PROJECT REPORT

Consensus Algorithms for Fault Tolerant Systems

Raft Consensus Algorithm

By:
Paramjeetsinh Ravalji (A20502751)

INDEX

1. Introduction

Distributed systems are essential to the functioning of many applications and services in today's linked digital world, from social media networks to cloud computing platforms. Multiple linked servers, or nodes, work together in distributed systems to efficiently complete tasks and offer users dependable services. But maintaining consistency, fault tolerance, and dependability in these kinds of systems can be hard, especially when dealing with unplanned events like network splits and server failures. In distributed systems, consensus algorithms are crucial in solving these problems. Even in the face of errors or failures, these methods allow servers to come to a consensus on a shared state or course of action. Consensus algorithms help to ensure fault tolerance, data consistency, and system reliability by guaranteeing that all servers in the system converge to a consistent view of the data or state. Enabling fault-tolerant operation, which guarantees that the system may continue to operate correctly even if individual servers fail or become unreachable, is one of the main goals of consensus algorithms. Mechanisms for identifying malfunctions, choosing new coordinators or leaders, and preserving consistency between all involved servers are necessary to achieve fault tolerance.

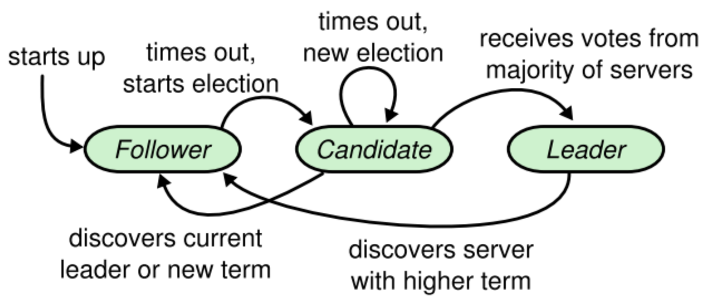
Although there are many different consensus algorithms, each with advantages and disadvantages, the Raft consensus method has become more well-known due to its ease of use, clarity, and practical use. Raft was created to offer strong fault tolerance and consistency guarantees while being more straightforward to comprehend and use than earlier consensus algorithms like Paxos. This project aims to investigate and apply the Raft consensus algorithm, with a particular focus on comprehending its fundamental concepts, design decisions, and operational features. My goal is to learn more about fault-tolerant distributed systems and make a valuable contribution to the development of scalable and dependable distributed computing technologies by thoroughly examining Raft and creating a functional implementation.

2. Background Information

Consensus algorithms are essential because they allow distributed nodes to come to a consensus on a shared state or choice even when there are errors. Maintaining data consistency, fault tolerance, and system integrity depends on all nodes in a distributed system having the same picture of the system state, which can only be achieved by consensus. The "Byzantine Generals Problem," which refers to the difficulty of reaching an agreement among a group of distributed entities (generals) that may fail or behave maliciously, is one of the basic issues with distributed consensus. To address this issue, several consensus algorithms have been put forth; each has advantages, disadvantages, and special qualities.

Introduced by Leslie Lamport in 1998, the Paxos algorithm is one of the oldest and most well-known consensus methods. Although Paxos offers a theoretical framework for reaching consensus in distributed systems, it is frequently seen as being difficult to comprehend and execute.

The Raft consensus method has been more well-known in recent years due to its ease of use, practical application, and understandability. Raft, which was put up by Diego Ongaro and John Ousterhout in 2014, attempts to offer a distributed consensus method that is easier to understand and less complicated than Paxos. Raft breaks down the consensus problem into three main subproblems, leader election, log replication, and safety characteristics. By addressing each subproblem separately, Raft makes the consensus problem easier to comprehend and use.


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

3. Problem Statement

This project focuses on comprehending, applying, and evaluating the Raft consensus method, which stands out for its aim to provide a straightforward yet reliable mechanism for achieving agreement in distributed systems. Raft's design principles prioritize simplicity and understandability, making it an appealing choice for addressing coordination challenges across distributed nodes. By studying Raft, the project aims to gain a deep understanding of its underlying mechanisms, implementation nuances, and performance characteristics. This understanding will allow to effectively apply Raft to real-world distributed system scenarios, ensuring better consistency and fault tolerance management.

The overarching goal of investigating Raft is to tackle the intricate issues surrounding consistency and fault tolerance within distributed systems. These issues often arise due to the complexities of maintaining synchronized states across multiple nodes and handling failures gracefully without compromising system integrity. By leveraging Raft's consensus algorithm, the project seeks to develop insights and strategies that enhance the system's ability to maintain consistency, recover from faults, and operate reliably under various conditions. Ultimately, this project aims to contribute to the advancement of distributed systems' resilience and performance through a thorough exploration and application of the Raft consensus method.
The current state of the art involves weakened semantics, stronger assumptions about the system, restricted functionality, special hardware support or
performance compromises.[1]

4. Related Work

There are numerous consensus algorithms, each having advantages and disadvantages. Raft, Practical Byzantine Fault Tolerance algorithm (PBFT) Paxos and ZAB (ZooKeeper Atomic Broadcast) are a few well-known algorithms. Raft is an ideal choice for this project because of its simplicity and ease of understanding, whereas Paxos is recognised for its theoretical soundness.

5. Proposed Solution

In addition to putting the Raft consensus algorithm into practice, my goal is to fully comprehend its inner workings, difficulties, and trade-offs. I will make sure the implemented solution satisfies the needs of fault-tolerant distributed systems and advances the field of distributed computing research and practice through thorough testing and assessment. The proposed solution comprises of :

- In-depth Study of the Raft Algorithm: Examine the Raft consensus algorithm in detail by reading the original research paper by John Ousterhout and Diego Ongaro, along with any additional materials and resources. Break down the algorithm into its key components, including leader election, log replication, consistency guarantees, and safety properties.

- Implementation of Raft Algorithm in C/C++: Implement the Raft algorithm in C/C++ to convert its conceptual understanding into executable code. Write code that is modular,well-documented, and covers every facet of the Raft protocol, such as fault management, state transitions, and message passing.

- Component-Level Implementation: Implement each component of the Raft algorithm individually, starting with leader election, followed by log replication, and finally consistency and safety mechanisms.

- Handling Concurrency and Networking: Deal with the issues that come with distributed systems, such as thread safety, synchronization, and concurrency management. Adequately implement data structures, synchronization primitives, and locking methods to manage concurrent access to shared resources and states.

- Error Handling and Fault Tolerance: Incorporate mechanisms for detecting and recovering from faults, such as node failures, network partitions, and message loss. Implement strategies for graceful degradation and fallback mechanisms to ensure that the system remains operational even in the presence of failures.

6. Evaluation

The effectiveness of the Raft implementation will be assessed using a range of benchmarks and tests. I will provide a thorough testing methodology to assess the accuracy, efficiency, and scalability of the Raft implementation. To validate the algorithm's behavior under various conditions, design and run a range of test scenarios, such as normal operation, leader failure, network partitions, and message loss. Analyze and quantify important performance indicators, such as fault tolerance, latency, and throughput, to evaluate the efficacy and efficiency of the Raft consensus method.

I performed a total of 9 different tests to see where the Raft consensus was the most efficient and where it wasn't.
# My tests

Here is the list of all the tests during the development of the project.

I wanted to create a function test suite that runs all the tests in one command but ran out of time for that. Instead, I chose to mark here all the results of my tests with all the commands that are used. I will be using the static logs files of the clients and some other files.

* Test 1:
  Parameters :
  > Client number : 2
  > Server number : 5

  Commands :
  > display_process x (in order to find the leader)

  > set_speed {leader} low

> display_process x (in order to find the leader)

Result :

> SUCCESS : Leader changed because of FOLLOWER's timeout.

* Test 2:

Parameters :

> Client number : 2

> Server number : 5

Commands :

> display_process x (in order to find the leader)

> start_client 1

> set_speed {leader} low

> display_process x (in order to find the leader)

Result :

> FAILURE : The leader did not change and the CANDIDATE is locked in CANDIDATE status (may be due to a condition on the logs for the election)

* Test 3:

Parameters :

> Client number : 2

> Server number : 5

Commands :

> start_client 1

> start_client 2

Result :

> SUCCESS : All the server logs contain all the entries sent by all the clients.

* Test 4:

Parameters :

> Client number : 2

> Server number : 5

Commands :

> start_client 1

> add_log_entry 1 testing a new log entry

> crash_process 1

> add_log_entry 1 testing a new log entry

Result :

> SUCCESS : the second add_log_entry was not written in the server logs

* Test 5:

Parameters :

> Client number : 2

> Server number : 5

Commands :

> start_client 1

> display_process x => follower

> crash_process {follower}

> start_client 2

Result :
 > SUCCESS : All the running servers have the correct logs but the crashed server only has the logs of client 1

* Test 6:
Parameters :
> Client number : 2

> Server number : 5

Commands :
> start_client 1

> add_log_entry 1 testing a new log entry

> crash_process 1

> add_log_entry 1 testing a new log entry while crashed

> recover_process 1

> add_log_entry 1 testing a new log entry after recovery

Result :

> SUCCESS : All the logs sent durring the crash have been written before the log after recovery.

* Test 7:

Parameters :

> Client number : 2

> Server number : 5

Commands :

> start_client 1

> display_process x to find the leader

> crash_process {leader}

> display_process x to find new leader

Result :

> FAILURE : No new LEADER stuck at status CANDIDATE

* Test 8:

Parameters :

> Client number : 2

> Server number : 5

Commands :

> display_process x to find the leader

> crash_process {leader}

> display_process x to find new leader

> start_client 1

> process_recover {old_leader}

Result :
> SUCCESS : The OLD LEADER has the same logs as all the other servers.

* Test 9:
Parameters :
> Client number : 2

> Server number : 5

Commands :
> start_client 1

> add_files_entries 1 client_commands/new_log_entries_files/new_single_log_entry.txt client_commands/new_log_entries_files/new_empty_log_entry.txt client_commands/new_log_entries_files/new_multiple_log_entries.txt =>

Result :
> SUCCESS : All the logs contains the entries of the given files.
`(Might have an error while parsing the empty one)`

* Test 8:
Parameters :

> Client number : 2

> Server number : 5

Commands :
> display_process x to find the leader

> timeout_server {any_non_leader}

Result :
> SUCCESS : The timeout server is now the `LEADER` and the found `LEADER` is now a `FOLLOWER`

* Test 8:
Parameters :
> Client number : 2

> Server number : 5

Commands :
> display_process x to find the leader

> stop_process {leader}

> display_process x to find the new leader

> start_client 1

> start_client 2

> stop_process 1

```
> add_log_entries 1 testing stopped client
```

```
> add_log_entries 2 testing stopped client
```

Result :

> SUCCESS : All the servers contains all the client's static logs and the entry of client 2, except for the old leader which has no logs at all.

`END OF TESTS`

Given the complete Raft algorithm, we can now argue more precisely that the Leader Completeness Property holds[2]

7. Result and Analysis

To provide a comprehensive result and analysis for the Raft consensus algorithm regarding fault tolerance and data prevention, we can structure it into several key areas:

7.1 Fault Tolerance

- Leader Election: Raft ensures fault tolerance through leader election. In the event of a leader failure, a new leader is elected based on a simple majority of nodes' votes.
- Log Replication: The algorithm guarantees that logs are replicated across the cluster, maintaining consistency and fault tolerance even if some nodes fail.
- Leader Crash Handling: Raft handles leader crashes by initiating a new leader election, ensuring continuity of operations without data loss or inconsistencies.

7.2 Data Prevention

- Commit Process: Raft ensures that a log entry is committed only when it has been replicated by a majority of nodes. This prevents data loss or corruption due to node failures during the commit process.
- Consistent State Machine: Raft maintains a consistent state machine across all nodes, ensuring that the same operations are applied in the same order, preventing divergent states and data inconsistencies.

7.3 Performance Analysis

- Latency: Raft typically exhibits low latency in leader election and log replication due to its efficient communication protocol and leader-centric approach.
- Throughput: The algorithm's throughput is primarily influenced by factors such as network latency, message size, and cluster size. Generally, Raft can achieve high throughput in well-configured environments.
- Scalability: Raft's scalability depends on the cluster size and network architecture. While it can scale to moderately large clusters, very large clusters may require optimizations or alternative algorithms.

7.4 Failure Scenarios

- Node Failure: Raft handles node failures by electing a new leader and ensuring log replication continues from the most up-to-date node, maintaining fault tolerance.
- Network Partition: In the presence of network partitions, Raft maintains consistency by preventing split-brain scenarios through leader election rules and quorum-based commit decisions.
- Transient Failures: Raft is resilient to transient failures, such as temporary network disruptions, by employing mechanisms like leader heartbeat and timeouts for detecting and recovering from failures.

7.5 Consistency Guarantees

- Linearizability: Raft provides linearizable consistency by ensuring that all operations appear to occur instantaneously and in a total order across the cluster.
- Durability: The algorithm guarantees durability by replicating logs and committing them based on a quorum, preventing data loss even in the face of node failures.
- Raft’s performance is similar to other consensus algorithms such as Paxos. The most important case for performance is when an established leader is replicating new log entries. Raft achieves this using the minimal number of messages (a single round-trip from the leader to half the cluster).[3]

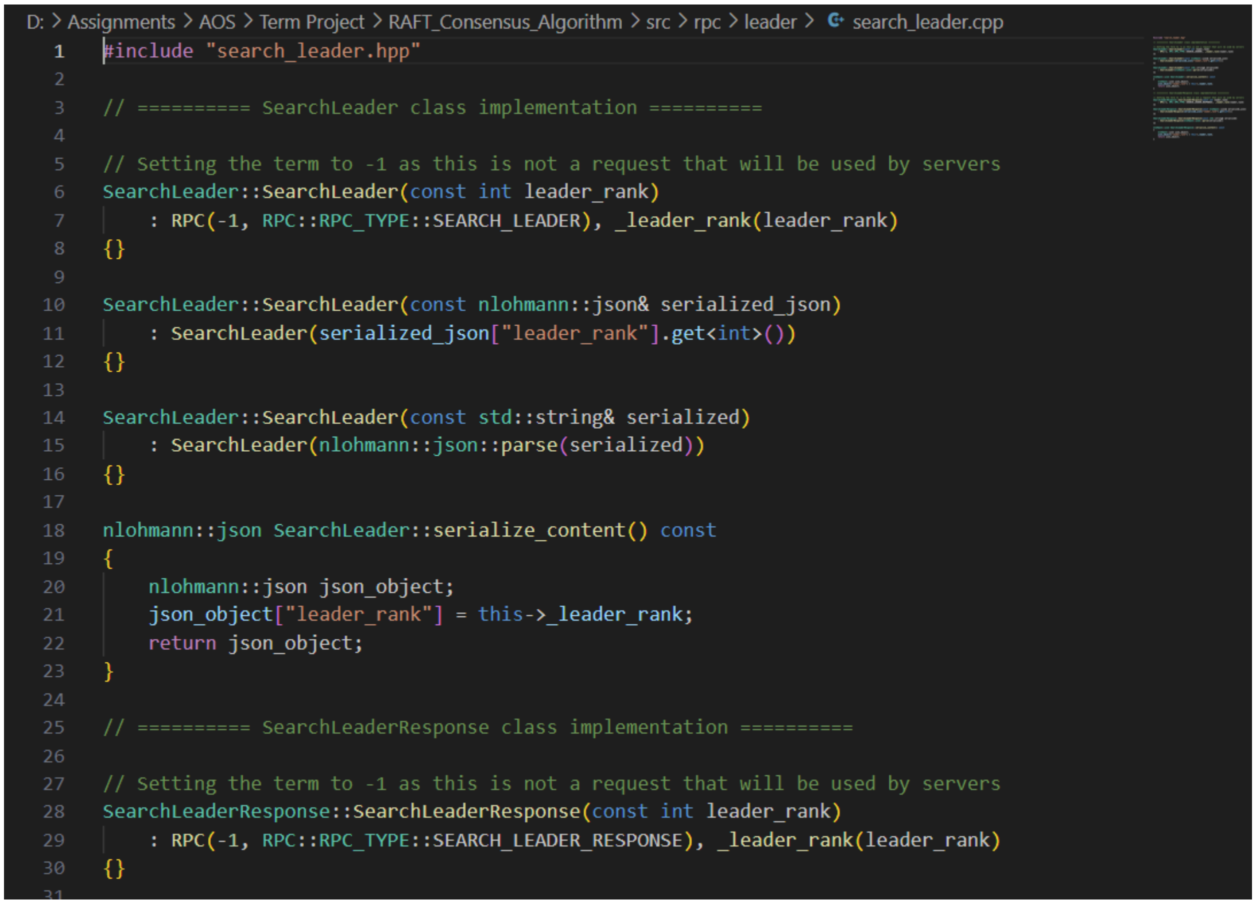
7.6 Comparison with Other Algorithms

- Paxos: Raft simplifies the consensus process compared to Paxos, making it easier to understand, implement, and maintain.
- ZooKeeper's Zab: Raft offers similar fault tolerance and consistency guarantees as Zab but with a clearer separation of leader and follower roles.

7.7 Real-world Use Cases

- Distributed Databases: Raft is commonly used in distributed databases like etcd and Consul for ensuring consistency and fault tolerance.

- Cloud Services: It's also used in cloud infrastructure services for managing distributed state and configuration.

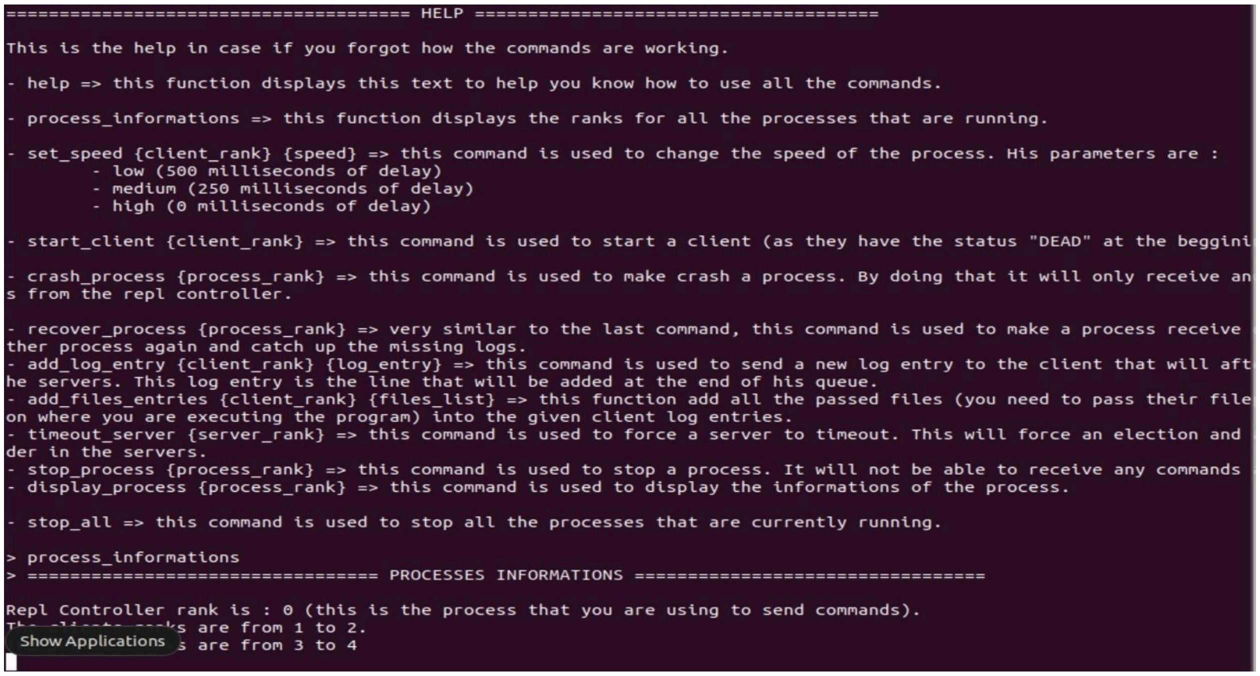

D: > Assignments > AOS > Term Project > RAFT_Consensus_Algorithm > src > rpc > leader > search_leader.cpp

```cpp
#include "search_leader.hpp"

// ========== SearchLeader class implementation ==========

// Setting the term to -1 as this is not a request that will be used by servers
SearchLeader::SearchLeader(const int leader_rank)
    : RPC(-1, RPC::RPC_TYPE::SEARCH_LEADER), _leader_rank(leader_rank)
{}

SearchLeader::SearchLeader(const nlohmann::json& serialized_json)
    : SearchLeader(serialized_json["leader_rank"].get<int>())
{}

SearchLeader::SearchLeader(const std::string& serialized)
    : SearchLeader(nlohmann::json::parse(serialized))
{}

nlohmann::json SearchLeader::serialize_content() const
{
    nlohmann::json json_object;
    json_object["leader_rank"] = this->_leader_rank;
    return json_object;
}

// ========== SearchLeaderResponse class implementation ==========

// Setting the term to -1 as this is not a request that will be used by servers
SearchLeaderResponse::SearchLeaderResponse(const int leader_rank)
    : RPC(-1, RPC::RPC_TYPE::SEARCH_LEADER_RESPONSE), _leader_rank(leader_rank)
{}
```

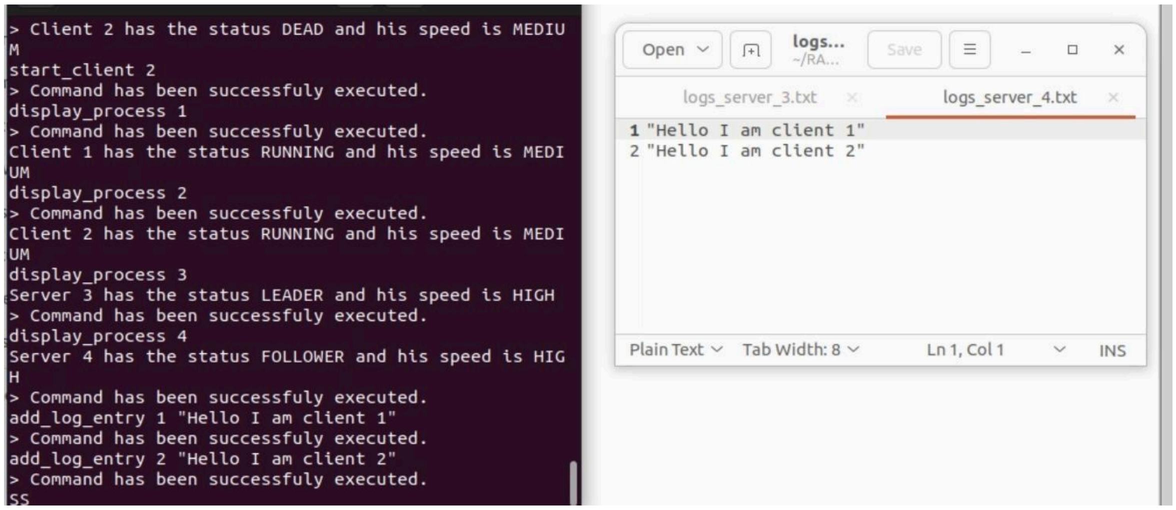
Incorporated mechanisms for detecting and recovering from faults, such as node failures, network partitions, and message loss.
Implemented strategies for graceful degradation and fallback mechanisms to ensure that the system remains operational even in the presence of failures.

The main issues I faced were when a peer initiates an election. It sends a RequestVote request to the other peers and collects their votes. If the majority of the responses are positive, the peer advances to the leader role. I was able to get the responses but the peer couldn't advance to the leader role. Code debugging and understanding RPC's in Raft helped solve it.

8. Code and Output



```
====================================== HELP ======================================

This is the help in case if you forgot how the commands are working.

- help => this function displays this text to help you know how to use all the commands.

- process_informations => this function displays the ranks for all the processes that are running.

- set_speed {client_rank} {speed} => this command is used to change the speed of the process. His parameters are :
        - low (500 milliseconds of delay)
        - medium (250 milliseconds of delay)
        - high (0 milliseconds of delay)

- start_client {client_rank} => this command is used to start a client (as they have the status "DEAD" at the beggini

- crash_process {process_rank} => this command is used to make crash a process. By doing that it will only receive an
s from the repl controller.

- recover_process {process_rank} => very similar to the last command, this command is used to make a process receive
ther process again and catch up the missing logs.
- add_log_entry {client_rank} {log_entry} => this command is used to send a new log entry to the client that will aft
he servers. This log entry is the line that will be added at the end of his queue.
- add_files_entries {client_rank} {files_list} => this function add all the passed files (you need to pass their file
on where you are executing the program) into the given client log entries.
- timeout_server {server_rank} => this command is used to force a server to timeout. This will force an election and
der in the servers.
- stop_process {process_rank} => this command is used to stop a process. It will not be able to receive any commands
- display_process {process_rank} => this command is used to display the informations of the process.

- stop_all => this command is used to stop all the processes that are currently running.

> process_informations
> ================================= PROCESSES INFORMATIONS =================================

Repl Controller rank is : 0 (this is the process that you are using to send commands).
The clients ranks are from 1 to 2.
Show Applications s are from 3 to 4
```



```
> Client 2 has the status DEAD and his speed is MEDIU
M
start_client 2
> Command has been successfuly executed.
display_process 1
> Command has been successfuly executed.
Client 1 has the status RUNNING and his speed is MEDI
UM
display_process 2
> Command has been successfuly executed.
Client 2 has the status RUNNING and his speed is MEDI
UM
display_process 3
Server 3 has the status LEADER and his speed is HIGH
> Command has been successfuly executed.
display_process 4
Server 4 has the status FOLLOWER and his speed is HIG
H
> Command has been successfuly executed.
add_log_entry 1 "Hello I am client 1"
> Command has been successfuly executed.
add_log_entry 2 "Hello I am client 2"
> Command has been successfuly executed.
SS
```

```
1 "Hello I am client 1"
2 "Hello I am client 2"
```

Github link to Project Code in C++
https://github.com/pravalji/CS_550-Project.git

9. Conclusion

In Conclusion, the proposed project on Consensus Algorithms for Fault-Tolerant Systems is an important step in improving distributed systems' resilience and dependability. I want to use the Raft Consensus technique to investigate and design strong consensus mechanisms that can withstand faults and failures in distributed contexts, hence guaranteeing system stability and uninterrupted operations.
Algorithms are often designed with correctness, efficiency, and/or conciseness as the primary goals. Although these are all worthy goals, we believe that understandability is just as important[4]

The Raft employs RPC(remote procedure calls) to request votes and sync up the cluster(using AppendEntries). So, the load of the calls does not fall on the leader node in the cluster.
Any node in the cluster can become the leader. So, it has a certain degree of fairness. Raft is strictly single Leader protocol. Too much traffic can choke the system. Some variants of Paxos algorithm exist that address this bottleneck. There are a lot of assumptions considered to be acting, like non-occurrence of Byzantine failures, which sort of reduces the real life applicability.

My goal is to make a significant contribution to the field of fault-tolerant systems by means of thorough study, experimentation, and implementation of Raft Consensus method. This will help enterprises create and manage highly dependable distributed architectures. This research has the potential to significantly improve consensus algorithm state-of-the-art, stimulating creativity and propelling advancement in the field of distributed computing.

References

[1] Bolosky, William J., Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. "Paxos replicated state machines as the basis of a {High-Performance} data store." In 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). 2011.
[2]Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14). USENIX Association, USA, 305–320.
[3]Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: have we reached consensus on distributed consensus? In Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20). Association for Computing Machinery, New York, NY, USA, Article 8, 1–9.
[4]Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14). USENIX Association, USA, 305–320.