

Rafting multiplayer video games

Gabriele Pozzan¹ | Tullio Vardanega²

Department of Mathematics, University of Padua, Padua, Italy

Correspondence

Gabriele Pozzan, Department of Mathematics, University of Padua, Via Trieste, 63, 35121 Padua, Italy.
Email: gabriele.pozzan@studenti.unipd.it

Abstract

Consensus is a central concern for distributed systems, paramount for fault-tolerant applications. Online multiplayer (video) games are an attractive instance of highly distributed application, where user experience requires resilience provisioning that includes distributed consensus. In this work, we report on experiments we performed on the use of the Raft consensus algorithm in two Proof-of-Concept instances of famous video games. Our experiments aim to show the feasibility of such a novel architectural approach, and to assess the ensuing scalability quantitatively against game-specific performance metrics. To enable the transferability of this effort, we discuss our implementation choices and testing method, as well as the findings from said empirical evaluation.

KEYWORDS

distributed consensus, fault tolerance, Raft, reference implementation, scalability evaluation

1 | INTRODUCTION

Computer systems are intrinsically complex: a single computer operates resting on the interaction of multiple hardware and software components, each of which can fail for a variety of reasons (power failures, human errors, etc.). Such complexity vastly amplifies for distributed systems, which require multiple computer nodes to interact remotely.

In spite of their considerable complexity, distributed systems have become fundamental to many current application domains, such as industrial control, infrastructure management, and internet banking, just to name a few. All of them are subject to requirements of availability, integrity and robustness in the face of failures. In other words, they have to be *fault-tolerant*.

A system is said to be *fault-tolerant* if it is able to react gracefully and in a planned manner to any fault that may occur by either entering a well-defined alternative behavior or resiliently continuing operation in the face of the fault.¹

Redundancy is key to achieving the degree of robustness needed to be able to mask faults, wholly or partially, to users. One common way to attain redundancy is by running replicated state machines on multiple nodes of the system.²

The Raft consensus algorithm³ allows the consistent replication of a log collection on a cluster of nodes, each of which running a replicated state machine. Raft was created to offer an easier-to-understand equivalent to the most famous

Abbreviations: FPS, first-person shooter; PoC, Proof-of-Concept; RTS, real-time strategy

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2021 The Authors. *Software: Practice and Experience* published by John Wiley & Sons Ltd.



Paxos algorithm.⁴ For this very reason, Raft's authors put great care in breaking the consensus problem down to smaller separate and easier-to-manage sub-problems (leader election, log replication, configuration changes, log compaction), also choosing realization mechanics based on simplicity and understandability (e.g., reliance on strong leaders, and indeed the whole leader election mechanism).

At the origin of this article was our wish to investigate whether the use of the Raft consensus algorithm might fit the domain of online multiplayer video games. While being less critical than others, said field is experiencing a steep rise in stakes due to their commercial thrust and the increased prominence of e-sports. On a more social tone, online games are being looked at by pedagogists as useful learning aids.⁵

Positive findings from our investigation would yield worthwhile benefits:

- Players could host Raft nodes themselves, giving rise to a wholly decentralized architecture, thereby dispensing with the need for a centralized server;
- A centralized architecture could also leverage having the game state distributed over a cluster of servers, as their networked aggregation—given sufficient nodes—would become robust to server failures.

The contributions of this work are as follows:

- A fresh open-source implementation of Raft using the Go programming language, which can be used as a learning base or a building block for multiple other uses, professional, educational, inspirational.
- The performance testing of two Proof-of-Concept (PoC) game implementations with different requirements (real-time vs. fully turn-based) against two alternative Raft network architectures.
- The open-source implementation of the Raft extension described in Reference 6, which addresses Byzantine behavior.
- A simple Node.js based tool called *raft_analyzer*, which can be used to aggregate and analyse Raft log traces to verify their adherence to Raft's safety properties (election safety, leader append-only, log matching, leader completeness, state machine safety). This tool also provides a web interface for the graphical display of all the coordination communication flowing within the cluster.
- Empirical evidence that hints quantifiably at the performance penalty of using replication to achieve robustness. Our results show how our two architectures scale, by measuring simple metrics such as the delay between user actions and game feedback.
- Our code is available in open source in the public domain, at <https://github.com/cornacchia/go-raft-multiplayer-poc>

To the best of our knowledge, this work is the first published attempt at applying the Raft algorithm to video games. In keeping with that specificity, we use game-specific metrics (action delays, number of actions per second) to study the scalability of alternative network architectures as the number of nodes increase.

The remainder of this article is organized as follows: Section 2 describes the general architecture of our implementation; Section 3 recalls the essence of the Raft algorithm, and describes our main implementation choices; Section 4 does the same for the Byzantine-tolerant extension of it; Section 5 presents our empirical evaluation method, and illustrates the results we obtained from it; Section 6 discusses related work; Section 7 draws conclusions from this work, and outlines the lessons that can be learned from it.

2 | ARCHITECTURAL DECISIONS

2.1 | About the games

In this section we describe the (simplified) multiplayer online games we implemented in this project and the two alternative Raft architectures on which such games were experimentally run. Subsequently, we discuss the applicability of Raft-like distributed consensus to distributed online games in general.

Our project did not aim at producing a commercially viable product or indeed full-fledged ready-for-use games. Conversely, we limited our implementation to a fraction of the overall game logic, sufficient to allow simulating the essential behavior of a full game client. We then added a touch of graphics on top of that, to make it less dull to observation:



- **go_skeletons**: this game, whose client interface is captured in Figure 1, is inspired to the classic Wolfenstein 3D first-person shooter (FPS) game of the early 90s, as emblematic of real-time games with strong emphasis on responsiveness and player feedback.
- **go_wanderer**: in this game, a screenshot of which is displayed in Figure 2, players can move around on a tile-based screen. Movement progresses in turns, and each player must make their move for a turn to pass.

In both games, each action is associated to a Raft message. Any such action (which also manifests in the client's interface as is typical of video games) affects the game state only after the corresponding message has been committed, which in fact means replicated, to all nodes in the game network.



FIGURE 1 A screenshot from **go_skeletons**



FIGURE 2 A (scarcely exciting) screenshot from **go_wanderer**



We reckon that these two simple examples stand on the opposite extremes of a spectrum that has fully real-time games (FPSs, real-time strategy (RTS), etc.) on one end, and fully turn-based games on the other (many board games, some strategic games, etc.). Most actual games stand in the middle of this spectrum, for example by allowing turns to expire without some players' input.

In both games, the only player input that we implemented in this experimental project was movement. Arguably though, this limitation does not impair our research objective, as it is completely transparent to the Raft algorithm, in that more variety of user input would merely correspond to more Raft traffic, whose increase we already explore in our experiments without running the risk of clogging the network by requiring the transfer of very large chunks of data. Such transfer is a challenge for any networked application, hence not particularly revealing for this work.

2.2 | About the Raft architectures

We tested those games on two different Raft architectures. In both of them, a Raft node is associated with a game engine, which operates as a replicated state machine. Clients run a user interface (UI) module that receives keyboard input and forwards it to the Raft network in a JSON encoded form. Updates to the UI differ according to the Raft architecture in use.

2.2.1 | The 1:1 architecture

This architecture model (implemented in branch *main* of the repository cited in Section 1) is depicted in Figure 3. In it, every client hosts a Raft node, whereby the game state is replicated across every individual participant. This arrangement allows UI modules to directly query their local game engines to receive game state updates.

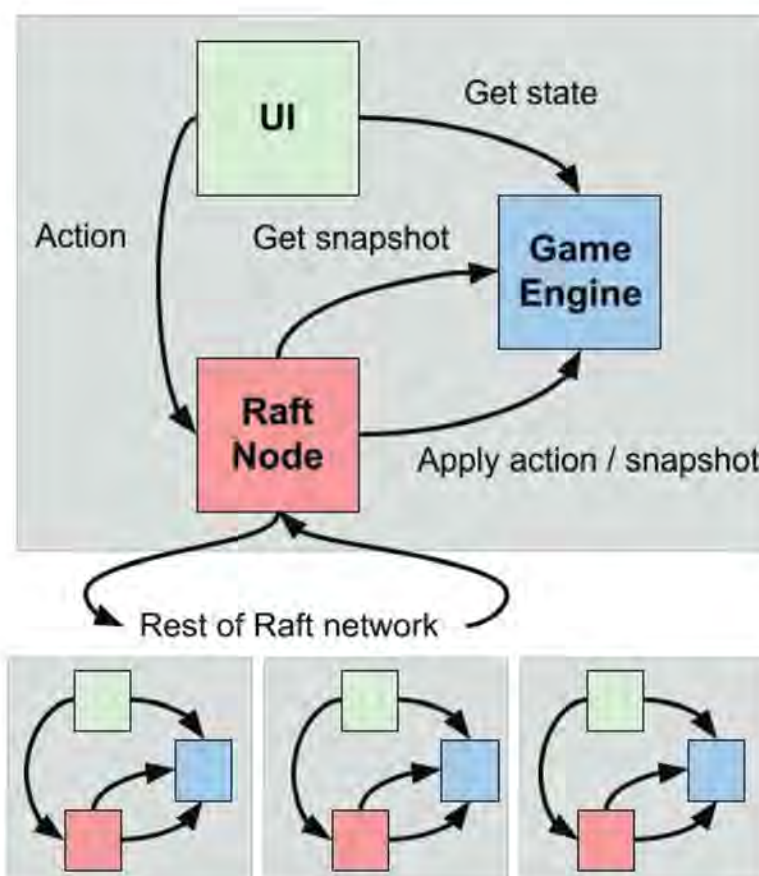


FIGURE 3 1:1 architecture: each system node has a replicated Raft node

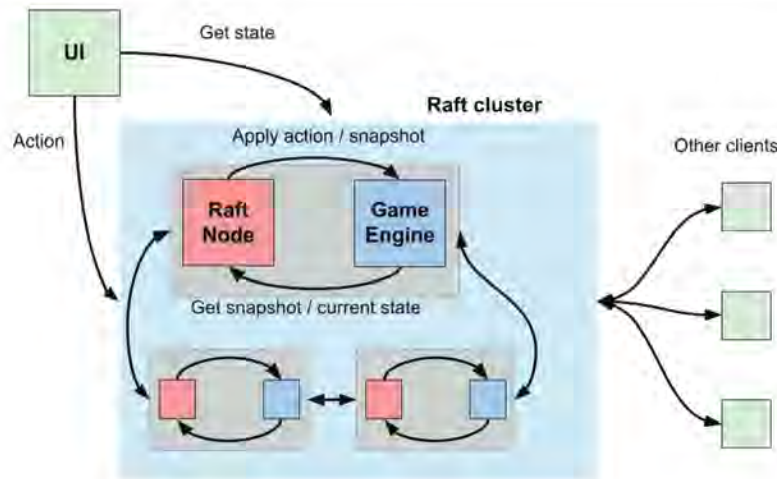


FIGURE 4 Cluster architecture: the Raft nodes are clustered, outside of client nodes

One interesting question that arises from this architecture is: how many nodes can a Raft network include while preserving adequate performance? We evaluated this aspect based on average response delays and actions per second. We discuss the outcomes of this strand of evaluation in Section 5.

2.2.2 | The cluster architecture

In this architecture model (implemented in branch *server-cluster* of the repository and depicted in Figure 4), clients and nodes are separated: Raft nodes update the game engines and clients receive game state updates as responses to the actions that they perform. (A special *nop* action is used to actively poll for updates when the client is idle.)

In order to maintain an acceptable frame rate, a client must require state updates very frequently. In our implementation, this is done every 25 ms. For this reason, *nop* actions return immediately with the results without being distributed as Raft logs. This is an optimization on the Agreement requirement of replicated state machines, which applies to read-only operations and is discussed in Reference 2.

2.3 | Threats to applicability

Regardless of actual performance, which we discuss separately, some considerations should be made in advance about the applicability of a Raft-like distributed consensus protocol to multiplayer online games.

2.3.1 | Randomness

The replicated state machines must behave exactly the same way in response to the same input in so far as state changes are concerned. This effectively means that such state machines must be *deterministic*, which disallows random state changes.

However, most games of this kind—albeit not every one of them—rest on some degree of randomness that causes some distortion in the perception of the game state by different players. Hence, the need to require strict determinism may negatively affect applicability. To counter this problem, we might have relied on pseudo-random number generators (PRNG), but even those should be treated carefully: it is not sufficient to equally seed two PRNG-based state machines to ensure that the ensuing execution will be deterministically identical across them. The possibility of distributing snapshots, which is intrinsic to Raft, implies that it is possible that not all state machines will “see” the exact same sequence of operations. Some will simply leap over large chunks of actions if they lag behind and receive a snapshot: we discuss this problem in Section 3.6.



A solution to this challenge would be to use a function that, based on a shared seed, produced a deterministic pseudo-random result in response to a given input and feed it with the unique id-term combination of a Raft log. This measure could be realized by serializing random numbers (e.g.) through a separate Raft network, but this direction fell far off the scope of our work.

2.3.2 | Secrecy

The challenge level of some multiplayer games heavily relies on mutual player secrecy. Knowing the exact position of an adversary in a FPS game gives a tactical advantage, so does knowing which buildings and troops another player is using in a RTS game.

Having each client host a Raft node means that any information stored in the game engine can be accessed by reading the process memory or directly tampering with the code. To this challenge, there is no immediate solution that we see.

However, we note that there are pretty successful multiplayer games that require no randomness and no secrecy: chess is an emblematic example of both.

3 | THE RAFT ALGORITHM

3.1 | Raft: An outline

In this section, we first provide a general picture of the components and behavior of a Raft network as described by Ongaro and Ousterhout in Reference 3, then we explain why we chose the Go programming language for our PoC implementation, and finally we discuss some of our implementation choices, illustrating them with selected snippets of source code.

A Raft network is composed of nodes that communicate with one another via remote procedure calls (RPCs). Each such node can be in one of the following three states at any time (cf. Figure 5):

- A **Follower** passively accepts messages and heartbeats, that is, void messages solely used to maintain leadership information current, from the currently elected Leader as long as they arrive in time, which means within bounded time intervals. If a given time interval expires with no incoming RPC, a *Follower election timeout* event occurs, which causes the Follower to change its state to Candidate.
- A **Candidate** issues vote requests to all other members of the network and waits for a majority of the nodes to accept its leadership before changing its state to Leader. If a given number of time intervals expire before receiving the necessary votes, a *Candidate election timeout* event occurs, which triggers the start of a new election. A Candidate node changes its state back to Follower if a different Leader emerges during an election period.
- A **Leader** handles all client requests and produces new logs to distribute to the other nodes via RPCs; it changes its state back to Follower if a more legitimate new Leader emerges.

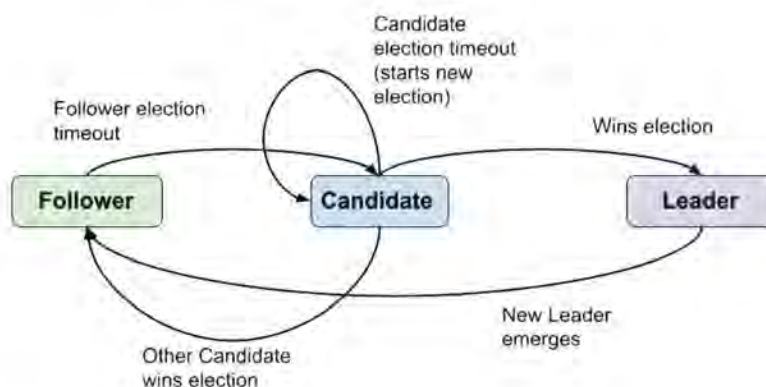


FIGURE 5 Raft node states

3.2 | Why Go

A Raft node will have to handle multiple requests from different actors (clients and other nodes acting as Candidates or Leaders). The protocol description specifically requires that such communications should be carried over RPCs, which warrants the prerequisite degree of delivery guarantee.

We decided to use the Go programming language for our PoC implementation because:

- Go is a statically typed language. It therefore supports the automatic detection of a whole load of errors, which we deemed a great aid in an already complicated project.
- Go features powerful concurrency primitives (along with other lower-level features) as well as channel-based communication, which aids the implementation of message-based protocols very much.
- Go supports RPC with a standard native package, part of its *net* module.

Moreover, as we preferred implementing the whole application in a single language, Go's offering of convenient image processing features (we used the *image* module) came handy for the implementation of our game UIs.

Our PoCs make extensive use of channel-based communication and goroutines.

Alongside that, we regarded the core state of a Raft node as a container of data that should be accessed and manipulated in a strictly sequential, non-concurrent manner. That notion lead us to implement it with a monitor data structure equipped with a mutex lock. In retrospect, however, this choice had distinct drawbacks:

- The mutex lock is not idiomatic Go programming;
- As long known to concurrent programming practice, the mutex lock is extremely vulnerable to misuse. The most common hazards occur for functions with more than one point of exit (which require unlocking the held mutex before every return, with consequent cluttering of code) and when nesting them, which is exposed to the risk of circular-wait situations.

While regretting the choice, we chose to concentrate on the more interesting aspects of our project and renounced undertaking a full re-engineering of the core state. Converting the current mutex-based logic of our implementation to a more idiomatic solution could be the subject of a future continuation of this work.

3.3 | Node states

To model the node states and the behavior described in Section 3.1, fitted an execution model based on the exchange of RPC. Accordingly, we implemented a Go version of the Actor model, making extensive use of channels and goroutines.

```
func run() {
    go handleClientMessages()
    go handleAppendEntriesRPCResponses()
    go handleInstallSnapshotResponses()
    go handleConfigurationMessages()
    for { // infinite loop
        checkLogsToApply()
        switch getState() {
            case Follower:
                // state handler
                handleFollower()
            case Candidate:
                // state handler
                handleCandidate()
            case Leader:
                // state handler
                handleLeader()
        }
    }
}
```

Listing 1: Main loop



As Listing 1 shows, each state handler accepts messages incoming from one channel before returning and letting the main loop iterate over. The goroutine handlers, launched with the keyword `go`, are inner infinite loops, used to serve frequent requests without starving the main loop.

Raft uses *terms* to divide time and provide a global partial ordering on messages and events. Terms always start with an election and are governed by at most one Leader. Some terms may expire with no Leader if there is no majority for their elections. The transition between terms may be observed at different times by different nodes.

3.4 | Leader election

Each Raft node starts its life as a Follower and transitions to Candidate if it does not receive updates within a bounded time period called the *Follower election timeout*.

When a node transitions to Candidate it will:

- Increment its current term;
- Vote for itself;
- Issue *RequestVote* RPCs to all other nodes in the network, in parallel.

After issuing the RPCs, the node will accept responses and keep track of the received votes. On receiving a majority of votes, it will immediately transition to Leader for the current term. If that does not happen, two other outcomes are possible:

- On receiving an *AppendEntries* RPC for a term greater or equal than its current term, it will consider the sender as the new Leader, and then revert to Follower;
- On failing to receive a decisive number (or type) of messages within a bounded amount of time, it will initiate a new election.

In our implementation (shown in Listing 2, the *RequestVote* RPCs are parallelized by using one distinct goroutine per response:

```
func sendRequestVoteRPCs(requestVoteArgs *RequestVoteArgs) {
    connections.Range(func(id ServerID, raftConn RaftConnection) bool {
        var requestVoteResponse RequestVoteResponse
        // This is an asynchronous RPC
        // implemented in the package net/rpc
        requestVoteCall := raftConn.Connection.Go(
            "RaftListener.RequestVoteRPC",
            requestVoteArgs,
            &requestVoteResponse,
            nil)

        // This goroutine waits for a response up to
        // "electionTimeout" ms and uses the
        // myRequestVoteResponseChan to communicate
        // the results back to the Candidate
        go func(requestVoteCall *rpc.Call, id ServerID) {
            select {
            case <-requestVoteCall.Done:
                myRequestVoteResponseChan <- &requestVoteResponse
            case <-time.After(time.Millisecond * electionTimeout):
                log.Warning("RequestVoteRPC: no response")
            }
        }(requestVoteCall, id)
        return true
    })
}
```

Listing 2: *RequestVote* logic



All recipients will concurrently evaluate and respond to the request. To avoid incurring race conditions at the Candidate's end, we use channel *myRequestVoteResponseChan* to consume the responses, as shown in Listing 3:

```
func handleCandidate() {
    // Create a new timeout only if it does not already exist
    var electionTimeoutTimer = checkElectionTimeout()
    select {
    ...
    case reqVoteResponse := <-myRequestVoteResponseChan:
        becomeLeader := updateElection(reqVoteResponse)
        if becomeLeader {
            // At this point the state will be changed
            // to Leader so at the next iteration
            // the main loop will execute the handleLeader
            // function
            sendAppendEntriesRPCs()
        }

        // By using a channel to handle this timeout
        // we can make it "global" that is, having it span
        // multiple handleCandidate executions
    case <-(*electionTimeoutTimer).C:
        stopElectionTimeout()
        // Start new election
        startElection()
        // Issue RequestVoteRPCs in parallel to other nodes
        var requestVoteArgs = prepareRequestVoteRPC()
        sendRequestVoteRPCs(requestVoteArgs)
    ...
    }
}
```

Listing 3: Candidate state handler

3.5 | Log replication

A Leader will accept requests from clients and generate logs accordingly. Subsequently, it will attempt to propagate each new log entry to the network and will consider it committed when it is replicated on a majority of nodes. Only when a request is committed, will the Leader apply it to its state machine and reply to the client. Followers will apply entries based on a commit index shared by the server.

The first interesting thing to note about handling client messages is that the Leader should be able to accept an arbitrary number of requests, at an arbitrary rate. In our first implementation we had the *handleLeader* function take care of every received message, whether from the Raft network or from clients. In test scenarios with large numbers of clients and high message rate, this would cause an effective denial of service (DOS), preventing the Leader from discharging its duties.

In order address this concern, we moved the client message logic to a separate goroutine, as shown in Listing 4:

```
func handleClientMessages() {
    for {
        // Blocking receive on channel "msgChan"
        act := <-msgChan
        switch getState() {
        // Followers and Candidates respond immediately
        // with the last known leader
        case Follower:
            act.ChanResponse <- &ActionResponse{false, currentLeader}
        case Candidate:
            act.ChanResponse <- &ActionResponse{false, currentLeader}
        case Leader:
            ...
        }
    }
}
```



```

// Add message to log and propagate it
// Wait for message to be committed and applied
// before responding to the client
go handleResponseToMessage(
    act.Msg.ChanApplied,
    act.ChanResponse
)
}
}
}

```

Listing 4: Client message handler

Now, a new *handleResponseToMessage* goroutine is called for each new incoming message, with two main parameters:

- A channel called **ChanApplied**, used by a node to communicate that a particular Raft log has been committed and applied to the state machine;
- A channel called **ChanResponse**, used to send a response to the RPC listener and consequently to the client.

```

func handleResponseToMessage(...) {
    const handleResponseTimeout = 1000
    select {
    case <-chanApplied:
        chanResponse <- &ActionResponse{true, currentLeader}
    case <-time.After(time.Millisecond * handleResponseTimeout):
        log.Warning("Timeout waiting for action to be applied")
    }
}

```

Listing 5: Single response handler

As shown in Listing 4, using the buffered channel **chanApplied**, the node will not be blocked when attempting to send a late completion message in the event of a timeout. The channel will be garbage collected later, after that.

The Leader will send *AppendEntries* RPCs similarly to *RequestVote* RPCs (cf. Section 3.4).

3.6 | Log compaction

In theory, a Raft node log collection can be regarded as unbounded, containing all of the logs ever produced by the network. Obviously, this is not tenable in practice. Accordingly, the Raft algorithm proposes a log compaction method: each node can compact its logs and produce a local snapshot of its state machine as and when required, to avoid running out of space.

A Leader can send *InstallSnapshot* RPCs to nodes that are failing to keep up with the rest of the network. This provision ensures that each state machine will eventually reach the same state. It also speeds up the update process for lagging nodes.

In our implementation, logs are kept in a fixed-size array, as shown in Listing 6:

```

// This is the structure containing the "core"
// state of a Raft node
type stateImpl struct {
    ...
    logs    [1024]RaftLog
    ...
}

```

Listing 6: Raft logs container



The original Raft paper regards checking whether the logs reach a certain size in bytes as a possible strategy to decide when to take a snapshot. In our version, a node just needs to check the number of logs to add to the array when handling an *AppendEntries* RPC or when adding new client-generated logs. Before adding any new log, it will check whether the number of currently held logs plus the number of new entries is greater than 1024 (our arbitrary choice of bound) and take a preventive snapshot if needed.

The snapshot consists of a JSON representation of the game state that is directly requested to the game engine via channel communication:

```
// Raft node side
func takeSnapshot() {
    snapshotRequestChan <- true
    currentGameState := <- snapshotResponseChan
    ...
}

// Game engine side
func run() {
    var gameState = GameState{...}
    for {
        select {
            ...
            case <- snapshotRequestChan:
                jsonGameState := json.Marshal(gameState)
                snapshotResponseChan <- jsonGameState
                ...
        }
    }
}
```

Listing 7: Main snapshot logic

The snapshot contains a map of the network, which allows nodes to keep up with configuration changes (cf. Section 3.7).

Our implementation decision of taking snapshots only when needed (i.e., when there is too much data for the log array to hold) worked fine in most cases for us. However, it did result in unexpected behavior in at least one occasion, which we were able to capture with the aid of our Raft analyzer (cf. Section 5.1.2). Interestingly, such a glitch does not occur in observable behavior as it is caused by the interaction of network delays, a Follower whose log array is nearing excess capacity, and our implementation of the receiver rules for *AppendEntries* RPCs. Specifically, we check whether new entries are already present in the log one by one: we first find their position in the log array and then compare them. However, we have no way of doing this if the entries are compressed in a snapshot, as we only have the last included index and term. Consequently, logs left in the log array after a snapshot risk being overwritten.

Let us now describe a representative scenario for this bug to occur:

- A Follower node whose log array is close to maximum storage is lagging behind the Leader and receives an *AppendEntries* RPC holding $n > 1$ logs, say from index 1015 to 1023. Moreover, the *AppendEntries* RPC notifies the Follower that the current commit index is 1019.
- The Follower adds the logs to its collection, reports success to the Leader and goes on to apply logs to its state machine up to index 1019.
- Owing to network delays, the response takes some time to reach the Leader, which in the meantime sends another round of *AppendEntries* RPCs. Specifically, the one RPC directed to the Follower holds logs from 1015 to 1026 because the Leader has not yet received the already-sent successful response.
- The Follower receives the RPC and verifies that it cannot hold all the new logs: it therefore takes a snapshot. In our implementation, snapshots are taken up to the latest applied log whereby any log with index $i \leq 1019$ is discarded and logs from 1020 to 1023 are copied to the start of the array.



- The Follower now proceeds with the *AppendEntries* RPC. The only check we made initially was that the previous log index j of the incoming RPC be included in the latest snapshot (i.e., $j \leq n$, the snapshot's latest included index). Hence, the function proceeds, checking that the first log in the collection does not correspond to the first new entry in the RPC, and discards it along with all the following logs.
- This scenario leads to double insertion of logs with indexes from 1015 to 1023 in the log array.

Since we pick logs to apply to the state machine by directly extracting them based on the *latest applied* property of our node state, we argue that this anomaly would not have shown up in observable game behavior. Nonetheless, this is a bug that violates Raft's log matching security property, and we were only able to single it out by using hashes of Raft logs specifically computed to check this property.

Our fix to this bug was to add a check to purge an *AppendEntries* RPC from logs that were compacted in a snapshot. In that manner, we are always able to compare the new entries with those saved in the log array.

In retrospect, we noticed that we implemented a consistency check more strictly than described in fig. 2 of Reference 3. Specifically, we deem logs to be inconsistent if their indexes or terms are different. The original rule instead checks whether two logs have the same indexes but different terms. Whereas this logic can lead to unnecessary operations, it is never a problem when there are no limits on the size of the log array. However, as shown above, it may create inconsistencies when combined with snapshots and network delays. We acknowledge that by changing this consistency check we could remove both the bug and the necessity for our solution. As such modification would require extremely thorough testing, we felt it should leave for further work.

3.7 | Configuration change

A configuration change consists of the addition or removal of a node from the Raft network. This effect could be easily achieved by taking the network off line and then restarting it with a different configuration. Evidently, this is not a viable approach for applications with availability requirements, such as multiplayer online games. The possibility to add or remove players on the go thus becomes desirable for them.

The challenge with an on-line configuration change arises from the fact that it is not possible to atomically update a group of servers all at once. Attempting to move to a new configuration immediately, would cause a transition period in which some of the servers will have received the new network map and some will not, which could result in the election of multiple Leaders for the same term, thereby breaking the election safety property.

The Raft paper³ proposes a method to model this transition period explicitly, by first distributing an intermediate configuration $C_{new,old}$ in which new servers only belong to C_{new} , old (i.e., removed) servers only belong to C_{old} , and persisting servers belong to both. During this transition phase any majority must be established on *both* configurations, which preserves the election safety property. Once the transition configuration is committed (i.e., replicated to a majority of the servers), the current Leader will distribute a final configuration consisting only of C_{new} . Once committed to the network, the configuration change will be over.

Initially, we implemented this version of the configuration change method, but soon found it inapt for our purposes.

- The method requires servers to keep track constantly of whether a server is in a new configuration, an old configuration, or both. Moreover, every majority decision must keep track of such configurations, which requires relevant bookkeeping.
- A server should start using a new configuration as soon as it is received. Evidently, in the event of a quick succession of requests to connect to the Raft network, as many configuration changes would have to take effect. An attempt to bundle a series of close-range requests into a single configuration change would require a lot of bookkeeping and be very vulnerable to error. Generating a new configuration change before the previous one is committed would mean that some Followers would effectively change their network configuration map before having committed the previous one (e.g., a server only in C_{new} would become a server in $C_{new,old}$). This is not necessarily an error, for it should only effect the number of servers in $C_{new,old}$ as seen by each node, but it is not proven to be safe. In fact, this very situation



is addressed in Reference 7 by saying that the joint consensus algorithm could be generalized to include these cases, but that there is no real advantage in doing so.

On account of that, we eventually changed our method to the one described in Diego Ongaro's doctoral dissertation.⁷ This change greatly simplified our code and improved the overall stability and performance of our implementation.

The strength of this method (as indeed of many parts of Raft) is its simplicity: configuration change requests are dealt with one at a time and can only add or remove *one* server from the cluster. This means that any majorities in the old and new configurations will overlap, which erases the risk of having different decisions.

Ongaro proposes two new RPCs to implement this method: *AddServer* RPC and *RemoveServer* RPC. In our implementation we combined them into a single parameterized RPC. Configuration change requests are handled immediately if possible, otherwise they are kept in a frequently checked FIFO queue. The RPC returns only when the configuration change is committed to the network. This mechanism is handled in the same way as responses to client actions (cf. Section 3.5).

4 | BYZANTINE FAULT-TOLERANT RAFT

4.1 | Introduction

The Raft algorithm relies on *strong leadership* and absolute mutual trust among participating nodes. However, there are several ways in which a malicious node could undermine such trust by disrupting inter-node communication. Examples of such circumstances include the following:

- Logs are exchanged only when a Leader is established; a malicious node might keep starting new elections, causing the system to incur a DoS situation;
- A Follower could lie about having saved logs received from an *AppendEntries* RPC, thereby breaking Raft's Log Matching safety property;
- A malicious Candidate could self-promote to Leader while the rightful leader receives the majority of votes; this event would cause two leaders to claim existence in the same term, thus breaking Raft's election safety property;
- A malicious Leader might ignore client requests, stopping them from being propagated to the other nodes; or it might forge fake ones.

Copeland and Zhong⁶ describe a Byzantine fault-tolerant variant for Raft (nicked BFTRaft), which is robust and able to reach consensus as long as more than two thirds of the nodes do not exhibit Byzantine (i.e., arbitrary) behavior.

We reckon that this extension could be useful in an often competitive field as that of multiplayer video games. However, any security addition to the system should be weighted thoroughly against its toll on performance.

To this end, we reproduced this extension in our PoC and tested it against the standard Raft protocol in the 1:1 architecture variant. The enhanced version is implemented in branch *byzantine-behavior* of our public repository.

In the remainder of Section 4, we describe the prerequisite Byzantine fault-tolerant extensions along with our implementation choices.

4.2 | Use of signatures

To avoid the risk of forgeries and to verify the integrity of the communicated data, all RPCs exchanged by servers are signed, as are client messages. The signatures are based on public-key cryptography (RSA). Hence, all system nodes should have access to the public-keys of other nodes and clients.



Each message is immediately verified for authenticity and integrity. If signature verification fails, the incoming message is simply ignored.

Go offers an implementation of RSA with the *crypto* module, which eased the implementation of this feature.

4.3 | Incremental hashing

In a trusted environment, having only one Leader per term (i.e., one source of logs) and checking that the entries of an *AppendEntries* RPC's recipient contain a log whose term and index match the *prevLogTerm* and *prevLogIndex* parameters of the RPC is sufficient to guarantee that the Leader's log collection is replicated with no gaps or re-orderings.

This condition of course no longer holds in a Byzantine environment. In BFTRaft, therefore, each log is hashed against the hash of the previous log so as to provide an inductive proof of exact replication of the log collection.

The hashes are sent along *AppendEntries* RPCs: their verification replaces the checks made on index values. In this manner, a Follower may rest assured that it shares the collection with the Leader up to that point.

In fact, we included this concept in our nominal (non-Byzantine) version of Raft with the intent of conducting checks on the Log Matching safety property, whose nature and outcomes we discuss in Section 5.1.

4.4 | Log replication

Because of the impossibility of having a single source of truth in a Byzantine environment, the log replication process involves broadcast messages in two steps:

- Clients cannot know for certain that the Leader they are currently sending messages to is actually truthful when reporting that an action has been committed to the replicated state machines. To mitigate this risk, nodes broadcast their actions to all the nodes and wait for a response from a quorum of $f + 1$ of them before deeming the action applied, where f is the maximum number of faulty nodes the network is able to sustain. The idea here is to allow at least one fully functioning node to have trust that the action was committed.
- Followers cannot rely uniquely on Leader-communicated commit indexes. Accordingly, *AppendEntries* RPC responses are broadcast along with the hash of the last inserted log in order that every node can verify that each log collection is consistent with its own and update its commit index itself.

To maintain liveness and avoid a DoS by a Leader refusing to apply actions, clients can broadcast *UpdateLeader* RPCs to all other nodes. The receiver nodes will thus stop listening to heartbeats for the current term and eventually incur an election timeout that will lead to the election of a new Leader. Nodes will still accept *AppendEntries* RPCs containing entries, which ensures protection against malicious *UpdateLeader* RPCs.

4.5 | Leader election

A node that should keep starting new elections would be sufficient to starve a Raft network. As a mitigation against that, BFTRaft introduces lazy voting: a node will grant a vote for a Candidate according to the usual rules, but it will only send that vote back on incurring an election timeout itself. Accordingly, no votes will be cast as long as there is a lively Leader.

Another form of malicious behavior would be for a node to increase its term arbitrarily and start sending heartbeats as a new Leader. In standard Raft, this deed would be sufficient to guarantee leadership as nodes immediately step down to Follower status on seeing higher terms. BFTRaft mitigates that risk by having Leaders add the votes they received to *AppendEntries* RPCs in order for Followers to verify their integrity and authenticity with the relative signatures. Only after this verification will a Follower actually increase its term and start adding logs to its collection.



4.6 | Cautionary considerations

4.6.1 | When to increase a node's term

We found a potential flaw in the design of this extension, which has to do with the rules for increasing a node's term. The authors enumerate three cases in which a node should increase its term:

1. When receiving an *AppendEntries* RPC with a quorum of votes;
2. When voting for a Candidate;
3. When becoming a Candidate.

Our experiments highlighted a situation in which these rules may cause some network nodes to starve. An exemplary scenario to that effect would work as follows:

- A Leader is elected for a particular term and starts sending heartbeats and entries to the other nodes;
- For any reason (e.g., network delays), a Follower does not receive heartbeats and exceeds its election timeout. This is one of the three occasions in which it should increase its term, and so it does;
- Because of lazy voting, the Follower does not receive enough votes from the rest of the network and does not become a new Leader;
- As its term is higher than the current Leader's, the Follower will keep rejecting heartbeats and *AppendEntries* RPCs, as per the nominal Raft rules, whereby it will be stuck in a Follower-Candidate cycle until the current Leader steps down.

To mitigate that hazard, we allowed Leaders to update their term if smaller than one contained in an *AppendEntries* RPC response. This means that if the current Leader, with term n , received an *AppendEntries* RPC response with term $m > n$, then it would update its term to m ; by doing this all subsequent heartbeats would be parameterized by term m and the Follower would accept the current Leader. We reckon this measure should not impair the correct functioning of BFTRaft: clients still are able to request the removal of a Leader; signatures and log hashes are still checked, and so forth.

4.6.2 | Specific downsides of our implementation

The use of signatures and the ability to identify attempted message spoofing would allow a robust network to automatically purge malicious nodes.⁸ However, our implementation of BFTRaft is unable to do this: the only defence mechanism against malicious nodes that we implemented was to discard their messages. This shortcoming is not deliberate, as it reflects a limitation of the Go library we used for RPCs (the *rpc* module), which hides the URL of the invoker from the listener and therefore prevents corrective action. The provenance URL would be required to tell unverified signatures, since we cannot trust all message contents of course.

In truth, other Go RPC libraries exist that allow this identification, for example, the library available at <https://github.com/valyala/gorpc>. Unfortunately, however, when we realized that need, it was past the time we could afford such a large change in our PoC implementation.

5 | EVALUATION

5.1 | Proving correctness

5.1.1 | Unit tests

Albeit theoretically simple and built for understandability, a full Raft implementation still remains a complex program, especially in the regard of asserting the correctness of its working.

The behavior of a single Raft node is well defined by a series of specifications, which are summarized in fig. 2 of the Raft extended paper.³ We built a series of unit tests to check our implementation against those instructions.



TABLE 1 Raft safety properties from Reference 3

Election safety	At most one leader can be elected in a given term.
Leader append-only	A leader never overwrites or deletes entries in its log.
Log matching	If two logs contain an entry with the same index and term, then the logs are identical in all entries up to the given index.
Leader completeness	If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
State machine safety	If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry at the same index.

The tests, as part of the *go_raft* module (our core implementation of Raft), can be ran by launching the *go test* command from the *go_raft* directory of our project repository.

5.1.2 | Raft analyzer

While unit tests can offer some level of assurance on the correct behavior of individual functions and small parts of a program, a complex system involving multiple actors also needs other types of verification. The Raft paper mentions 5 safety properties that should be true at all times for correct communication to take place, but there is no way of verifying them without looking at the system as a whole.

For this reason, we built a simple tool to (a) aggregate the log traces produced by Raft nodes during their normal operation, (b) sort them by timestamp, and finally (c) analyze them to determine whether the safety properties hold. We called this tool *Raft analyzer* and provided it with a simple graphical web interface from which one can conveniently observe the exchanges that take place in the recorded communication.

The analysis is performed by replaying the communication while cycling through the logs and keeping the state current. We now describe the analysis required for the properties of interest, which are recalled in Table 1.

The following paragraphs describe how we practically approach the analysis in our Go implementation of Raft nodes and in *Raft Analyzer*.

Election safety

A node produces a log indicating the current term every time it becomes Leader. It is sufficient to check that no two leader logs share the same value.

Leader append-only

A node produces a log every time it removes elements from its logs, in our implementation this can only happen in two functions. The analysis keeps track of the current leader and reports if it signals a log removal.

Log matching

This property is more difficult to prove because it involves *all logs* up to a particular index-term combination. We took inspiration from Reference 6 and attached to every Raft log a hash produced from the previous log hash and the byte representation of the current log. This inductive hashing ensures that if two logs share the same hash then they should also share every log up to that point. During analysis we keep track of each term-index hash combination and report divergences.

Leader completeness

A node produces a log every time it attaches a new entry. This allows us to keep track of the latest log it received. Moreover, we log every new commit index, which allows us to compare the latest log index signaled by a new Leader against the current commit index.

State machine safety

A node will trace the index of any log it finally applies to its state machine along with a byte representation of its content (i.e., the game action). For this reason, we keep track of the byte representation of every applied log and report divergences.



5.2 | Measuring performance

To get performance data, we tested our two PoCs in a variety of scenarios. To enable automatic generation of action data, we ran clients in a very simple “bot” mode, which yields random actions at fixed intervals:

- In the **go_skeletons** PoC, a bot generates an action approximately every ~200 ms, to simulate a faster paced real-time game.
- In the **go_wanderer** PoC, a bot generates an action approximately every second, to simulate a slower game where players have more time to think between actions.

Our evaluation metrics are as follows:

- **Action delay:** the time span between the moment a client sends a new action to the Raft network and the moment it receives a successful response (i.e., after the action has been committed to the network and applied to most replicated state machines). We consider results below ~20 ms to be good, results between ~20 and ~200 ms to be acceptable, and higher values to be bad.
- **Actions per client:** the number of successful actions a client is able to perform during its lifetime.
- **Actions per second:** the number of actions per second a client is able to perform during its lifetime. For both this metric and the preceding one, the target values are roughly based on the frequency at which the corresponding bots generate actions in the two games (ideally, 5 actions per second in **go_skeletons**, and 1 action per second in **go_wanderer**).

The devised three test scenarios, as follows:

- A **basic** scenario, in which nodes run undisturbed for the total duration of the test. Each test runs for ~60 s after all nodes are up and networked. The results are based on a median of 5 subsequent runs. For this scenario, we collected results while varying the number of clients from 5 to 105.
 - For the **1:1 architecture** this means also varying the number of nodes in the Raft network.
 - For the **Cluster architecture** we kept a fixed number of 5 Raft nodes.
- A **dynamic** scenario, in which some nodes are first shut down in a clean way, and then restarted after 10 s. To this end, we use a SIGTERM signal, whose handling triggers a configuration change (cf. Section 3.7).
- A **faulty** scenario, in which individual nodes are shut down abruptly (which we accomplish by killing them with a SIGKILL signal) and restarted after ~10 s. For both this and the dynamic scenario, the Raft network consists of 5 nodes. Each test runs for ~120 s after all nodes are up and networked. The results are based on a median over 5 subsequent runs. We vary the kill frequency and test for a kill after ~50, ~30, or ~10 s from the latest restart.

We ran the Raft network nodes on distinct processes communicating through dedicated ports at localhost. As this setting does not reflect a real-world scenario, we ran some tests while simulating network delays in localhost with the `tc` and `netem` commands, adding a ~40 ms latency, plus/minus ~10 ms and a degree of correlation among subsequent packet delays.

We ran all tests on a laptop with eight 2.60 GHz cores and 16 GB of RAM.

5.2.1 | Results for the 1:1 and Cluster architecture models

Tables 2 and 3 report the results obtained for **go_skeletons** in the basic scenario, on normal and delayed localhost.

These results immediately show that the *Cluster architecture* has the best performance both in normal localhost and in the event of delays. Both architectures suffer increased delays as the number of clients grows. Such delays soon reach unacceptable levels in the *1:1 architecture* model, whereas they always stay under or close to acceptable levels in the *Cluster architecture* model.



TABLE 2 Action delays (ms) for **go_skeletons**: these results show how badly the *1:1 architecture* scales when the network size reaches ~60 nodes and the delays start to exceed 200 ms; the *Cluster architecture* instead always stays within acceptable levels

No. of clients	Normal		Network delays	
	1:1	Cluster	1:1	Cluster
5	12.000	11.407	19.981	15.985
25	22.606	11.725	28.399	16.766
45	30.121	12.884	30.183	18.719
65	307.697	13.154	273.447	19.326
85	501.599	13.470	316.532	19.660
105	416.683	13.980	400.163	34.418

TABLE 3 Median actions per client in **go_skeletons**: these results show that the *Cluster architecture* is able to sustain approximately 3.5 actions per second, that is, more than 200 actions per minute for each player regardless of the number of nodes

No. of clients	Normal	
	1:1	Cluster
5	223.000 (3.54/s)	220.680 (3.53/s)
25	268.048 (3.67/s)	256.744 (3.54/s)
45	304.320 (3.68/s)	288.453 (3.5/s)
65	245.375 (2.59/s)	321.126 (3.47/s)
85	191.642 (1.77/s)	355.245 (3.47/s)
105	155.905 (1.26/s)	390.166 (3.47/s)
No. of clients	Network delays	
	1:1	Cluster
5	231.560 (3.68/s)	216.240 (3.45/s)
25	267.576 (3.66/s)	250.823 (3.46/s)
45	305.862 (3.68/s)	286.671 (3.48/s)
65	248.415 (2.63/s)	323.883 (3.50/s)
85	199.925 (1.86/s)	356.791 (3.48/s)
105	146.072 (1.16/s)	380.459 (3.38/s)

Note: Conversely, the *1:1 architecture* starts losing performance around ~65 nodes.

Somewhat surprisingly, the action delays for the *1:1 architecture* model appear to be better with network delays for higher number of client nodes. This phenomenon likely reflects the lighter transport load that the delays cause on the network.

Table 3 shows how the *Cluster architecture* always manages to stay in the vicinity of 3.5 actions per second, hence more than 200 actions per minute, which would reflect a rather good game performance for experienced players.

Table 4 shows action delays for **go_wanderer**. In both architecture models, the action delays in this PoC always stay within acceptable levels. Once again, however, the *Cluster architecture* provides slightly better results for higher number of nodes.

Table 5 shows the number of actions and actions per second metric indicators for **go_wanderer**. In those dimensions, the *Cluster architecture* nearly doubles the performance of the *1:1 architecture*.



TABLE 4 Action delays (ms) for **go_wanderer**: these results show better scalability (compared to Table 2) for the *1:1 architecture*, which stays within acceptable levels up to ~85 nodes; again, the *Cluster architecture* always stays within acceptable levels

No. of clients	Normal		Network delays	
	1:1	Cluster	1:1	Cluster
5	11.664	11.180	20.474	16.280
25	21.572	13.389	29.108	17.801
45	47.663	22.180	52.201	25.822
65	98.815	43.839	104.926	38.781
85	166.781	48.957	167.096	43.207
105	273.139	48.086	276.661	46.313

TABLE 5 Median actions per client in **go_wanderer**: in this test, where the goal is to perform 1 action/second, the *Cluster architecture* once again shows better performance as the number of nodes rises

No. of clients	Normal	
	1:1	Cluster
5	39.680 (0.63/s)	61.800 (0.99/s)
25	31.464 (0.43/s)	61.960 (0.86/s)
45	33.480 (0.40/s)	61.978 (0.77/s)
65	33.997 (0.36/s)	61.978 (0.70/s)
85	38.085 (0.37/s)	61.381 (0.63/s)
105	37.171 (0.33/s)	61.114 (0.59/s)
	Network delays	
	1:1	Cluster
5	37.600 (0.60/s)	61.920 (0.99/s)
25	33.448 (0.46/s)	62.280 (0.86/s)
45	32.916 (0.40/s)	62.507 (0.77/s)
65	34.049 (0.37/s)	62.234 (0.70/s)
85	36.704 (0.35/s)	61.941 (0.64/s)
105	37.815 (0.33/s)	61.790 (0.59/s)

Tables 6 and 7 show action delays in dynamic and faulty situations. All of these results stay within optimal levels.

Tables 8 and 9 show the median number of actions and actions per second in dynamic or faulty situation.

For the **go_skeletons** case, the *1:1 architecture* appears to suffer some (albeit modest) performance decay as the kill timeout shortens. The *Cluster architecture* appears to be more stable; this is especially evident for **go_wanderer**.

A kill timeout of ~50 s in effect means that a node will exit the network every minute or so on average, and a new node will immediately come replace it. (In fact, in our tests, it is the same node that comes and goes, but this is immaterial to the experiment results.) This frequency of departures and arrivals is highly unusual in real-world situations, and particularly harsh at that. The statistics we collected for this situation however are comparable to those arising in the normal situation, which makes us think that a Raft network can be quite robust to configuration changes and single node failures.

Overall, we find these results promising. While they show that the *1:1 architecture* model is not really fit for a real-time game with a large number of nodes, they also show that the *Cluster architecture* model yields good or acceptable performance even in the face of network delays.



TABLE 6 Action delays for **go_skeletons**: dynamic/faulty (ms)

Dynamic	Normal		Network delays	
Kill timeout	1:1	Cluster	1:1	Cluster
50	10.146	11.797	15.882	16.929
30	9.884	11.612	15.785	17.212
10	8.512	11.933	14.973	18.313
Faulty				
50	10.254	11.529	15.972	16.981
30	9.948	11.593	15.817	17.217
10	8.578	11.877	15.150	18.476

TABLE 7 Action delays for **go_wanderer**: dynamic/faulty (ms)

Dynamic	Normal		Network delays	
No. of clients	1:1	Cluster	1:1	Cluster
50	10.053	11.228	15.975	17.070
30	9.733	11.465	15.625	17.103
10	8.705	11.434	14.603	18.876
Faulty				
50	10.569	11.428	15.682	16.988
30	10.986	11.340	15.914	17.070
10	10.036	11.561	14.185	18.672

TABLE 8 Median actions per client in **go_skeletons**: dynamic/faulty

Dynamic	Normal	
Kill timeout	1:1	Cluster
50	420.160 (3.50/s)	424.320 (3.57/s)
30	411.920 (3.43/s)	423.600 (3.57/s)
10	403.800 (3.36/s)	420.800 (3.54/s)
Faulty		
50	419.840 (3.50/s)	413.960 (3.49/s)
30	410.320 (3.42/s)	429.360 (3.62/s)
10	398.200 (3.32/s)	425.960 (3.58/s)
Dynamic	Network delays	
Kill timeout	1:1	Cluster
50	422.160 (3.61/2)	416.600 (3.50/s)
30	408.080 (3.40/s)	425.960 (3.58/s)
10	397.640 (3.31/s)	423.520 (3.56/s)
Faulty		
50	418.640 (3.49/s)	413.560 (3.48/s)
30	410.360 (3.42/s)	420.040 (3.54/s)
10	394.520 (3.29/s)	420.880 (3.54/s)



TABLE 9 Median actions per client in **go_wanderer**: dynamic/faulty

Dynamic	Normal	
Kill timeout	1:1	Cluster
50	71.000 (0.59/s)	119.800 (1.00/s)
30	68.120 (0.57/s)	120.000 (1.00/s)
10	56.480 (0.48/s)	120.000 (1.0/s)
Faulty		
50	31.000 (0.42/s)	119.000 (1.00/s)
30	19.400 (0.21/s)	119.000 (1.00/s)
10	8.960 (0.08/s)	120.000 (1.00/s)
Dynamic	Network delays	
Kill timeout	1:1	Cluster
50	70.560 (0.59/s)	119.000 (1/s)
30	73.280 (0.61/s)	119.0000 (1.00/s)
10	45.600 (0.38/s)	119.000 (1.00/s)
Faulty		
50	31.680 (0.39/s)	119.000 (1.00/s)
30	20.120 (0.22/s)	119.000 (1.00/s)
10	7.560 (0.07/s)	119.000 (1.00/s)

5.2.2 | Results for BFTRaft

We built our version of BFTRaft on top of the *1:1 architecture*. When we attempted to run it in the same scenarios as the other tests we soon found out that our host device was unable to sustain a very large BFTRaft network. In particular, we observed that any test run with more than 20 nodes would saturate our CPU and consequently yield extremely poor performance, owing—probably—to the much higher number of messages exchanged in the broadcast RPCs.

At that point, we thought of building a *Cluster architecture* variant of BFTRaft, but this proved much harder than expected. The central problem we encountered was that, as in BFTRaft a client will get the current game state from node replies, we should apply to these returned values the same reasoning we apply to all responses from nodes, that is, we should check that a quorum of $f + 1$ nodes agrees on a game state. This might be done by hashing the game state and comparing hashes to determine a consensus. On reflection, we deferred this arrangement to future work. Thus, our results for BFTRaft are based on a *1:1 architecture* of nodes running the byzantine fault-tolerant variant of the algorithm described in Section 4.

We tested our implementation in two different scenarios:

- A **best case** scenario, in which n BFTRaft nodes behave normally (i.e., there are no rogue nodes).
- A **worst case** scenario in which, for n BFTRaft nodes, there are $f = \lfloor (n - 1)/3 \rfloor$ rogues who perform a *DoS attack* on the others (for more details cf. Section 5.3.2).

Tables 10 and 11 show the results we obtained for the **best case** scenario. While they look quite discouraging in the present state, we regard the results for **go_wanderer** as coherent with the results shown in fig. 2 of Reference 8, which describes a Byzantine fault-tolerant distributed game engine built on similar foundations to BFTRaft, which shows a delay of around ~ 200 ms for a cluster of 16 nodes.

Tables 12 and 13 show the results we obtained for the **worst case** scenario. Both games clearly suffer as the number of nodes (and rogues) increases. Overall, the turn-based game **go_wanderers** shows more robustness and keeps its performance close to acceptable up to a cluster size of $n = 15$ (with $f = 4$).



TABLE 10 Byzantine results for **go_skeletons** (best case scenario); these results show how the overhead imposed by the byzantine requirements greatly affects the performance of real-time games as the delays only stay within acceptable levels for less than 10 nodes

No. of nodes	Action delays (ms)	No. of actions
5	24.010	216.000 (3.60/s)
10	345.175	158.620 (2.64/s)
15	158.655	47.533 (0.79/s)
20	225.674	16.350 (0.27/s)

TABLE 11 Byzantine results for **go_wanderer** (best case scenario); these results show better performance (compared to Table 10) as the delays always stay close or under acceptable levels

No. of nodes	Action delays (ms)	No. of actions
5	18.779	37.760 (0.63/s)
10	86.082	31.680 (0.53/s)
15	193.884	26.800 (0.44/s)
20	241.720	8.490 (0.14/s)

TABLE 12 Byzantine results for **go_skeletons** (worst case scenario)

No. of nodes	Action delays (ms)	No. of actions
5	36.975	216.360 (3.60/s)
10	345.368	155.340 (2.58/s)
15	259.776	20.693 (0.34/s)
20	249.537	1.257 (0.02/s)

TABLE 13 Byzantine results for **go_wanderer** (worst case scenario)

No. of nodes	Action delays (ms)	No. of actions
5	33.328	37.850 (0.63/s)
10	91.601	34.343 (0.57/s)
15	217.326	16.104 (0.27/s)
20	254.057	2.557 (0.04/s)

These results are not surprising: a drop in performance is not unexpected for a service under attack. It would be interesting to study exactly how the performance varies before, during and after an attack (e.g., how quickly can the network recover after the attack stops), we leave this investigations to further work.

5.3 | Security

BFTRaft is the blue-print of fault-tolerant Raft networks able to face Byzantine behavior. To check its capabilities for real, we tested BFTRaft on top of the *1:1 architecture*, with some Byzantine nodes playing the rogue.



We chose two DoS scenarios inspired by examples of correctness and availability vulnerabilities of Raft described in Reference 6 and one Impersonation scenario to test the signature verification of BFTRaft.

The test environment is **go_skeletons** running on 5 nodes with ids 6666–6670 for a duration of ~60 s. We used game logs to extract the results. In presenting the results, we contrast nominal Raft versus BFTRaft.

5.3.1 | DoS towards Client nodes: Leader not applying actions

In this scenario, a rogue Leader gets elected and then omits to propagate client actions to the network while still reporting them as committed.

Raft

Test results for this run indicate a median action delay of ~0.42 ms (because the rogue Leader responds immediately) and of 229 actions per node. However engine logs indicate that no action was actually applied.

BFTRaft

Shortly into the test, the clients start broadcasting *UpdateLeader* RPCs to all other nodes, which consequently stop accepting heartbeats from the rogue Leader and quickly elect a new one. Soon after that, logs are propagated correctly and the clients end up with around 200 actions applied each.

5.3.2 | DoS towards Raft nodes: A candidate keeps starting new elections

In this scenario a rogue Candidate keeps starting new elections without ever actually becoming a Leader.

Raft

In the time of the test run, the rogue Candidate starts 290 elections. No Leader is ever elected in them, as a consequence of which, no Raft log is ever created or propagated.

BFTRaft

The rogue Candidate launches 530 elections, but is always ignored owing to the presence of a legitimate, active Leader. The Raft network keeps on working normally, ignoring the DoS RPCs. The approximately double number of rogue elections is explained by the fact that the Candidate never receives a response and is actually able, although to no avail, to cycle through its malicious logic much faster.

5.3.3 | Impersonation: A client sending actions for another

In this scenario a rogue Client sends spoofed messages, impersonating another, adding a 1000 offset to their action ids. The actions themselves are irrelevant, the only goal is ensuring that no legitimate client action, with id < 1000, will ever be accepted, as Raft nodes only accept actions with increasing ids.

Raft

From the engine's point of view, the last applied action for node 6668 (the "victim") has id 1233, this means that the rogue Client was able to send 234 spoofed messages with ids in range $[1000 + 0, \dots, 1000 + 233]$.

From Node 6668's point of view, it generated actions from id 0 to 267 and always got a successful response because the Raft network thought the actions were already applied, having seen the higher ids of the spoofed actions.

BFTRaft

The last engine-applied action for node 6668 has id 199, which is legitimate because our rogue Client always sends action ids ≥ 1000 . The rogue Client's actions were always immediately discarded because of bad signatures.

We summarize the results of our security tests in Table 14.



TABLE 14 Security tests summary

Attack	Effects on Raft	Effects on BFTRaft
DoS towards Client nodes	Correctness violation: Clients receive responses from the network but no action is ever applied to the game engine, that is, no log is ever committed.	Clients force the network to change Leader by sending <i>UpdateLeader</i> RPCs. From then on, logs are replicated normally.
DoS towards Raft nodes	Availability violation: Raft nodes keep handling rogue <i>RequestVote</i> RPCs and no Leader is ever elected.	Lazy voting keeps nodes from falling for rogue elections; the rogue node is ignored while the network keeps functioning correctly.
Impersonation	A victim Client is effectively cut out of the game by a rogue Client and its actions are never applied to the game engine.	All BFTRaft nodes immediately recognize the spoofed nature of the rogue actions by signature verification and consequently ignore them.

6 | RELATED WORK

We now summarize work related to Raft and verging on the subject of distributed game architectures.

Heidi Howard published several articles on Raft^{9,10} along with a detailed technical report.¹¹ All of them helped us study and understand the Raft protocol. In particular, article 9 discusses a fresh implementation of the Raft algorithm and an attempt to reproduce the original article's results. That work overlaps in part with our effort, but it differs for goals and method of evaluation.

Fuhrer, Mostéfaoui and Pasquier-Rocha, in Reference 12 discuss *MaDViWorld*: a software framework for massively distributed virtual worlds. The worlds supported by this architecture consist of *rooms* that can be hosted on different machines and are populated by *objects*, for example, games like tic-tac-toe. Different rooms can be connected through *doors*, and users interact with this virtual environment via *avatars*, that is, virtual representations of clients. In general, users are only aware of objects and other users if they are simultaneously present in the same room (they also have a list of nearby rooms, connected through doors). This is the main strength of this architecture: the *global* state of the world is effectively distributed on different machines, thus reducing the load and the danger of bottlenecks. This is also the main difference from our approach: in our architectures, each Raft node maintains the complete log of a game, that is, the global state. We think this aspect opens up the possibility for more expressive games supporting a wider variety of events (e.g., world-changing spells, which affect the entire virtual environment). However, some level of separation could indeed improve performance and could be applied to our architectures, for example, by having different game levels hosted on different clusters.

Arora et al. in Reference 13 discuss improvements to the Raft algorithm implemented in the open-source database CockroachDB. The main concept is that of *quorum reads* aimed at reducing the Leader's workload: a Client sends a read request to a majority of nodes, each of which responds with the last stable value for the key requested (CockroachDB works as a key-value map) and the relative timestamp. At least one server in any majority will contain the latest committed value, which will correspond to the highest timestamp (there are some edge cases further discussed in the article). This approach reminds of the one used by BFTRaft Clients to ensure consistent reads (cf. Section 4) and of the Leader bottleneck problems, which we solved by using Go's strong concurrency primitives (cf. Section 3).

Deyerl and Distler in Reference 14 present Niagara: a distributed architecture based on Raft with the goal of improving its scalability. The architecture consists of a series of servers running loosely coupled Raft instances, which do not interact with each other but only with instances running on other machines. Each server runs a further component, called *sequencer*, which deterministically merges the logs of the different instances. By running loosely coupled Raft instances on different cores, each server should maximize the performance improvement gained from parallel execution. The results of this work seem interesting and could be used for example, to improve the scalability of real-time games (cf. Section 5.2).

Wada et al. in Reference 15 discuss a cheat resistant online game protocol based on a Byzantine agreement protocol and a lockstep protocol that is used to prevent so-called time-cheats. Those are a form of cheating in which a player gains advantage by being the last one to make their move for a given turn, after seeing all the other players' moves. In a lockstep protocol, each player must send their next move's hash to all other players before publicly disclosing the actual contents of the move. This measure allows players to later check that the contents of the actual actions are equal to the ones declared



in the hashes, before players' intentions were disclosed. In our work, the *1:1 architecture* would be vulnerable to this kind of cheating for **go_wanderer**. A lockstep protocol could thus be an interesting addition.

Martel et al. in Reference 8 discuss Lila: a distributed multiplayer Byzantine fault-tolerant game engine. The architecture of this engine resembles our *1:1 architecture* in that each player also hosts a replicated node. The Byzantine fault-tolerant algorithm is based on Liskov and Castro's work in Reference 16, as is the BFTRaft extension we re-implemented from Reference 6 and discuss in Section 4. One interesting aspect of the analysis conducted in this work is that the size of the cluster is limited to at most 16 nodes, which reduces the scope of game categories that could run on this architecture.

Bursztein et al. in Reference 17 discuss a tool that can be used to hack RTS games. The hack lifts the so-called fog of war and proceeds to propose methods to distribute game state among players, which greatly reduce the amount of leaked game state information by using oblivious functions. One of the final sections of the article discusses a method to prevent active attacks inside single game engines (e.g., favoring a player by tampering with the random number generator), which involves sharing signed hashes of the game state during play and finally having a trusted party verify them along with individual traces of the performed actions (i.e., logs of game execution). A method of this kind could be applied to an execution in the *Cluster architecture* to verify that each node's engine actually performs the received actions (the integrity of which is ensured by the Raft protocol) and does not engage in malicious behavior.

Yuen et al. in Reference 18 discuss a blockchain-based peer-to-peer gaming system. They replace the classic Proof-of-Work (PoW) consensus model with Proof-of-Play (PoP): blockchain users gain the right to write a new block by an evaluation of the effort they put in the game. This can be done regardless of the presence of actual monetary rewards, the idea is that this system should be transparently used to form consensus on the match history of already-engaging games. The computational cost of rewriting the blockchain in PoW is replaced in PoP by the actual time and mental effort spent in playing the games (i.e., instead of controlling 51% of the computational power, malicious actors should be the ones putting 51% of the effort in the games themselves). One major difference from our work, design-wise, is that consensus here is used to preserve a history of game results and not a sequence of moment-to-moment game interactions.

7 | CONCLUSIONS

Finally, we draw conclusions on the work we presented in this article and on the lessons we learned from it.

Arguably, our results suggest that a consensus algorithm like Raft could be used to support some types of multiplayer video games, especially turn-based ones like **go_wanderer**. We also show how two different architectures (fully distributed vs. clustered) scale according to quality-of-life parameters such as the delay between user actions and game feedback (cf. Tables 2 and 4). We tried to produce test results as rigorously as possible. This caused our test cycles to often take a long time (e.g., the results of Tables 2 and 3 combined take approximately one hour per column to produce) and made us limit our comparative evaluations to "normal" versus "delayed" situations. We considered other possibilities (packet drops and reordering), but eventually discarded them owing to the excess effort that they would incur.

Other than the observable results, we based our confidence in the correctness of our implementation in both unit and integration tests and checked many communication traces against Raft's safety properties. Doing so allowed us to catch subtle bugs in our implementation.

We regarded as interesting to explore extensions to the algorithm and developed a version of BFTRaft for which we drew results comparable to those of other published works (cf. Section 5.2.2).

Of course, our work has downsides and limitations, which in most cases we intentionally abode, in order to focus on what we considered the more interesting parts of our research, in the time we scheduled for it:

- Our BFTRaft implementation is less robust than the *1:1* and *Cluster architectures*. We could not afford to update our unit tests and, more importantly, to properly test all of its specific mechanisms and assumptions.
- Our *raft analyzer* is not very efficient and can only verify traces produced by small clusters for a few minutes of communication. Indeed, there are some quick improvements that can be made to it (e.g., clearly dividing traces destined to the analyzer and other logs in order to reduce the amount of data to go through), but a solid tool aimed at the analysis of an arbitrary communication trace should be probably based on a different design;
- As noted in Section 3, some of our Go code in the PoC implementations is not very idiomatic and could be much improved.



All in all, we think that the good results obtained by the *Cluster architecture* with turn-based games are not surprising: the Raft algorithm is agnostic with respect to the type of data being replicated. On the other hand, the less promising results we obtained for real-time games highlight how solutions to robustness requirements (i.e., replication) can weigh down an application and go against performance requirements. This problem could potentially be addressed by using well known techniques such as client-side prediction and this could be the interesting subject of further work.

To summarize and confirm our contributions:

- We implemented an open-source version of Raft in Go: our codebase is publicly available along with instructions on how to install it and run tests. The discussion of our implementation choices presented in this article, along with the code documentation, should provide a good basis to improve and extend it.
- We have shown how turn-based games running on a small cluster of Raft nodes can have good performance.
- We implemented an alternative version of BFTRaft which can be compared against the original (available at <https://github.com/chrisnc/tangaroa>) to gain better understanding of its mechanisms.
- We implemented *raft_analyzer* and detailed its main ideas in this article; these techniques, along with our public code, can be used as building blocks for better analysis tools.
- Our test results show empirically the trade-off between robustness and performance requirements.

ACKNOWLEDGEMENT

Open Access Funding provided by Universita degli Studi di Padova within the CRUI-CARE Agreement. [Correction added on 24 May 2022, after first online publication: CRUI funding statement has been added.]


AUTHOR CONTRIBUTION


Gabriele Pozzan: Investigation; software, writing - original draft. **Tullio Vardanega:** Conceptualization; project administration; supervision; writing - review and editing.

DATA AVAILABILITY STATEMENT

Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

ORCID

Gabriele Pozzan  <https://orcid.org/0000-0002-3575-6233>

Tullio Vardanega  <https://orcid.org/0000-0002-0089-0889>

REFERENCES

1. Cristian F. Understanding Fault-Tolerant distributed systems. *Commun ACM*. 1991;34(2):56-78.
2. Schneider FB. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput Surv*. 1990;22(4):299-319.
3. Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. *USENIX ATC '14*. USENIX; 2014;305-320.
4. Lamport L. The part-time parliament. *ACM Trans Comput Syst*. 1998;16(2):133-169. doi:10.1145/279227.279229
5. Jimenez-Diaz G, Gonzalez-Calero PA, Gomez-Albarran M. Role-play virtual worlds for teaching object-oriented design: the ViRPlay development experience. *Softw Pract Exp*. 2012;42(2):235-253.
6. Copeland C, Zhong H. Tangaroa: a byzantine fault tolerant raft. Technical report. Stanford: Stanford University; 2016.
7. Ongaro D. *Consensus: Bridging Theory and Practice*. Vol 1. Stanford University; 2014.
8. Martel L, Tu S, Moreland A. Lila: a cheating-resistant distributed game engine; 2014. Accessed June 10, 2021. https://www.scs.stanford.edu/14au-cs244b/labs/projects/martel_tu_moreland.pdf
9. Howard H, Schwarzkopf M, Madhavapeddy A, Crowcroft J. Raft refloated: do we have consensus? *ACM SIGOPS Operat Syst Rev*. 2015;49(1):12-21.
10. Howard H, Mortier R. Paxos vs Raft: have we reached consensus on distributed consensus? *PaPoC '20*. Association for Computing Machinery; 2020.
11. Howard H. ARC: analysis of raft consensus. Technical report. Cambridge, UK: University of Cambridge, Computer Laboratory; 2014.
12. Fuhrer P, Kouadri Mostéfaoui G, Pasquier-Rocha J. MaDViWorld: a software framework for massively distributed virtual worlds. *Softw Pract Exp*. 2002;32(7):645-668.
13. Arora V, Mittal T, Agrawal D, et al. Leader or majority: why have one when you can have both? improving read scalability in raft-like consensus protocols; 2017:USENIX Association, Santa Clara, CA.
14. Deyerl C, Distler T. In search of a scalable raft-based replication architecture. *PaPoC '19*. Association for Computing Machinery; 2019.



15. Wada D, Kitagawa J, Kobayashi H. Online game protocol for P2P using Byzantine agreement; 2009:753-758; IEEE.
16. Castro M, Liskov B. Practical byzantine fault tolerance. *OSDI '99*. USENIX Association; 1999:173-186.
17. Bursztein E, Hamburg M, Lagarenne J, Boneh D. Openconflict: preventing real time map hacks in online games. Proceedings of the 2011 IEEE Symposium on Security and Privacy; 2011:506-520; IEEE.
18. Yuen HY, Wu F, Cai W, Chan HC, Yan Q, Leung VC. Proof-of-play: a novel consensus model for blockchain-based peer-to-peer gaming system. *BSCI '19*. Association for Computing Machinery; 2019:19-28.

How to cite this article: Pozzan G, Vardanega T. Rafting multiplayer video games. *Softw Pract Exper*. 2022;52(4):1065-1091. doi: 10.1002/spe.3048

