# Implementation of the raft algorithm in a simulated seismic sensor network

Maximizing data availability and consistency, in a fault-prone network

Angeli Jacopo, Alberto Angeloni

*Department of Mathematics*

*University of Padua, Italy*

*e-mail: {jacopo.angeli, alberto.angeloni}@studenti.unipd.com*

**Abstract - This technical report details the implementation of the Raft algorithm within a simulated seismic sensor network, with the aim of improving data availability and consistency amidst network faults. The report focuses on the practical aspects of implementing the Raft algorithm in Ada programming language, offering insights into the design choices, data structures, and communication mechanisms employed.**

**The report provides the specifics of implementing Raft within the context of the simulated seismic sensor network, discussing the intricacies of adapting the algorithm to Ada's programming paradigm.**

**Key aspects of the implementation, such as message passing, state management, log replication and fault tolerance mechanisms, are described in detail. The report highlights the challenges encountered during the implementation process and the strategies employed to address them, providing valuable insights for developers seeking to implement Raft in similar environments.**

**Keywords: Raft algorithm, Distributed consensus, Seismic sensor network, Ada programming language, Fault tolerance, Technical implementation.**

## *I Problem statement*

Aspects to be investigated, boundary of the study, and goals of investigation and experiments

The deployment of sensor networks in real-world environments, particularly in remote or challenging locations, presents formidable obstacles. Chief among these is the task of ensuring the consistent reliability and availability of gathered data despite network disruptions and unpredictable operating conditions. An illustrative scenario involves the planned establishment of a seismic sensor network on the southern polar region of the lunar surface. Given the logistical complexities associated with managing such networks on-site and the critical need for precise data collection, there is a pressing demand for resilient solutions to confront these obstacles.

The primary aim of this experiment is to delve into the feasibility and efficacy of incorporating the Raft algorithm within a simulated seismic sensor network operating amidst a fault-prone environment. This entails a thorough examination of various aspects such as message transmission, state management, log replication, and mechanisms for fault tolerance. Throughout the implementation process, careful consideration is given to every design decision, with a view to elucidating its impact on the overall system.

The overarching objectives are twofold: firstly, to gauge whether the adoption of the Raft algorithm can bolster data availability, and secondly, to evaluate its proficiency in ensuring robust data replication within the simulated network. By mimicking a cluster of seismic sensors characterized by unpredictable operational behaviors, the aim is to ascertain the suitability of the Raft algorithm in mitigating the repercussions of network faults and guaranteeing the reliability of data transmission and storage.

This application seeks to explore the potential efficacy of the Raft algorithm in facing the challenges that operating a seismic sensor network within a fault-prone environment provides. Specifically, it aims to:

1. Simulate a cluster of seismic sensors within an environment with network faults using the Ada programming language.
2. Implement the Raft algorithm within the simulated network to facilitate consensus building and mechanisms for fault tolerance.
3. Establish a baseline for assessing the performance of the Raft algorithm in terms of data availability and replication across various fault scenarios.
4. Evaluate the suitability of the Raft algorithm in fortifying the reliability and resilience of data transmission and storage within the simulated seismic sensor network.

It is important to note that this study primarily focuses on the implementation and evaluation of the Raft algorithm within the context of a simulated seismic sensor network. The scope is confined to examining the algorithm's performance in achieving consensus, fault tolerance, and data replication within the simulated environment. As such, the study does not delve into the physical deployment or hardware aspects of the sensor network. Furthermore, while the Ada programming language is employed for implementation purposes, the study does not undertake a comparative analysis of the Raft algorithm with other consensus algorithms but rather aims to provide a solution to the aforementioned challenges.

## *II Experimental implementation*

### *II.I Synopsis*

Our project focuses on the development of an Ada application aimed at simulating the behavior of a network of servers for testing the Raft algorithm in a distributed system context. In this simulation, each node within the network operates in parallel, facilitating coordination and data sharing with other nodes through message passing facilitated by a shared queue structure.

The focus of the application is the simulation of a network of servers running with the Raft algorithm in a distributed system context. In this simulation, each node within the network operates in parallel, facilitating coordination and data sharing with other nodes through message passing facilitated by a shared queue structure.

### *II.II Technologies*

To develop our application, we opted for Ada, a programming language developed in the late 1970s for the United States Department of Defense. Ada stands out for its unique blend of programming paradigms, making it an intriguing choice for simulating the complex entities within our application. Its real-time and concurrent capabilities align well with the nature of the entities being simulated, ensuring efficient handling of concurrent tasks and timely response to real-world events.

The development process of the application heavily relied on real-time synchronization both between multiple parallel running entities and within the entities themselves. The nodes of the network operate in parallel, adhering to a precise lifecycle, and respond to external stimuli and time constraints. The choice of technology was heavily influenced by the libraries and features offered by different programming languages.

Ultimately, the decision to use Ada was driven by two key factors:

1. Ada Task Types: Ada task types serve as a fundamental concept in the application's design. Analogous to Java's Thread subclasses, Ada task types allow for both declared and dynamically allocated instances. Task types can be parameterized, offering flexibility similar to passing arguments to a Java constructor. Tasks instantiated from a task type exhibit consistent behavior, initiated within their declaring scope, and persist until termination. This feature was extremely useful in the concurrent management of Node tasks.
2. Real-Time Library: Ada provides a comprehensive Real_Time library tailored for real-time applications. This standard library offers essential procedures for time operations, catering to the precise requirements of real-time programming. With access to an absolute clock and support for handling time spans, the Real_Time

library enhances precision and reliability, crucial for developing robust real-time systems.

## *II.III Design aspects*

### *Overview*

For the scope of the application we identified three main objects to represent the real-world scenario of choice: the nodes of the network, the mechanism and the packet used in the interaction between nodes.

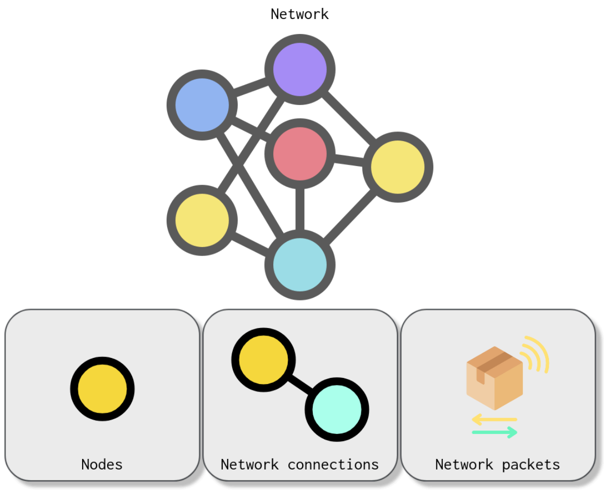

**Fig 1**: Object represented in our application.

To capture the essence of network functionality as accurately as possible, we chose to design the nodes of the network using the Ada task type. The interconnection between the network entities with a shared memory data structure, and the messages exchanged by the nodes using the network using a record type. In the following sections, we dive into the specific design details for each component.

### *Nodes*

In order to accurately simulate the network's functionality, we've crafted each node in the application as a task. Every node contains a state composed of essential data for executing the Raft algorithm and in addition, the state contains supplementary information crucial for simulation purposes and for generating specific values that encapsulate the application's context.

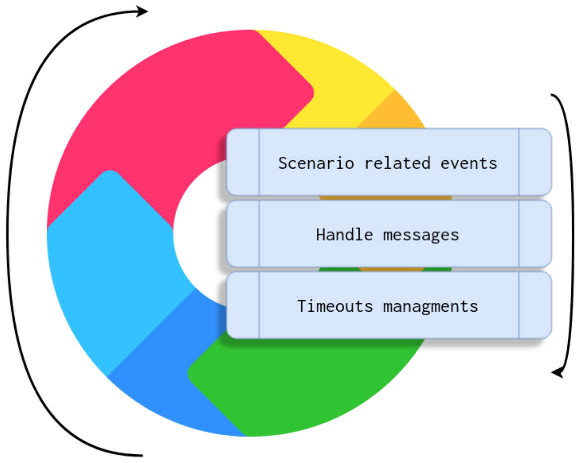

**Fig 2**: Node periodic and sequential lifecycle: scenario related events, handling of messages and timeout management.

A node lifecycle is made of three different processes:

***Scenario related events***

The first phase is dedicated to the simulation of real-world related to the context events, in the scenario the application is depicting every node can fault and resume at a random time (fault-prone aspect) or it can sense some data (seismic sensor aspect).

In our modeling, we simulate two main events: the generation of earthquakes and the random crash of a node. To accomplish this, we employ two specific procedures: CrashSimulator and QuakeSimulation. Every two seconds, these procedures follow a similar pattern: they calculate the probability of their respective events occurring using the probability density function of a Poisson distribution. Subsequently, they compare this calculated probability with a randomly generated number. If the generated number exceeds the calculated probability, the event is managed based on the current state of the node. This approach allows us to dynamically simulate the occurrence of earthquakes and node crashes within our system model.

Moreover, CrashSimulator provides the mechanisms to the system to resume a node that crashed, according to the specification and the behavior of Raft resume phase, clearing the messages queue to simulate the non receiving messages during the down time ad resetting its state with type equal to Follower and fresh timestamp.

### *Message handling*

Message handling within the node is a critical aspect of the Raft algorithm implementation. Each node is responsible for processing every message present in its own queue, provided the queue is not empty. Further details regarding the workflow of the program and the design of the messageHandler function will be elaborated later in the document, providing a more specific explanation of the message handling process.

### *Timeouts management*

In a Raft algorithm implementation, timeout management is crucial for system stability. Leaders send heartbeat messages to prevent synchronization issues (every between 50 and 100 milliseconds), while Followers transition to Candidates if they don't hear from the Leader within a time span between 150 to 300 milliseconds.

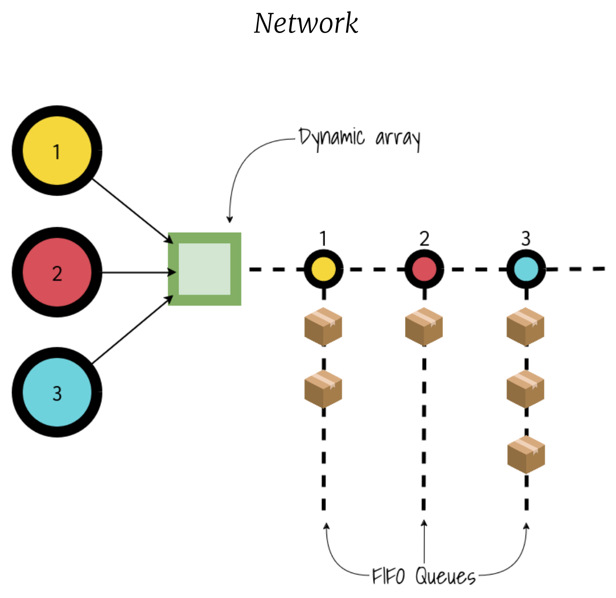

**Fig 3**: To represent the network we used a dynamic array of FIFO queues.The image is an example with 3 nodes.

To replicate the network environment where nodes communicate by exchanging packets, we've employed a shared memory methodology. We used a vector of queues, where each queue is dedicated to a specific node within the network.

Every node is allocated with a distinct integer identifier that increases monotonically. This identifier serves as a reference point for accessing the corresponding queue within the vector. Thus, a node can seamlessly interact with its designated queue by simply referencing its unique identifier.

In the application setup, each node possesses the capability to enqueue elements into the queue associated with any other node. However, when it comes to dequeuing elements, a node can only operate on its own queue. This ensures a controlled and secure data exchange process within the network simulation.

*Messages*

Within the context of the Raft algorithm, nodes can receive four types of packets. Leader nodes send AppendEntry messages to every other node in the network and wait for responses of type AppendEntryResponse, indicating successful replication or failure. If a follower does not receive any AppendEntry messages before the CandidationTimeout deadline, it transits to the Candidate state and sends RequestVote messages to every other node, seeking their votes for leadership. The handling of these messages is dependent on the type of node (Leader, Follower, or Candidate). We designed a message hierarchy as follows.

```
type Message is abstract tagged null record;

type AppendEntry is new Message with
record
        --  Leader’s term
        Term : Integer;

        --  So follower can redirect clients
        LeaderId : Integer;

        --  Index of log entry immediately
            preceding new ones
        PrevLogIndex : Integer;

        --  Term of prevLogIndex entry
        PrevLogTerm : Integer;

        --  Log entries to store (empty for
            heartbeat; may send more than one
            for efficiency)
        LogEntries : LogEntryVector.Vector;

        --  Leader’s commitIndex
        LeaderCommit : Integer;
end record;

type AppendEntryResponse is new Message with
record
        --  CurrentTerm, for leader to update
            itself
        Term : Integer;

        --  To manage NextIndex Vector
        Sender : Integer;

        --  True if follower contained entry
            matching prevLogIndex and
            prevLogTerm
        Success : Boolean;
end record;

type RequestVote is new Message with record
        -- Candidate’s term
        Term : Integer;

        -- Candidate requesting vote
        CandidateId : Integer;

        -- Index of candidate’s last log entry
        LastLogIndex : Integer;

        -- Term of candidate’s last log entry
        LastLogTerm : Integer;
end record;

type RequestVoteResponse is new Message with
record
        -- currentTerm, for candidate to
           update itself
        Term : Integer;

        -- true means candidate received vote
        VoteGranted : Boolean;
end record;
```

The design closely adheres to the principles outlined in 'In Search of an Understandable Consensus Algorithm' by Diego Ongaro and John Ousterhout [1], with adjustments made to accommodate the transition between Remote Procedure Call (RPC), where actual returns are expected, and network request-response context where procedure returns are in form of network packages.

## *Core processes*

### ***Leader election***

As stated before, the application focus is on the implementation of the Raft algorithm. One of the core concepts of the algorithm is the leader election. At any moment the cluster of nodes has to have a single Leader. In the application there are XXX different process in which an election happen and in which a new Leader emerges:

- Start-up: At start-up, each node is initialized as a Follower with a randomized election timeout. When the election timeout of a node expires, it transits to the Candidate state and initiates a vote request by broadcasting it to other nodes in the cluster. Upon receiving positive responses from a majority of nodes, the Candidate becomes the Leader for the current term, initiating the heartbeat mechanism to maintain its leadership status and coordinate the cluster's activities.
- Heartbeat mechanism fails: If the current Leader fails to send the heartbeat message within the expected timeframe, the Raft algorithm initiates a new leader election process. The process is the same as the case above.

### ***Log replication***

The most time-consuming aspect of development was dedicated to log replication. Following the specifications outlined in the original paper, when a Leader receives a client request, the operational flow unfolds as follows:

- The Leader appends the entry with the payload of the client request to its own log.
- It then broadcasts an append entry request to all nodes on the network and tracks their log indices.
- Upon synchronization with the majority of nodes, the Leader appends the entry to its database and advances the commit index accordingly.
- However, if some nodes are not synchronized, the Leader sends the last portion of its log in ascending order until the asynchronous node is updated. This process persists within its timeout management framework, even after the Leader has responded to the client.

### ***Client request***

The follower nodes constantly monitor the environment for significant events, such as seismic activity. When a follower detects an earthquake, it initiates a process to contact the leader through an AppendEntry request. This request contains specific details of the detected event. Upon receiving the AppendEntry, the leader triggers the log replication process, which involves updating and distributing the data related to the event to all nodes in the system via broadcast. This process ensures data consistency between the leader and followers, enabling the system to react uniformly and efficiently to relevant events.

In case that the leader itself detects seismic activity, it will proceed by inserting an AppendEntry message into its queue, with a lower or equal term, making the event recognizable, and handy separately from the follower and candidate behavior.

The enqueued AppendEntry then, would be handled in respect of Raft algorithm principles explained in section Log replication.

### ***Log Activity***

We designed each node to conduct three distinct logging activities, documenting a series of information in dedicated log files. The application has been designed to facilitate observation of the functionality of every instantiated node. In a reactive manner, each node generates and maintains the "Node_<id>.log" file to capture intricate details regarding its activities. Additionally and passively, the nodes write during the execution in "State_Node_<id>.log" and "DB_Node_<id>.log", the state value and the local database content, providing a comprehensive snapshot of its operational status.

*Design of evaluation experiments*

The application architecture incorporates a configuration file, allowing for the customization of various parameters pertaining to node failure, data logging, and timeouts. Testing activity has been conducted by systematically adjusting these parameters and comparing the database contents across nodes, checking robustness and reliability in diverse operational scenarios.

```
package Config is
        ------------------------ FAILURE
        --  Node Failure Rate per 10 seconds
        NodeFR  : Float := 1.0;

        --  Average Node resume times
        NodeART : Time_Span := Seconds (3);

        -- Network Failure Rate
        NetFR   : Float := 0.0;

        ------------------------ LOGGER
        --  Append_File(Append) or Out_File
        -- (Rewrite)
        DBLogFileType : File_Mode := Out_File;

        ------------------------ TIMEOUTS
        --  Min Election Timeout Duration In
        -- Millisecond
        MinETD : Integer := 300;

        --  Max Election Timeout Duration In
                   --Millisecond
        MaxETD : Integer := 600;

        --  Min Heartbeat Timeout Duration In
                   --Millisecond
        MinHTD : Integer := 150;

        --  Max Heartbeat Timeout  Duration In
                   --Millisecond
        MaxHTD : Integer := 300;

end Config;
```

*Results of evaluation experiments*

The application's functionality, particularly concerning Log Replication, reveals a significant bug. Strikingly, this flaw seems to be non-existent in the absence of external factors such as server crashes or network failures. Upon closer examination, it becomes apparent that the duration of a node's crash state is intricately linked to an escalating probability of multiple nodes becoming simultaneously unreachable. This phenomenon, in combination with prolonged node downtime, leads to coordination challenges within the system, thus impeding seamless log replication processes. Ultimately, despite encountering challenges, the Raft algorithm demonstrates its effectiveness within the application under specific conditions. While acknowledging the existence of issues, it's essential to recognize that the application still serves as a pertinent example of the algorithm's application. Furthermore, it stands as a robust foundation for the development of more specialized and stable applications tailored to address the intricacies of the problem domain. This acknowledgment underscores the significance of iterative refinement and continuous improvement in software development, leveraging the lessons learned from current implementations to inform and enhance future iterations, thereby advancing the efficacy and relevance of the solution.

## *III Self-assessment*

### *III.I Challenges faced*

*Technology learning*

Initially, the obstacle encountered was mastering the learning curve of the Ada programming language, which, once properly controlled, allowed us to develop the desired functionalities.

Other examples of challenges we had to face included detailed design activities and subsequent coding, where even a minor error in the code could result in failures during execution, failures that might have been difficult to trace back to such errors.

### *Synchronization*

Given our strict adherence to the original paper on the Raft algorithm, one of the most challenging adaptations we faced was synchronizing nodes. In the paper, nodes were treated as separate entities within the same application, enabling direct calls to each other's public functions. However, we opted for a different approach, configuring each node as a server. Consequently, the Remote Procedure Calls (RPC) utilized in the theoretical implementation were transformed into network packets within our system. This adjustment was pivotal in ensuring seamless communication and coordination among nodes in our implementation.

The change in methodology introduced synchronization challenges, notably evident in the implementation of the AppendEntryRPC. In the formal specification, the Leader directly invokes the AppendEntryRPC method of other nodes, processes the results, and, if certain conditions are met, responds to the client in a procedural manner. However, our approach diverged from this model. When a client request arrives at the Leader from other nodes, instead of directly invoking AppendEntryRPC on individual nodes, the Leader broadcasts the AppendEntry request. Subsequently, it iterates through its message queue, managing incoming messages and handling its own timers. Only upon receiving positive responses from a majority of the nodes does the Leader respond to the client. Additionally, during the handling of incoming messages, the Leader also dispatches synchronization packets to non-updated nodes, further enhancing the synchronization mechanism.

### *Simulation aspects*

One significant challenge we faced lies in the adaptation of existing algorithms to suit different contexts. In our case, we tackled the implementation of the Raft algorithm within a distinct framework, necessitating adjustments to its leader election and log replication mechanisms. Additionally, while striving for completeness, constraints in time and resources led to the exclusion of log compaction from our implementation, showcasing the pragmatic decisions inherent in development.

## *III.II Critique of own exam product*

### *Scalability and adaptability*

The scalability and adaptability features of our system can be partially expressed by the ease with which new nodes can be added to the cluster without compromising the system's functionalities.

To add a new node to the network, it is sufficient to dynamically create a queue with its corresponding pointer and add it to the vector containing all the queue pointers. Then, the actual Node entity is instantiated, providing it with the above mentioned vector and an increasing integer ID.

The modification introduced in the execution flow to simulate seismic events does not affect the scalability of the system, as it does not introduce changes in the operation and application model of the Raft algorithm.

Some improvement that could potentially be implemented in a new version of the system are:

- Enable the addition of nodes during runtime to simulate the process of installation of a new seismic sensor server.
- Including a UI to manage the node status, giving the opportunity to introduce manually specific errors.

### *Improvement on real-world relates aspects*

To enhance the realism of earthquake simulations, it's essential to improve the algorithms used for generating seismic events. This can be achieved by incorporating historical seismic data, geological factors, and plate tectonic movements into the simulation models. By utilizing historical seismicity data, we can better model the frequency, intensity, and spatial distribution of earthquakes within the target region. Additionally, accounting for local geological features such as fault types and building structures can improve the fidelity of earthquake simulations. Through these enhancements, the simulation can provide a more accurate representation of earthquake occurrences, aiding in disaster preparedness and risk assessment.

In order to better reflect the complexities of real-world scenarios, it's important to introduce a wider range of parameters into the simulation framework. These parameters should encompass various factors that contribute to the challenges faced during emergency situations following an earthquake. This includes parameters related to building resilience, infrastructure quality, emergency response capabilities, and socio-economic factors. By incorporating such parameters, the simulation can better capture the multifaceted nature of disaster management, enabling more comprehensive risk analysis and response planning.

Incorporating the simulation of node energy consumption is crucial for assessing the impact of earthquakes on energy infrastructure and resilience. This involves modeling the energy consumption patterns of different types of nodes within the simulation environment, such as residential, commercial, and critical infrastructure nodes. By simulating pre- and post-earthquake energy demand, including heating/cooling, lighting, communication systems, and emergency response activities, the simulation can provide insights into the energy requirements and vulnerabilities of affected areas. Understanding the dynamics of energy consumption can inform strategies for improving energy resilience and recovery efforts in the aftermath of earthquakes.

These enhancements aim to provide a more comprehensive and realistic representation of earthquake scenarios, enabling better decision-making and preparedness for mitigating the impact of seismic events on communities and infrastructure.

### *III.III Learning outcomes*

Studying ADA has provided a comprehensive understanding of its principles and applications in software engineering. ADA's focus on safety-critical systems and its strong typing discipline have been particularly insightful. By learning ADA, I have gained expertise in developing robust and reliable software solutions, especially in domains where safety and security are paramount.

Acquiring knowledge of the Raft consensus algorithm has enriched my understanding of distributed systems and fault tolerance mechanisms. Raft's simplicity and ease of understanding make it a valuable tool for designing resilient distributed systems. By studying Raft, we have gained insights into consensus protocols and their role in ensuring the consistency and reliability of distributed data stores and services.

Mastering concurrent and distributed programming techniques has equipped us with the skills to design and implement scalable and efficient software systems. Understanding concepts such as parallelism, synchronization, and message passing has been instrumental in developing high-performance and fault-tolerant applications. Moreover, proficiency in concurrent and distributed programming opens up opportunities to tackle complex computational problems and leverage the full potential of modern computing architectures.

The knowledge gained by this experience presents opportunities to apply these skills in innovative application contexts. From developing real-time embedded systems to designing cloud-native distributed applications, the versatility of these concepts enables their application in a wide range of domains. Whether it's creating resilient IoT networks or implementing scalable web services, the foundational understanding acquired in ADA, Raft, and concurrent/distributed programming provides a solid framework for addressing emerging challenges in the rapidly evolving landscape of technology.

The expertise acquired during this project opens up avenues for conducting intriguing experiments and tests. Whether it's evaluating the performance and scalability of distributed algorithms, analyzing the fault tolerance properties of consensus protocols, or benchmarking the efficiency of concurrent programming constructs, there are ample opportunities to explore and experiment with these concepts. By conducting rigorous tests and experiments, valuable insights can be gained into the behavior and characteristics of complex distributed systems, contributing to the advancement of knowledge in the field of computer science and engineering.

## *VI References*

- [1] In search of an understandable consensus algorithm