Raftmsgstream: A Message Stream System Based on Raft

GIANLUCA BRESOLIN, University of Padua, Computer Science, ITA

Consensus is a fundamental property of distributed systems to ensure data consistency, and Raft is a consensus algorithm designed to be easy to understand and implement while providing fault tolerance. In this paper, we present RaftMsgStream, a message stream system based on the Raft consensus algorithm. In the messaging context, consensus ensures that messages are delivered in the same order to all nodes. The focus of this work is to provide a simple and efficient Proof-of-Concept where users can join a group, send and receive messages, and eventually leave the group, while all these operations are managed using the Raft algorithm.

CCS Concepts: • **Raft**; • **Distributed systems**; • **Fault tolerance**; • **Message Streaming**;

Additional Key Words and Phrases: Distributed systems, Raft, Message streaming, Fault tolerance

ACM Reference Format:
Gianluca Bresolin. 2025. Raftmsgstream: A Message Stream System Based on Raft. In *Technical Report for RaftMsgStream: A Message Stream System Based on Raft*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In distributed systems, the benefits of having multiple nodes, such as fault tolerance, resource pooling, and improved scalability, must be balanced against the challenges of achieving consensus among nodes when required. The challenge lies in ensuring that the nodes can agree on the system's state and continue to operate in a consistent way, even in the presence of failures, or delays, or loss of communication between the nodes.

In the past, a conceptual solutions that tried to address this problem was the Paxos algorithm [6], proposed by *Leslie Lamport*. However, Paxos provides a complex concept that only a few could fully understand. To address these difficulties and to provide a clearer and more practical solution to the needs of distributed systems, *Diego Ongaro* and *John Ousterhout* introduced in the Raft paper [9] the Raft consensus algorithm.

The real value of Raft is its simplicity and understandability, which make it a good choice for teaching and for building practical systems that require consensus.

Starting from these premises, we present in this technical report *RaftMsgStream*, a message streaming system based on the Raft algorithm. The focus of this work is to provide an implementation of the Raft consensus algorithm and to apply it in the context of a message-streaming system. Given the variety of distributed solutions in message streaming, we wanted to explore how Raft could be used to provide a consistent solution.

Hence *RaftMsgStream*, a Proof-of-Concept where the focus is to provide consistency of the actions performed by the users in the system rather than attempting to build a complete messaging application with all the features it should have.

Author's Contact Information: Gianluca Bresolin, gianbreso02@gmail.com, University of Padua, Computer Science, Padua, ITA.

Technical Report, Padua, ITA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

The primary goal of this work is to provide a simple and efficient demonstration that, by tackling the implementation of Raft, can provide a clear understanding of the algorithm and its application in a real-world context, leading to the investigation of issues and aspects that might otherwise be overlooked, even with a thorough study of the algorithm's papers.

2 Raft

The Raft consensus algorithm, as explained in the Raft paper [9], decomposes the consensus problem into three subproblems:

- Leader election: selecting by voting a leader within the cluster to manage log replication and handle client requests;
- Log replication: ensuring that the leader propagates log entries to followers, maintaining consistency across the cluster;
- Safety: guaranteeing that a committed log entry remains immutable and is present in the logs of all future leaders.

To manage these sub-problems, the nodes collocated in a cluster are divided into three roles: leader, follower and candidate. In normal operation, there is only one leader and all the other nodes are followers. The leader handles client requests while all the other nodes are passive and replicate the leader's log. The third role, candidate, is intended to elect a new leader when the current leader fails or cannot communicate with the other operational nodes.
In *Raft*, as in our implementation, nodes communicate with each other using RPCs (Remote Procedure Calls), in the asynchronous variant, to ensure transparent and efficient communication. In particular, the following RPCs are used:

- *RequestVoteRPC*: invoked by candidates to gather votes;
- *AppendEntriesRPC*: invoked by leader to replicate log entries and also to provide heartbeats;
- *InstallSnapshotRPC*: invoked by leader to send snapshot to a follower.

To enhance performance, those RPCs are in parallel.

2.1 Leader Election

Raft divides time into terms of arbitrary length enumerated with consecutive integers. Each term begins with an election, and within a single term, multiple elections (and therefore multiple candidates) can occur, but only one leader per term exists.
A candidate wins an election if it receives votes from a majority (i.e. more than half) of the cluster nodes and, when it becomes leader, it starts sending heartbeats to the other nodes to maintain its leadership. Those heartbeats are *AppendEntriesRPC* with empty entries.
Each node in the cluster has an election timeout (*electionTimer* in our implementation) that restarts every time the node receives an RPCs from the leader: if the timeout expires, the node becomes a candidate and starts a new election with a new and incremented term by sending in parallel *RequestVoteRPCs* to the other nodes.
When a candidate receives a heartbeat from a leader of the same or higher term, it reverts to follower and the election is stopped.
If multiple nodes become candidates in a short period of time, the cluster can end up in a *split vote*, where no candidate reaches the majority of votes for a term: in this situation candidates will time out and start a new election with an increasing term. However, split votes could repeat indefinitely, each causing an outage. To reduce their probability, the election timeout is randomized within a fixed interval which, as suggested by the Raft paper [9] and implemented in our system, is set between 150ms (*MinElectionTimeout*) and 300ms (*MaxElectionTimeout*).
Leader election is the aspect of Raft where timing is most critical: in order to provide steady progress,

the systems must adhere to the following timing requirement:

$$broadcastTime \ll electionTimeout \ll MTBF$$

where *broadcastTime* is the time required to send in parallel RPCs to all the other nodes and *MTBF* is the *'Mean Time Between Failures'*: those values are properties of the underlying system and only the *electionTimeout* is something that we have to choose.
At the base of all the election logic there is the concept that a node, for each term, can vote only once.

2.2 Log Replication

When a leader is elected, it begins to serve client requests and so it starts populating its log and replicating it to the other nodes using the AppendEntriesRPCs. Each client request contains a command to be executed by the replicated state machines and only when an entry is safely replicated (i.e. replicated on a majority of nodes), the leader marks the entry as committed. Since Raft is a consensus algorithm designed for replicated state machines, each node maintains its own state machine. Therefore, once a leader marks an entry as committed, it applies the entry's command to its associated state machine. When a log entry is committed, it is guaranteed that all the other nodes will have the same entry in their logs at the same index, and so they will apply the same command to their state machines in the same order. The other nodes will acknowledge that a certain entry is committed thanks to the last committed index that the leader sends in the AppendEntriesRPCs.
If a follower fails, even temporarily, or runs slowly (e.g., due to high load or network congestion, it is unable to provide a timely response but is still available) or if network packets are lost, the leader retries the AppendEntriesRPC an indefinite amount of times, even after it has responded to the client, until the follower responds with a success.
To provide consistent log replication, each log entry contains the term in which it was created and the index of the entry in the log. Those values are essential to perform the *consistency check*: when sending an AppendEntriesRPC, the leader sends the term and the index of the previous entry in the log that immediately precedes the new entry. If the follower does not have those values in the previous entry, it will reject the AppendEntriesRPC, otherwise it will append to its node the entries contained in the AppendEntriesRPC's arguments. By this inductive approach, when the AppendEntriesRPC return successfully, the leader knows that the follower's log is identical to its own log up through the new entries. As a consequence, when a leader commits an entry, all previous entries in its log are also committed, even if they were created by a previous leader.
When a leader fails, it may leave the log in an inconsistent state: to solve this problem, the following leader will handle inconsistencies by forcing the followers logs to duplicate its own (i.e. the followers logs will be overwritten). To do this, the leader previously has to find the latest log entry where the two logs agree and then send all the entries from that point to the follower: this process is enabled by the *nextIndex*, a value stored and updated by the leader for each follower that indicates the index of the next log entry to send to it. When an AppendEntriesRPC is rejected, the leader decrements the *nextIndex* and retries the AppendEntriesRPC. However, if the AppendEntriesRPC is successful, the leader will increment the *nextIndex* at the value of the last sent entry plus one.
As a consequence of this log replication mechanism, a leader will never overwrite or delete an entry in its own log.

2.3 Safety

The previous solutions for the leader election and log replication are not enough to ensure the safety of the system: to provide safety, Raft applies a restriction in the election process.
The restriction concerns which node can become a leader: a candidate can win an election only if it has the most up-to-date log. This restriction is applied by the RequestVoteRPC: when a candidate sends a RequestVoteRPC, it sends the term and the index of the last entry in its log. If the receiver has a more up-to-date log, it will reject the vote.
The more up-to-date log property is established by comparing the last log entries' terms: the log with the later term is more up-to-date, otherwise, if the terms are the same, the log with the longer wins.
This restriction ensures that the leader will have all the committed entries in its log since the start of its term: this allows avoiding the transfer of entries to the leader and guarantees a one-direction flow, from the leader to the followers.
Another issue that can affect safety is the case where the previous leader had replicated some entries but was not able to commit them, even if it had replicated them on a majority of the cluster nodes. Raft addresses the problems of uncommitted previous leaders' entries not by counting their number of replicas (as illustrated in Figure 8 of the Raft paper [9], which demonstrates why this approach is ineffective) but by considering only the replicas that have been recorded during the leader's current term. This is a consequence of the *Log Matching Property*, which guarantees that 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. In fact, this property ensures that, when a leader commits an entry from its current term, all previous entries are also indirectly committed.

2.4 Follower and Candidate Crashes

Followers and candidates failures in the Raft algorithm are handled in a very simple way: all VoteRequestRPCs and AppendEntriesRPCs will fail and so the leader will retry an indefinite amount of times, until the faulty node will eventually return.
If the faulty node resumes, it will start responding to the RPCs invoked by the leader, allowing the leader to identify the last consistent point in the node's log and send all the missing entries, enabling it to catch up with the leader's log.
If the node fails after receiving the RPCs but before responding to the leader, the leader will retry the RPCs until the node responds. Due to the idempotency of RPCs, meaning that invoking the same RPC multiple times has the same effect as invoking it once, log consistency is ensured, meaning that the same command will not appear multiple times in the log entries.

2.5 Cluster Membership Changes

The Raft algorithm, as provided by our implementation, supports cluster membership changes, such as adding or removing nodes from the cluster to face failures or to change the degree of replication. The cluster membership changes are managed without taking the entire cluster off-line. The key idea behind a safe cluster membership change is the absence of possibility that two leaders are elected for the same term during the transition. Any approach where nodes switch directly from the old configuration to the new one doesn't guarantee this property and so it is unsafe: starting from this premise, Raft provides a two-phase approach to change the cluster membership.

(1) The first phase is a transitional one and it is called *joint consensus*: in this phase, the new configuration is added to the old one. During joint consensus, the log entries are replicated by the leader to all nodes in both configurations and any node from both configurations

may serve as a leader. For both replication and election, the agreements are required from a majority of the nodes in both configurations. In practice, when the leader receives a request to change configuration from C_{old} to C_{new}, it will create a new log entry with the configuration for joint consensus called $C_{old,new}$ and then it will replicate that entry to all nodes in both configurations.

(2) Once joint consensus is committed in both configurations, the second phase starts: the leader will create a new log entry with C_{new} and it will replicate it to all nodes in $C_{old,new}$. When the entry is committed, the configuration change is completed and the nodes that are no longer in the configuration will be removed from the cluster. Even the leader, if it is not in C_{new} anymore, will step down, allowing a new leader from C_{new} to be elected.

Thanks to this transitional joint consensus phase, the cluster can continue to serve requests even while configuration changes are happening.
Moreover, to avoid outages caused by the time required from the new nodes to catch up with the leader's log (during which committing new log entries might not be possible), Raft introduces the concept of *non-voting nodes*. These nodes are part of the cluster but do not participate in the leader election process and do not count toward the majority required to reach consensus for log replication. However, they can still interact with the leader through AppendEntriesRPCs to synchronize with the current state, allowing them to join the cluster as voting nodes only when they have reached a suitable condition (i.e., they are not too far behind the current state).
As another availability issue, removed nodes that are not aware of the commitment of C_{new} will not receive any more heartbeats from the leader and so they will timeout, leading to a repeated election process that causes poor availability. To solve this problem, the nodes in Raft disregard RequestVoteRPCs when they believe the current leader is alive (i.e. they received heartbeats from it before their election timeout expired). This solution does not affect normal elections but helps to avoid disruptions from removed nodes.

2.6 Log Compaction

In Raft, to avoid indefinite growth of logs that isn't sustainable in practical systems, the logs are compacted via a snapshot mechanism.
In snapshotting, the entire current system is written to a snapshot and then the entire log, up to the that point, is discarded.
The snapshotting process can have a leader-based approach or an independent one: as discussed in the Raft paper [9], the leader-based approach presents some disadvantages, such as the waste of bandwidth since each node already has the information needed to produce its own snapshot. For this reason, in addition to simpler logic, in our implementation we decided to adopt the independent approach, where each node creates its own snapshot. Most of the snapshotting process is handled by the state machine (*msgStreamStateMachine* in our implementation), but also the *RaftNode* provides some metadata, such as the last included index and the last included term to preserve the AppendEntriesRPCs consistency check.
Other metadata stored in the snapshot include the last configuration change removed from the log by the snapshot and the last USN (*Unique Sequence Number*) recorded for each user at the moment the snapshot is taken.
The snapshotting process is triggered by the *RaftNode* when the committed entries in the log grow over a certain size: in our implementation this value is set through the *Snapshot threshold* constant.
Although snapshotting is an independent node process, the leader will need to send a snapshot with an InstallSnapshotRPC to a follower if the next log entry it has to send has already been

discarded. The node that receives the snapshot will then apply it to its state and then it will discard its entire log.

3 Technologies and Architectural Decisions

3.1 Why *Go*

The choice of *Go* as the implementation language for this project is based on several key advantages that align well with the objectives of developing an efficient but especially an understandable Raft-based system.
First, Raft is a consensus algorithm that leverages concurrency to improve its performance. As a consequence, the language used for the implementation should provide good support for that: *Go* is a language that provides a built-in and powerful concurrency model based on *goroutines*, which are lightweight M-N threads managed by the *Go* runtime. Compared to OS-level threads, *goroutines* offer minimal overhead and efficient scheduling, making them ideal for handling concurrency.
In addition, *Go* provides *channels*, a powerful communication mechanism that allows *goroutines* to communicate with each other in a safe and efficient way. This aligns with the core *Go* philosophy: *"Don't communicate by sharing memory, share memory by communicating."* [3]
As communication among the nodes in Raft is based on RPCs, the support of *Go* for them is another key advantage. It is important to note that Go also offers *gRPCs*, which provide better performance and interoperability compared to RPCs, as they are based on *Protocol Buffers*, an interface definition language that allows efficient serialization and compact data representation, while also providing support for communication among nodes that are developed using different technologies. However, since the entire project is implemented in *Go* and is intended as a Proof-of-Concept, which does not require interoperability, we have chosen to adopt RPCs to maintain the focus on understanding the Raft algorithm without introducing additional complexity. Another key advantage of *Go* is its simplicity and readability, which is particularly beneficial for a project centered around *Raft*, designed to be understandable compared to the complexity of *Paxos*. Given these key priorities and runtime advantages, *Go* stands out as the most suitable language.

3.2 Raft

The architectural decisions taken for our Raft implementation are guided by the need to provide a clear and understandable implementation of the Raft consensus algorithm.
Our consensus module *RaftNode* is designed to be a standalone module that can be easily integrated with a state machine to provide a *replicated state machine architecture*.

RaftNode and Log: The *RaftNode* module is responsible for managing the Raft algorithm, including leader election, log replication, membership changes and log compaction. Since the primary goal of this project is to provide a Proof-of-Concept for the Raft algorithm, we deliberately chose not to integrate a persistence layer in order to avoid unnecessary complexity. As a result, the *RaftNode* module directly manages the log, ensuring full control over its operations.
Another aspect in relation to log management is the decision to apply the *'Dummy log entry'* approach: this approach consists in adding a dummy log entry at the beginning of the log, with its own index and term taken from the last entries of the node's most recent snapshot (both zero if no snapshot is present), to simplify the consistency check in the *AppendEntriesRPCs*, without the need to handle special cases for the first log entry.

RPCs: Each *RaftNode* listens for incoming RPC invocations on a dedicated port. To minimize the overhead associated with establishing multiple connections on demand, we have adopted a single persistent connection model. In this approach, each node maintains a dedicated connection to every other node in the cluster, which are established at the beginning of the node's life cycle, prior to the actual execution of the *RaftNode* module. This design choice is based on the assumption that the number of nodes in our Proof-of-Concept cluster is relatively small, so the overhead of maintaining multiple connections is acceptable. In addition, this approach simplifies the implementation of RPCs, as each node can directly invoke the appropriate method on the target without having to establish a new connection each time, which would introduce additional complexity and latency.

Concurrency: The Raft algorithm is inherently concurrent. To handle this concurrency, we have designed a *goroutines*-based model that allows each node to handle concurrency. In particular, we identify the following responsibilities:

- *handleTimer()*: This *goroutine* manages the timers of the node, which has an election timer and a minimum timer (to prevent disruptions caused by removed nodes);
- *askForVotes()*: This *goroutine* is responsible for sending *RequestVoteRPCs* in parallel to other nodes when the node transitions to the candidate state;
- *handleVotes()*: This *goroutine* manages the votes received by the node. If a majority is obtained, it transitions the node to the leader state, and subsequently, it initiates another *goroutine* for leadership management;
- *handleLeadership()*: This *goroutine* manages the leader's operations, including log replication and heartbeat generation;
- *handleSnapshot()*: This *goroutine* manages the snapshotting process that will lead to the creation of a snapshot and the compaction of the log;
- *handleRaftNode()*: This *goroutine* is responsible for initializing the previous *goroutines*, managing the node if it is a non-voting one and ensuring the graceful termination of the node when necessary. This includes closing connections and channels, as well as notifying other *goroutines* to terminate.

Additionally, Go's runtime provides built-in support for concurrent RPCs, allowing the system to scale efficiently while maintaining responsiveness across the cluster.
To avoid race conditions for the internal state of *RaftNode*, each instance is protected by a dedicated mutex to ensure that only one *goroutine* can access the state at a time. This design choice, while introducing some overhead, provides a simple and effective way to manage concurrent access to the state. In addition to that, the design of our *RaftNode* module tries to minimize critical sections and leverages communication through channels while maintaining simplicity over efficiency.

Cluster membership changes: The design of our *RaftNode* module supports changes in cluster membership. The *joint consensus* phase is managed by the leader and is enabled within the *RaftNode* module by always maintaining a configuration made by the union of the old and the new configurations: while the old configuration may not always exist, the new configuration is always present.

Client interactions: Clients doesn't interact directly with Raft, however each of their requests that contain a command for the state machine of the system, before being applied to it, requires to be committed through the *RaftNode* module. In particular, the leader node will append the command to its log and replicate it to the other nodes in the cluster. If the client interacts with a server that is not the leader in the Raft cluster, it will reject the client's request and supply information about the most recent leader it has heard from. Once the command is committed, the leader will apply it to

its state machine by using dedicated channels. This approach ensures that all client requests are handled consistently across the cluster. Additionally, to ensure idempotent requests, each client associates a Unique Sequence Number (USN) with its requests. Each *RaftNode* keeps track of these values for each client by storing the last USN committed. Those values are then used to eventually discard stale requests, preventing state inconsistencies.
When a client request leads to a read of the state, our Raft design provides a read-only approach that ensures the most up-to-date state. This approach is based on the commitment of a *no-op* entry (i.e. an entry with an empty command) by the leader at the start of its term, and by checking if it has been deposed before processing the read-only request by successfully exchanging heartbeats with a majority of the cluster. All these operations are required to avoid the risk of reading stale data: in particular, the *no-op* entry, due to *Leader Completeness Property*, ensures that a leader must have the latest information on which entries are committed, while the exchanging of heartbeats excludes the election of a more recent leader.

3.3 RaftMsgStream

The architectural decisions for our RaftMsgStream system are guided by the integration of the Raft logic within a replicated state machine architecture.
The *RaftMsgStream* system is built around the *Server* component, which exposes HTTP APIs to clients and processes incoming requests through the *StateManager* component.

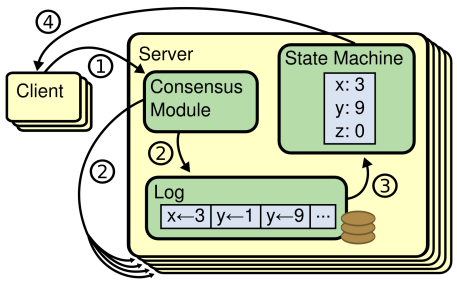

Fig. 1. Replicated state machine architecture, from the Raft paper [9]

The *StateManager* is responsible for ensuring the consistent management of the *msgStreamStateMachine* via the *RaftNode* module, maintaining transparency with the server.
The *msgStreamStateMachine* is a simple state machine that manages the state of the message streaming system, where we can find all the groups and their related users and messages that are part of the system.

The system allows clients to perform basic operations via HTTP requests. Specifically, the allowed user operations are:

- *Join a group*: this operation allows a user to join a group by providing a group name and a username. To keep the system as simple as possible, we decided to design this operation by sending an empty message command to the group;
- *Send a message*: this operation allows a user to send a message to a group by making an HTTP request to the *'/send'* API provided by *Servers*, including the group name, username and message;
- *Leave a group*: this operation allows a user to leave a group by making an HTTP request to the *'/leave'* API provided by *Servers*, including the group name and username.

In the design of the *RaftMsgStream*, we have adopted the *publish-subscribe* pattern to manage notifications between *Servers* and *Clients*. By choosing this pattern, *Servers* will notify the *Clients* in a group when a new message is received through *Server-Sent Events*, a unidirectional communication mechanism in which servers push updates to subscribed clients over HTTP. By joining a group, the client subscribes to a specified topic (in this case a group) to receive relevant notifications.

The subscription process is managed via HTTP requests to the *'/subscribe'* and *'/unsubscribe'* APIs provided by *Servers*.

Upon receiving a notification, the *Client* updates its state by making an HTTP request to the *'/update'* API offered by *Servers*, including relevant data such as the *lastMessageIndex* and the group name.
By using this design, we enable real-time notifications and enhance scalability of the system, since *Servers* do not need to know each *Client*'s state, thus improving decoupling between them.

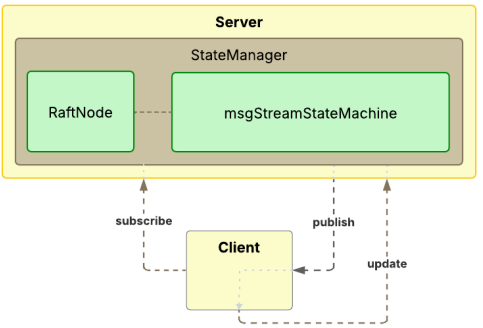

Fig. 2. Publish-Subscribe pattern in RaftMsgStream.

In addition, we identify the possibility of using *non-voting nodes* in the *RaftMsgStream* system to provide read-only requests with relaxed consistency (i.e., we lose the guarantee that the state we read is the most up-to-date). This approach allows the *non-voting nodes* to serve read-only requests without participating in the leader election process, thus reducing the load on the system and improving the horizontal scaling of the system *'by-cloning'*. This design would be particularly beneficial in read-heavy scenarios, where users primarily retrieve the system state rather than modifying it by sending messages or joining/leaving groups. Furthermore, this approach would improve the scalability of the system, enabling easier horizontal growth by adding more *non-voting nodes* to the cluster and even supporting data partitioning between users and groups.

4 Evaluations

4.1 Correctness

To evaluate the correctness of our *Raft* implementation, we have conducted a series of unit and integration tests to verify the behavior of our *RaftNode* module.
Our testing process primarily focused on the key aspects and behaviors outlined in the *Raft* paper, particularly in Figure 2 and Figure 13 [9], while deliberately omitting tests for features we identified as trivially implemented. Moreover, we have included other tests to assess the more general functionality and reliability of the *RaftNode* as a whole.

4.2 Performance

To evaluate the performance of our *RaftMsgStream* system, we performed a series of tests in relation to the number of nodes in the cluster and the number of clients. The tests were carried out through *K6*, running on *K6 Cloud*, to simulate various load scenarios, utilizing *K6*'s infrastructure to simulate virtual users. Performance tests were carried out hosting the servers on *EC2 t2.micro* instances. The results of the benchmark in Table 1 show the average time in milliseconds (ms) to send a request and receive the related response. The API tested was the *'/send'* endpoint, as we consider it the main operation that will be performed by clients and that requires consensus agreement within the cluster. We went with this approach to provide information on the latency of the system as a direct indicator of our system's efficiency.

Based on the results, we can observe how the system's performance is influenced both by the

Table 1. Average Time per Request

Nodes	10 Users	50 Users	100 Users	200 Users
3	13.27 ms	14.81 ms	17.52 ms	20.87 ms
5	13.90 ms	16.46 ms	21.03 ms	24.15 ms
10	15.32 ms	18.88 ms	24.59 ms	28.73 ms
15	16.47 ms	20.12 ms	26.41 ms	31.24 ms

number of nodes in the cluster and by the number of users. As expected, performance tends to decrease with an increasing number of nodes and users. The relations with the number of nodes in the cluster are mainly due to the additional load that the system has to handle to achieve consensus across a larger set of nodes.
In conclusion, the trade-offs between performance and fault tolerance are a crucial aspect to consider, which strictly depend on the requirements of the context in which the system will be used. Furthermore, future tests could explore clusters with *non-voting nodes* to evaluate their impact on consensus efficiency, since they allow us to reduce the leader's load and improve read operations on the state.

5 Related Works

In the context of message streaming in distributed systems, several technologies have applied the Raft consensus algorithm to provide reliable and fault-tolerant solutions. In particular, in the development of our *RaftMsgStream* system, we have drawn inspiration from *NATS Jetstream* [8] and *Etcd* [1] (a distributed key-value store, unrelated to message streaming). Those systems provide a reliable foundation that has also served as reference material during the design and implementation phase of this project. In particular, *Etcd* provides the idea of using *non-voting nodes* to improve the performance of the system.
In addition, since those systems aren't based on RPCs communication among the nodes, we have also taken inspiration from the *HashiCorp Raft* repository [4], which provides an established implementation of the algorithm in Go.

6 Conclusions

Finally, we can conclude that this project has provided a comprehensive understanding of the Raft consensus algorithm and its application in a possible real-world scenario.
As other systems provide, the Raft algorithm offers a reliable and fault-tolerant solution to manage the state of a distributed system in a consistent way.
The *RaftMsgStream* system has been designed to provide a clear and understandable implementation of the Raft algorithm while exploring the context of message streaming.
Ultimately, the RaftMsgStream system is available at *github.com/GianlucaBresolin/RaftMsgStream* where we provide a simple SPA, shown in Figure 3, which allows users to prove the basic functionalities of the system. As a Proof-of-Concept, it is primarily intended for learning and experimentation rather than production use. However, with further refinements in design and implementation, it could be effectively adapted for real-world applications.

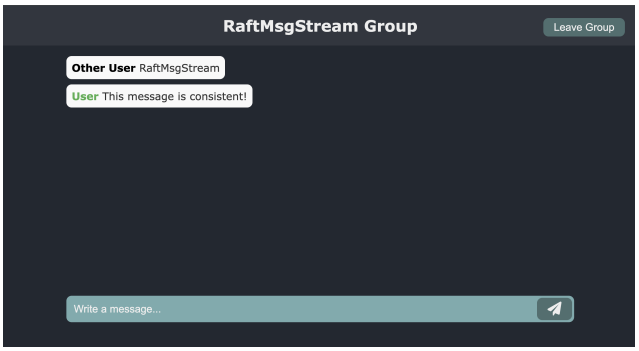

Fig. 3. Example of the SPA provided by RaftMsgStream.

References

[1] etcd.io. 2025. Etcd Clustering Guide. https://etcd.io/docs/v3.5/op-guide/clustering/. Last accessed: 5 February 2025.
[2] etcd.io. 2025. Etcd Repository. https://github.com/etcd-io/etcd. Last accessed: 5 February 2025.
[3] Andrew Gerrand. 2010. Share Memory By Communicating. https://go.dev/blog/codelab-share. Last accessed: 3 February 2025.
[4] HashiCorp. [n. d.]. Raft. https://github.com/hashicorp/raft. Last accessed: 5 February 2025.
[5] Heidi Howard. 2014. *ARC: Analysis of Raft Consensus.* Technical Report UCAM-CL-TR-857. University of Cambridge, Computer Laboratory. doi:10.48456/tr-857
[6] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998). doi:10.1145/279227.279229
[7] NATS.io. 2025. JetStream Clustering. https://docs.nats.io/running-a-nats-service/configuration/clustering/jetstream_clustering. Last accessed: 5 February 2025.
[8] NATS.io. 2025. JetStream Repository. https://github.com/nats-io/jetstream. Last accessed: 5 February 2025.
[9] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm (Extended Version). In *Proceedings of the USENIX Annual Technical Conference.* doi:10.5555/2643634.2643666

Received 7 February 2025; revised ; accepted