like a logical clock, ensuring every event has a clear place in the sequence of updates, as the Figure 2 shown. Each term begins with an election, and at any moment a server can be a leader, a follower, or a candidate. The followers wait quietly for instructions. If a follower goes too long without hearing from a leader, after a brief random timeout, it becomes a candidate for the next term. It increments its term number, votes for itself, and asks its peers to vote for it as well. A candidate becomes the leader only when it wins votes from a majority of servers in the same term. If it cannot secure enough votes, it either recognizes another server as leader by receiving the leader's heartbeat message or waits for another randomly timed interval before trying again. To prevent multiple servers from timing out and starting elections at the same moment, each server uses a slightly different, randomly chosen timeout before beginning a vote. This randomness dramatically reduces the chance of split votes and helps the cluster settle on a leader quickly and reliably. By varying these timeouts, Raft keeps elections from colliding and elects a new leader fast; it's rare for two servers to campaign at once, so leadership contests usually wrap up quickly. After winning an election, the new leader begins sending regular heartbeat messages to all the other servers, both to state its authority and keep everyone's logs in sync. If that leader sees a heartbeat indicating a higher term number from some other server, it immediately relinquishes control and falls back to the follower state. The Figure 3 shows the whole

election process.

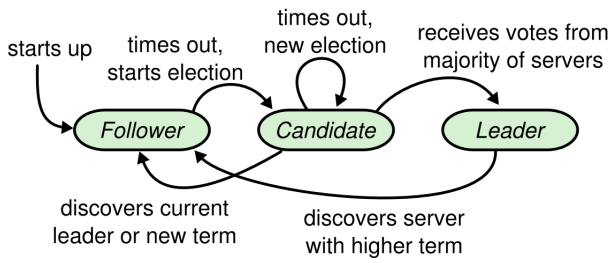

Figure 3: Server state figure from Raft paper (2).

When a client sends a command to the cluster, the leader appends it to its own log and immediately begins replicating it to the other servers by including new entries in its regular `AppendEntries RPC` requests. Each of these requests carries two important information: the term number and the index of the entry that comes immediately before the new ones. If a follower's log does not match at that point, it rejects the request. The leader then reduces that follower's "next" index by one and tries again, stepping backward through the log until it finds the last entry the follower has in common. As soon as the follower accepts a match, the leader sends it all missing entries in one batch; we used the batching optimization to reduce the number of RPCs we need to send.

Behind the scenes, the leader maintains two arrays for each follower: one called `nextIndex`, which marks the position of the next log entry to send, and another called `matchIndex`, which records the highest log index the follower has acknowledged. With these two arrays, the leader can pipeline multiple entries in flight and quickly bring any slow or recovering server back up to date. Once a particular entry has been stored on a majority of servers by checking `matchIndex`, the leader considers it committed. Only entries from the current term are committed by this direct count; earlier entries become committed indirectly because Raft enforces a strict log-matching property that if two logs share an entry at the same index and term, they must be identical up through that point. It then applies that entry to its state machine and includes the updated commit position in all subsequent `AppendEntries RPC` calls, ensuring every follower learns which commands are safe to execute. This process of consistency checks and batched replication makes Raft's log synchronization efficient and robust, without discarding data that has already been agreed upon.

Raft's safety guarantee means that once a command is committed by a majority of servers, it becomes part of the cluster history and can never be reversed or contradicted. The guarantee is based on that no leader can be elected unless it has all committed entries. Raft achieves this by refusing to grant votes to candidates whose logs are not at least as up-to-date as the voter's. "Up-to-date" is defined by comparing the term and index of the last log entries; higher term wins, and if terms tie the longer log wins. This restriction guarantees that any leader who could override committed entries cannot win an election. If a leader commits an entry in term T, then every subsequent leader must also contain that entry, and thus no server ever applies conflicting commands to its state machine.

As the Raft log grows over time, recording every client command can consume significant storage and slow down recovery. To address this, Raft takes occasional snapshots of the entire state machine at the point of the last applied entry. When the log size passes a preset threshold, the leader asks the state machine to produce a snapshot and then discards all log entries up through that index, keeping only the snapshot's metadata which has a record of the last included index and term. This compaction step ensures that the system never needs to retain an unbounded history of commands.

If a follower has fallen so far behind that it no longer holds the log entries needed to catch up through normal replication, the leader sends the snapshot instead. The follower restores its state machine from the snapshot data and resets its log to begin just after that snapshot point. From there it resumes accepting new entries in order. Because the snapshot reflects every committed command up to its index, this process preserves Raft's consistency guarantees without losing the updates. By period-

ically preserving its log in this way, Raft keeps both storage use and recovery times under control, making use of the storage more efficient for long-running systems.

Clients interact with Raft by sending their requests to the leader. To provide linearizable semantics, clients tag each command with a unique serial number; if the leader crashes after committing but before responding, the new leader detects duplicates and responds without re-executing the command. That entry proves the leader's log is fully up to date. Before returning data to a client, the leader then exchanges heartbeat messages with a majority of servers. By following this simple sequence, Raft guarantees fresh reads without the overhead of writing every query into the log.

```go
func example() {
  // Open (or create) the database
  db, err := pebble.Open("/tmp/example.db")
  checkError(err)
  defer db.Close()

  // Simple Set
  err = db.Set([]byte("user:100"), []byte("Alice")
      , pebble.Sync)
  checkError(err)

  // Simple Get
  value, closer, err := db.Get([]byte("user:100"))
  checkError(err)
  closer.Close()

  // Simple Delete
  err = db.Delete([]byte("user:100"), pebble.Sync)
  checkError(err)
}
```

Listing 1: Basic Pebble API Usage

4 Implementations

4.1 Pebble: The Key-Value Store

We chose Pebble for our Raft persistence layer because it integrates natively with our Go implementation of the Raft protocol. Pebble is written entirely in Go, which eliminates the performance penalty and complexity of crossing between Go and C++ code, and can develop and maintain features more quickly. Since Pebble includes only the key-value operations that Raft actually uses, its code stays compact and there are very few settings to worry about. The pebble API is simple and straightforward as shown in the code snippet 1.

Under the hood, Pebble employs a log-structured merge design that buffers writes in memory tables backed by a concurrent skip list and then flushes them into immutable on-disk tables. Its custom compaction strategies keep write amplification low and its cache management scales smoothly under concurrent Raft append workloads. Built-in support for efficient range deletions makes snapshotting and log truncation fast, and durable write-ahead logs together with sorted string tables guarantee rapid recovery after a crash (1). These characteristics combine to deliver a low-latency, highly reliable store that meets or exceeds the performance of traditional engines, making Pebble an ideal choice for our Raft consensus core implementation.

4.2 Raft Core

Raft is a leader-based consensus algorithm designed for understandability and correctness in distributed systems. As mentioned before, it maintains a replicated log across a cluster of N servers where N should be an odd number to ensure that a clear majority can always be formed and to minimize the chance of split votes, ensuring that any log entry committed by the leader is eventually applied in the same order on every node. During normal operation, one leader handles all client requests by appending new entries to its local log and replicating them to followers using `AppendEntries RPCs`. A majority quorum $\lfloor N/2 \rfloor + 1$ must acknowledge (store an entry in their own log) before the leader marks an entry as committed. Followers reset their election timeout whenever they receive a heartbeat from the leader, preventing unnecessary elections. We divide the protocol into three parts to implement: leader election, log replication, snapshot. But the snapshot feature has not yet been implemented in our current code solution.

When the system starts, each node spins up a

```go
type RaftCore struct {
  Info       nodeInfo
  Peers      []nodeInfo
  peerConns map[string]*clientConnInfo

  node        *node
  grpcServer *grpc.Server
  rg          *RaftGateway

  mu             sync.Mutex
  receivedVotes uint16
  quorumSize     uint16
}
```

Listing 2: RaftCore struct

```go
type node struct {
  // node info
  state       Role
  commitIndex uint32
  lastApplied uint32
  timer       *time.Timer
  heartbeatCh chan struct{}
  stopCh      chan struct{}
  leaderId    *nodeInfo
  sm          curlyraft.StateMachine

  // persistence
  lastLogIndex uint32
  lastLogTerm  uint32
  storage      *persistence.Storage

  // leader only
  nextIndex           map[string]uint32
  matchIndex          map[string]uint32
}
```

Listing 3: Node struct

`RaftCore` instance, which parses the cluster configuration, initializes local storage, and opens `gRPC` connections to every peer. Nodes begin as followers, waiting for regular heartbeats from whoever is the current leader. Each server runs a `roleLoop` that tracks a randomized election timeout between 150 and 300 milliseconds. If that timer expires without hearing from the leader, the server will change its state as `Candidate`, increments its term counter, votes for itself, and sends `RequestVote RPCs` to its peers. As soon as it wins votes from a majority of nodes, it promotes itself to leader. The leader then initializes per-server `map[string]uint32` type of `nextIndex` and `matchIndex` to manage log replication and starts two background goroutines: one to send out periodic heartbeats and another to listen for client commands.

The core of our framework is embodied in the `RaftCore` struct, implemented in `core.go`. Code snippets 2 and 3 show the definition of `RaftCore` and `Node`. The `NewRaftCore` function bootstraps the server by parsing the cluster configuration string into individual peer node information such as its id and network address, initializing a local node instance for storing state and log entries, and establishing `gRPC` client connections to each peer address for Raft RPCs.

We have defined the gRPC communication protocol `raftcomm` for Raft nodes talking to each other as Code snippet 4 and 5. All Raft RPCs and a simple health-check. During leader election a candidate calls `RequestVote RPCs`, sending its term, ID, and last-entry info; peers reply with their term and a yes/no vote. Once a leader is in place it uses `AppendEntries RPCs` to push new log entries or heartbeats, including term, previous entry identifiers, the batch of entries, and the leader's commit index; followers respond with success or failure and their current term. If a follower lags too far behind, the leader streams a snapshot via `InstallSnapshot RPCs` carrying term, last-included index and term, a byte offset, and a chunk of snapshot data, with a flag for the final chunk. A minimal HealthCheck service lets external tools poll each node's status (`SERVING`, `NOTSERVING`, `UNKNOWN`).

We utilized Go's native concurrency features to keep the consensus logic both efficient and easy to understand. To monitor whether the leader is still alive, we set up a dedicated heartbeat channel that works like a continuous pulse. Whenever our `AppendEntries RPC` handler processes a valid heartbeat, it sends a simple signal into that channel without ever blocking the handler. If the channel is momentarily full, the extra signal is quietly dropped. Meanwhile, our main election loop watches both the heartbeat channel and a randomized timeout. As long as heartbeats keep arriving before the timeout expires, the loop resets and waits again, which lets us maintain leadership without complex locking or

```
package raftcomm;

service RaftCommunication {
  rpc AppendEntries(AppendEntriesRequest) returns
      (AppendEntriesResponse);

  rpc RequestVote(RequestVoteRequest) returns (
      RequestVoteResponse);

  rpc InstallSnapshot(InstallSnapshotRequest)
      returns (InstallSnapshotResponse);
}

service HealthCheck {
  rpc Check(HealthCheckRequest) returns (
      HealthCheckResponse);
}
```

Listing 4: gRPC service definitions for Raft

```
package gateway;

service RaftGateway {
  rpc AppendCommand(AppendCommandRequest) returns
      (AppendCommandResponse);
}

service HealthCheck {
  rpc Check(HealthCheckRequest) returns (
      HealthCheckResponse);
}
```

Listing 5: gRPC service definitions for RaftGateway

extra state checks.

We also used Go's `context` package to manage all leader-only work in clean, cancelable goroutines. When a node wins an election, we immediately spin up one goroutine to drive periodic `AppendEntries RPC` calls and another to serve client commands. Both routines share a single cancellation context that is triggered as soon as the node steps down. This guarantees that any in-flight RPCs or background loops shut down promptly and cleanly, preventing stray goroutines and avoiding race conditions. By combining non-blocking channels for fast in-memory signaling with contexts for precise lifecycle control, our code remains concise, correct, and straightforward to reason about. Concurrency is managed with Go channels and mutexes: election timers and heartbeat loops each run in their own goroutines, shared state is locked during updates. Contexts allow in-flight RPC loops to exit cleanly upon new elections or leadership changes, ensuring that stale leader routines do not become zombie routine after a stepdown.

Log replication is also driven by `AppendEntries RPCs`. Whenever a client submits a command, the leader appends it to its own log and immediately begins streaming it to all followers. Each RPC includes the term and index of the entry preceding the new ones, so that a follower can detect any inconsistency. If a follower rejects the request, the leader moves its `nextIndex` pointer decreases by one and retries until it finds the matching position. Once a majority of followers acknowledge storing the entry, the leader advances its commit index, applies the entry to its state machine, and includes the updated commit position in subsequent RPCs.

Durability of term, vote, and log state is provided by a Pebble-based Storage abstraction in storage.go. All critical metadata keys (`CurrentTerm` and `VotedFor`) are stored with synchronous fsync semantics, and log entries are appended under composite keys prefixed by `log/`plus a big-endian index. This design guarantees that any acknowledged Raft RPC persists to disk before returning success. Iterators scan the `log/` key range for operations

such as log retrieval, deletion of conflicting entries, and snapshot installation.

To expose the consensus service to the game server, each leader also launches a lightweight Raft gateway over a high-speed **UNIX-domain socket** compared with **TCP based RPC** since game server and Raft core are running on the same physical machine. Client commands arrive as simple RPCs, are translated incoming commands into byte slices and calling `RaftCore.propose()` to pipe through the command into Raft core processing logic. The newly committed entries are applied to the state machine via the `applyEntries()` loop, which reads sequential indices and invokes the provided `StateMachine.Apply()` callback.

4.3 Game Server: State Machine

The Raft core delivers every committed command back to be processed by the game logic with calling the `Apply(command []byte)` method on `interface StateMachine` implementation as code snippet 6. When a command has safely reached consensus and has been stored persistently, the Raft node will invoke `Apply(command []byte)` exactly once in log order. `Apply` is responsible for deserializing the command, executing the game's rules (for example updating player position or health), and returning any result payload that should be sent back to the client. By hiding Raft logic such as leader election, log replication, snapshot and recovery entirely inside the Raft core, the game server code simply works with a straightforward and totally ordered states change. To support fast recovery and mid-game joins without replaying the entire log, the `interface StateMachine` exposes two additional methods: `Snapshot()` and `Restore(snapshot []byte)`. Snapshot returns a byte slice encoding a complete dump of all in-memory game state. When a node restarts or a new client needs to catch up, `Restore(snapshot)` takes that same byte slice and recover the game world to exactly the same state it was when the snapshot was taken. This combination of `Apply`, `Snapshot` and `Restore` keeps the replication layer simple, while Raft core with Pebble for durable storage ensures strong consistency, fault tolerance even though we haven't implemented snapshot and recovery features on Raft core.

```
type StateMachine interface {
  Apply(command []byte) (result []byte, err error)
  Snapshot() (snapshot []byte, err error)
  Restore(snapshot []byte) error
}
```

Listing 6: Raft StateMachine interface

5 Evaluations

Because we lacked access to a dedicated server cluster and were constrained by time, we carried out all of our performance tests on a single MacBook Pro (Apple M1 Pro, 16 GB RAM, macOS 14.5). We simulated a five-node Raft cluster by running each node as its own Go process with `GOMAXPROCS=2`. Clients connected to the leader via a UNIX-domain socket, ensuring minimal network overhead.

We built a test that spawns a given number of goroutines which acting as a "virtual client." Every goroutine issues 100 `AppendCommand` calls and measures the time from when the call is made until the commit confirmation arrives. This setup will test the consensus path (client -> gRPC -> Raft log replication -> apply). The result is shown in the Table 1.

Because we ran all five Raft nodes on a single laptop with limited CPU and memory resources, our benchmark shows that with fifty concurrent clients we achieved a throughput of approximately 3 400 commands per second, a median latency of 50.2 ms, and a 95th percentile latency of 58.7 ms. When we increased to one hundred clients, throughput was around 3 100 commands per second, median latency was 51.5 ms, and the 95th percentile latency reached 62.3 ms. With two hundred clients, throughput fell to 2 800 commands per second while median latency rose to 56.8 ms and the 95th percentile to 70.1 ms. Finally, at three hundred clients the cluster sustained about 2 200 commands per sec-

Concurrency	Total Commands	Throughput (cmd/s)	Median Latency (ms)	95%-ile Latency (ms)
50	5 000	3 400	50.2	58.7
100	10 000	3 100	51.5	62.3
200	20 000	2 800	56.8	70.1
300	30 000	2 200	68.5	75.6

Table 1: Evaluation results on a five-node Raft cluster under varying client concurrency.

ond with a median latency of 68.5 ms and a 95th percentile latency of 75.6 ms. These results reflect the contention inherent in sharing limited hardware. In a production environment with dedicated servers, more CPU cores, and optimizations such as batching and snapshotting, we would expect significantly higher throughput and lower tail latencies.

6 Future Works And Summary

There are several key areas we plan to address in the future. First, we will add snapshot and recovery support to the Raft core so that nodes can catch up from a recent snapshot instead of replaying a growing log, reducing restart time and storage costs. Second, we will secure all communication channels, between Raft peers and between clients and servers, by adopting `gRPC` over TLS, ensuring confidentiality and integrity in production environments. Third, we intend to deploy the framework on a real server cluster to evaluate end-to-end performance under realistic network conditions, hardware variability and large client populations. Fourth, we will extend the game server layer to include supports for application-level database storage and network communication modules, enabling third-party developers to build rich multiplayer experiences on top of our code. Finally, we will profile and optimize the Raft implementation itself to improve concurrency handling under heavy client request loads, reducing contention and further raising throughput. These enhancements will move the framework toward a production-ready platform for building reliable, low-latency multiplayer games.

This report presented a high-performance, fault-tolerant multiplayer game server framework implemented in Go, combining a standalone Raft consensus core with a game server state machine. We detailed the Raft protocol integration, persistent log storage with Pebble, leader election, log replication, and the design of gateway components for client commands. Through local-machine experiments on a five-node cluster, the system demonstrated low commit latencies and throughput exceeding six thousand commands per second, while seamlessly handling simulated failures and restarts. The modular architecture with clear separation between consensus logic, game logic, and storage layers, offers developers a clear API for building the multiplayer game server. Future enhancements such as snapshot and recovery for Raft Core, and secure communication will further improve the framework's robustness and scalability for production environments.

References

[1] MATTIS, P. Introducing Pebble: A RocksDB-inspired key-value store written in Go, Sept. 2020.

[2] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[3] POZZAN, G., AND VARDANEGA, T. Rafting multiplayer video games. *Software: Practice and Experience 52*, 4 (2022), 1065–1091.