

DISTRIBUTED SYSTEMS REPORT

Raft study and implementation

Andrea Bedei

`andrea.bedei2@studio.unibo.it`

Fabio Notaro

`fabio.notaro2@studio.unibo.it`

Giacomo Leo Bertuccioli

`giacomo.bertuccioli@studio.unibo.it`

December 19, 2023

The purpose of this report is to introduce and explain our proposed project regarding the Raft consensus algorithm.

In particular, our project has required a preliminary phase of studying and understanding the recommended paper[1].

The project then involved an implementation of the consensus protocol in Javascript language and then it has been used as the basis of a web application for online auction management, so as to identify and report any advantages and disadvantages of the studied algorithm.

1 Goals/requirements

The goals to be achieved by this project included:

- opportunity to reason in depth and try to solve the concurrency, security, and replication issues raised during the course by identifying the most suitable approach to the consensus algorithm under consideration
- depth understanding of the structure and functionality of consensus protocols
- create truly distributed network infrastructure with 5 physical servers (with different capabilities and heterogeneous systems) and several clients → alternatively, in case of high difficulty in pursuing this goal, the deployment will be managed with Docker.

Instead, as far as system requirements are concerned, it is good to mention:

- architectural requirements → the system will be based on a client-server architecture with communication via HTTP protocol
- database requirements → the web application for online auctions will not be based on a simple key-value store but will include a realistic relational database, composed of a set of interrelated tables, among which will surely be needed tables for users, auctions, bids... and in general everything needed for the functionalities expressed in the following point
- web application functionalities →
 1. new user registration
 2. new auction creation
 3. closing auction
 4. display list of open and closed auctions
 5. display auction details (details of the item for sale, last bids)
 6. possibility of bidding on an auction.

From the features listed above, it is easy to derive the use case diagram:

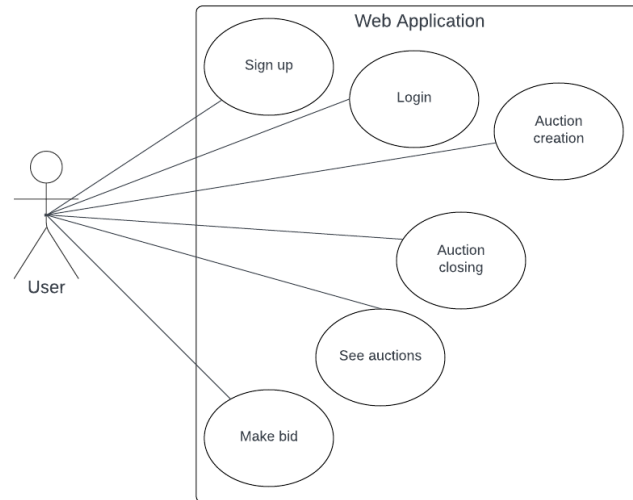


Figure 1: as also noted in the diagram, the simplicity of the web application makes it possible to ensure a comprehensive and realistic service while offering a small number of features.

1.1 Scenarios

To analyze the usage scenarios, we consider the following aspects:

- overview and purpose of the system → as mentioned above, the main functionality of the developed distributed system is to make available a complete online auction service using Raft as a consensus protocol for transactions on the database
- identification of system users → anyone interested in selling items via auction or participating in online auctions is a potential user of the system → there are no roles with different privileges or responsibilities
- usage scenario → typical usage scenario is an already registered user who occasionally accesses the web application to browse and participate in active auctions or to create an auction himself in an attempt to sell some object
- users' motivations → users are encouraged in using the system by the combination of simplicity and security (also guaranteed by the Raft algorithm used) that has always been central to the development of the project, as well as by the attention that has been paid to all aspects of distributed systems addressed in class and that emerged during development, described more accurately in the sections that follow → users are also motivated to use the created service because the reliability and fault tolerance achieved guarantee an acceptable level of service continuity
- mode of interaction → to take advantage of the services offered by the developed system, it is sufficient to reach the site <http://asteraft.ydns.eu> via any browser.

1.2 Self-assessment policy

The overall quality of the software produced was evaluated using metrics strongly related to the central concepts that emerged during the course, for which significant efforts were expended to achieve acceptable levels of quality (not only in view of the time available but also in view of the purposes of this project).

The quality of the software in its complexity was therefore evaluated in relation to the following metrics:

- usability, simplicity, and clarity of written code → although it may be trivial or obvious, the group identified early on as synonymous with quality the writing of modular code that is easy to maintain (thanks to the decomposition into interrelated components) and understandable (thanks to comprehensive comments and documentation)
- reliability → measured in terms of fault tolerance and automatic recovery following failure
- acceptable performance → ensured through efforts made with regard to scalability, response time, and reasonable use of available resources
- security → guaranteed not only by the Raft protocol but also by some standard arrangements in web application design and development, such as authentication and access authorization
- correctness and testability → certified by the test coverage achieved and their ease of implementation.

On the other hand, as far as the effectiveness of the project is concerned, it was evaluated in relation to compliance and fulfillment of the requirements and functionalities agreed upon with the teachers during the submission of the project proposal.

2 Background

In general, since the project focuses on the actual implementation of a consensus algorithm, almost all the concepts regarding distributed systems that emerged in class are useful in understanding the motivations, technical choices, and the project itself.

With regard to the characteristics of the Raft protocol, however, we recommend reading the paper [1] on which the group relied to develop the project; however, a brief summary is given below that highlights its main aspects, which are certainly useful for understanding the sections that follow.

2.1 Background on the Raft protocol

The consensus protocol is a fundamental component in distributed systems as it is responsible for ensuring data consistency among distributed nodes.

Among the various consensus protocols, Raft stands out for its simplicity and conceptual clarity.

In fact, its operation is essentially based on the concepts of node roles and remote procedure calls (RPCs).

The operation focuses specifically on the presence of a leader, who guides the other nodes through the consensus process.

The consensus is achieved through a majority voting and election process, following which one node becomes the leader, effectively deciding and propagating the status of its registry through replication.

The protocol consists mainly of three phases:

- election of leader → initially all nodes start from the follower state, but when a node detects the leader's absence for a specified period of time (via timeout), it enters the candidate state and starts an election in an attempt to become the leader itself, thus asking for the other nodes' votes
- log replication → the leader guides the other nodes in replicating the logs, i.e., the logs that contain the operations to be performed and that for this reason must be replicated consistently on all nodes → the leader sends periodic AppendEntries messages to update the other nodes with new log entries
- security and persistence checks → overall security and persistence are ensured through a set of rules that must be checked periodically so as to ensure data consistency.

As anticipated, fundamental to the Raft protocol is the concept of role, as each node can assume one of three roles present:

- **follower** → initial role for each node, involves waiting for commands and directions from the leader and participating in elections when the leader is no longer active
- **candidate** → a node becomes a candidate when, no longer receiving messages from the leader, it decides to attempt to become the new leader by initiating an election and asking for votes from the other nodes
- **leader** → is the node responsible for managing log replication and coordinating the activities of follower nodes.

As mentioned, Raft is also based on the concept of RPCs; in fact, it provides three basic types:

- **RPC RequestVote** → a candidate sends this request to get the votes of other nodes during the election
- **RPC AppendEntries** → the leader uses this request to replicate new log entries to all other nodes
- **RPC Snapshot** → this request is used to optimize the synchronization of the nodes, allowing them to receive a snapshot of the database instead of replicating the entire log → since this is a request that brings purely performance benefits it was decided not to implement it in our project, so as to focus more on the relevant development and deployment aspects.

2.2 Background on technologies used

. Another preliminary aspect of great importance in order to clearly understand the following sections is the technologies and frameworks used, as well as the reasons for using them:

- **MySQL** for relational DB → for its ease of use.
- **HTML, Javascript** and **CSS** for web clients
- **Node.js** for backend → because it is the reference framework for backend written in JS
- **Jest** for the testing phase
- **HTTP protocol** and web sockets for communication
- **Docker** for a simulated deployment.

3 Requirements Analysis

3.1 Requirements Analysis: functional requirements

In the functional requirements of the application fall all the features and services that the system offers:

- new user registration and login
- new auction creation with the possibility of describing the item for sale and its starting price
- closing of an auction (at a time of the creator's choice)
- display of all auctions with possible filtering between only open or only closed ones
- display of the details of an auction, such as specifications of the item for sale and last bids

- participation in an auction by submitting a bid
- all editing and entry operations on the database must comply with the Raft consensus protocol.

3.2 Requirements Analysis: nonfunctional requirements

Non-functional requirements express the characteristics and properties that a system must meet.

The following nonfunctional requirements are present in the project:

- **scalability** (both with regard to the consensus algorithm and the web application) → the system ensures proper operation even in the face of a large number of users in the site or nodes that make up the Raft cluster → however in any case an excessive increase of connected users could cause slowdowns and malfunctions due to the inherent structure of the Raft algorithm
- **security** → in addition to the benefits brought by the use of Raft, the system also covers the most common areas of security such as information replication and redundancy or site access via authentication
- **performance** → the system must be able to provide answers in an acceptable time, as well as to withstand a variable amount of active users at the same time
- **reliability** → the system must be available and reachable at all times
- **usability** → the web application has been designed and realized, not only in the interface but also in the functionalities, putting in first place the simplicity and ease of use for the end user → it follows also special attention towards accessibility towards users with disabilities
- **maintainability** → the code has been written in a clear, simple and understandable way → special attention to writing documentation ensues.

3.3 Requirements Analysis: implicit requirements

Implicit requirements, not explicitly defined above but nevertheless fundamental, include:

- **compatibility** → the website is designed to be supported on all popular browsers and also on mobile devices
- **software quality testing** → the software has been subjected to rigorous testing and checks, both at the level of the entire system and at the level of verifying the correctness of individual components.

3.4 Requirements Analysis: implicit assumptions

There are some implicit assumptions that are worth mentioning, as they influence the design of the system:

- implicit assumption on the initial configuration of the Raft cluster → although the algorithm is implemented in such a way as to ensure extensibility and customization of the cluster of Raft nodes, it must be assumed that there is some sort of system administrator who knows the IP addresses and ports that each Raft node makes available for the implementation of the protocol → to this administrator is therefore reserved the task of compiling, prior to the startup of the cluster, the configuration file of the cluster in which precisely these aspects related to sockets and connections between nodes are specified
- implicit assumption on the minimum cardinality of the Raft cluster → for a proper functioning of the consensus algorithm it is necessary to assume that the cluster is composed of at least 5 Raft nodes, so as to have a clear and defined majority even if some node is temporarily inactive → this assumption is actually guaranteed in our implementation, since we have 5 machines playing the role of 5 nodes in the Raft cluster.

3.5 Requirements Analysis: models/paradigms/technologies

Among the models, paradigms and technologies that are best suited to address the previously mentioned requirements are:

- client-server architectural paradigm → however it is good to remember that the paradigm is client-server for the service and server-server for the consensus algorithm
- relational database model → chosen for its simplicity
- common and popular paradigms such as node partitioning, load distribution on available nodes, microservice-oriented development, and multi-level security.

3.6 Requirements Analysis: abstraction gap

An abstraction gap is present between the models and paradigms available today and the problem to be solved.

This difference actually concerns multiple aspects, but it is certainly worth mentioning:

- abstraction gap with the client-server paradigm → actually the architecture of the web application is not simply client-server but needs to be revisited and adapted since it is not simply a matter of getting responses from the server but an additional layer of components needs to be inserted between these two components to which the task of implementing the consensus algorithm is entrusted

- abstraction gap between the required simplicity and the implementations already present at the state of the art → the implementations present today of Raft, although complete, lack in our opinion important aspects such as comprehensibility, simplicity and adaptability with respect to a realistic deployment, i.e., on truly distributed nodes in the network.

4 Design

The system design activity was long and complex, but it can be broken down into 6 phases, each focusing on a particular and different aspect:

- network design
- logical architecture design
- architectural style
- structural design
- behavioral design
- interactive design.

The following subsections describe the above steps in more detail.

4.1 Network design

Before proceeding with the designs of the various structural, behavioral, and interactive aspects, it was decided to focus a fair amount of the analysis on evaluating the possible configurations of the overall network infrastructure in charge of providing the required distributed service.

The various options that emerged can be summarized in the two types of architectures that are described below.

The first type of structure analyzed was as follows:

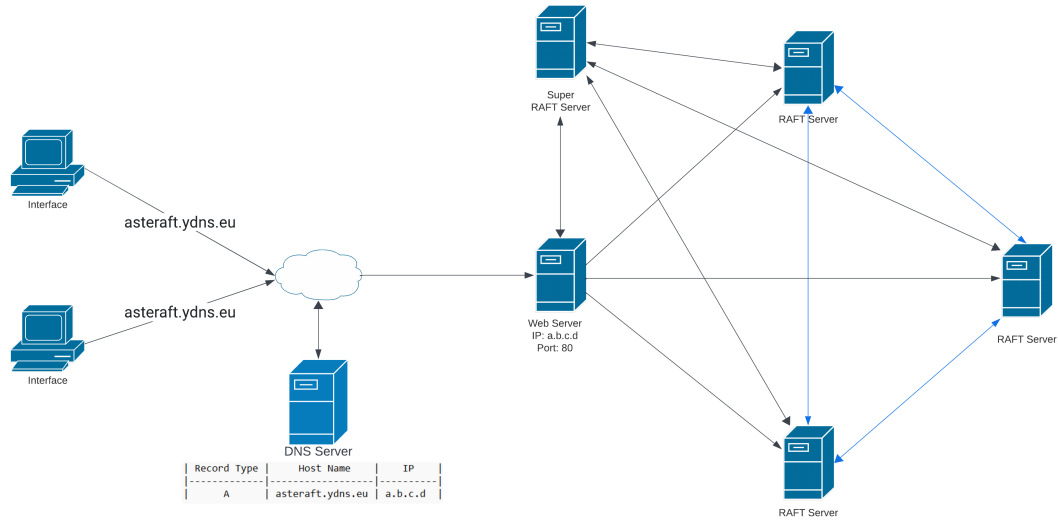


Figure 2: the first idea of network infrastructure analyzed shows the vastness and heterogeneity of components needed (client, DNS server, web server, Raft super server, Raft server...).

As can be seen from the figure, this type of network allows users to connect to the only web server present using a browser.

In this architecture a client, in order to access or act on the database, is required to contact the web server and perform the following chain of steps:

- contact the super server by forwarding to it the request it wishes to implement on the database
- wait for the super server to tell it the address of the Raft server that currently holds the leader role
- contact the leader by forwarding to it the request it wishes to implement on the database.

Note that in order to correctly apply the consensus algorithm, the servers belonging to the Raft cluster need to refer to the super server: this is because it plays a central role in the structure in question since it contains a configuration file in which the addresses of all the other servers participating in the protocol are listed.

Although this mode has interesting advantages in terms of network scalability (which is implemented by simply adding the addresses of the desired servers in the configuration file of the super server) and in terms of flexibility (since the nodes do not need to know each other since the super node acts as an intermediary), this solution shows numerous problems, mainly related to the low reliability and vulnerability of the only web server

present: if for some reason it is attacked or does not behave correctly, the entire infrastructure stops working.

In addition to the fragilities of the web server, one must also consider those of the super server, which in fact represents a single point of failure despite the fact that it performs two fundamental tasks such as redirecting the web server to the leader and communicating by consensus protocol the addresses of the servers participating in the cluster.

To resolve the problems of the previously described network infrastructure and attempt to simplify it, it was then decided to implement the following alternative architectural solution:

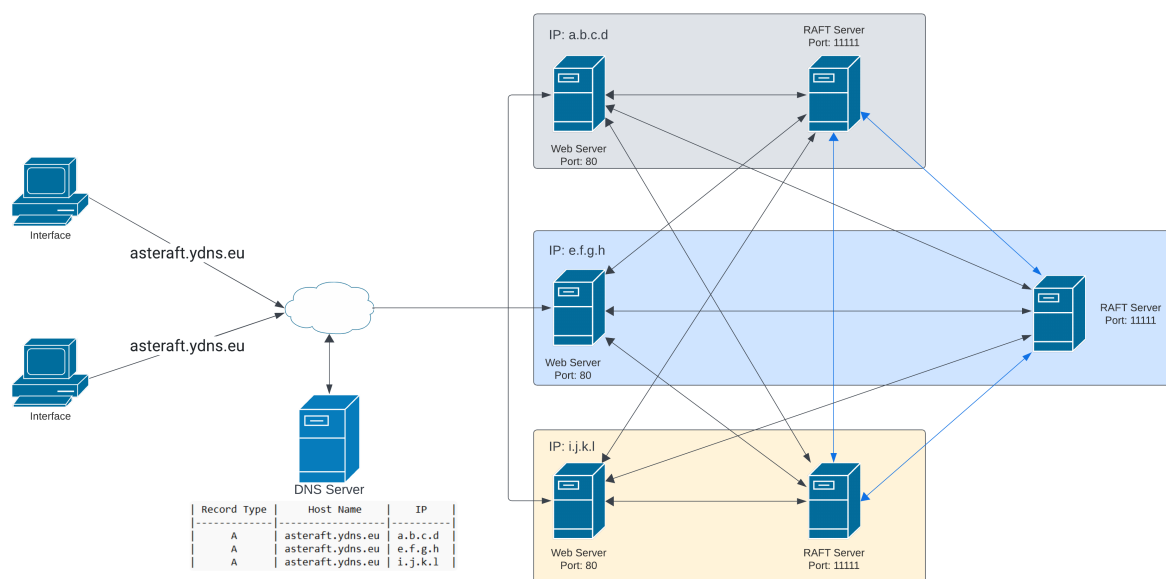


Figure 3: architectural idea actually implemented.

The peculiarity of this other network infrastructure compared to the previous one lies in the distribution of Web servers.

In order to properly implement this configuration, it was necessary to use a load balancing technique known as Round Robin DNS, which provides several IP addresses associated with a common hostname.

In fact, the following screenshots show that trying to contact the same hostname `http://asteraft.ydns.eu` results in being directed to two different IP addresses:

```

:~$ ping asteraft.ydns.eu
PING asteraft.ydns.eu (91.81.161.7) 56(84) bytes of data:
64 bytes from 91.81.161.7: icmp_seq=2 ttl=47 time=61.8 ms
64 bytes from 91.81.161.7 (91.81.161.7): icmp_seq=6 ttl=47 time=58.5 ms
64 bytes from 91.81.161.7 (91.81.161.7): icmp_seq=7 ttl=47 time=67.6 ms
64 bytes from 91.81.161.7 (91.81.161.7): icmp_seq=9 ttl=47 time=61.1 ms
64 bytes from 91.81.161.7 (91.81.161.7): icmp_seq=10 ttl=47 time=75.6 ms
64 bytes from 91.81.161.7 (91.81.161.7): icmp_seq=13 ttl=47 time=67.2 ms
64 bytes from 91.81.161.7 (91.81.161.7): icmp_seq=14 ttl=47 time=68.4 ms
64 bytes from 91.81.161.7 (91.81.161.7): icmp_seq=17 ttl=47 time=63.2 ms

```

```

C:\Users\andre>ping asteraft.ydns.eu -t
Pinging asteraft.ydns.eu [79.23.243.113] with 32 bytes of data:
Reply from 79.23.243.113: bytes=32 time=73ms TTL=49
Reply from 79.23.243.113: bytes=32 time=69ms TTL=49
Reply from 79.23.243.113: bytes=32 time=71ms TTL=49
Reply from 79.23.243.113: bytes=32 time=70ms TTL=49
Reply from 79.23.243.113: bytes=32 time=70ms TTL=49

```

Using this particular DNS technique, the steps to be followed for a client to send its requests are:

- request the page to the URL `http://asteraft.ydns.eu` → as mentioned several IP addresses (web servers) are associated with that hostname
- any of the available web servers, upon receiving the client's request, contacts any of the Raft servers
- such a Raft server responds to the web server saying whether it is the leader or not (in the latter case it can tell it the ID of the leader node, if it knows it)
- the web server then contacts the leader directly, asking it to append to the replicated log the operation the client wishes to perform on the database, effectively initiating the consensus algorithm.

Although the number of steps is slightly higher than in the previously analyzed network infrastructure, the complexity of these steps turns out to be much lower, and what is more, no central super server is needed, which actually drastically reduces the overall vulnerability of the system.

As can also be seen in the figure, due to the limited availability of machines and IP addresses, it was decided that each of the available machines will start a process internally for both a web server and a server for the Raft consensus protocol, of course remaining listening on different ports: port 15000 is for communications between nodes, while 15001 is reserved for communications between the node and the web server.

It is also very important to note that in both this mode and the previous one, the Raft consensus algorithm client actually switches from being the client interface to the web server making requests on its behalf.

The only issue here is related to network scalability: if Raft servers are to be added (or even removed), each configuration file on each server must be modified.

Despite the fact that this aspect hindering the ease of extension of the network infrastructure could be mitigated with some expedients such as the creation and sharing in some server of an updated list of active servers participating in the consensus protocol, since this is not a central aspect with respect to the objectives of the present project it was thought momentarily to ignore it, keeping it however in consideration for possible future developments.

In any case, it is worth noting that despite the disadvantage noted above, the structure turns out to be quite reliable in case of failures.

4.2 Logical Architecture

If in the previous section we described from the physical point of view the architecture of the implemented network infrastructure, let us now see the same architecture but from the logical point of view:

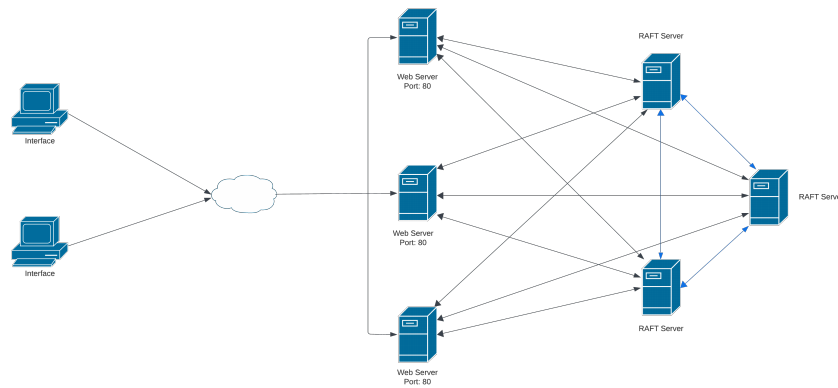


Figure 5: logical architecture of the system.

Very briefly, as can be seen from the image, from the logical point of view there are different clients that connect to the web server in order to request the desired web page. The task of the web servers is not only to serve the client requests but also to update themselves according to the indications that emerge from the consensus reached by the cluster.

Note that the overall operation of such an architecture is totally independent of the physical placement of the various web servers and Raft.

4.3 Architectural style

As also mentioned during the course, in the design phase of a distributed system it can be extremely useful to identify the correct architectural style in order to facilitate the next steps.

Of the various architectural styles presented to us in class, the one that best suits the project is the object-based architectural style:

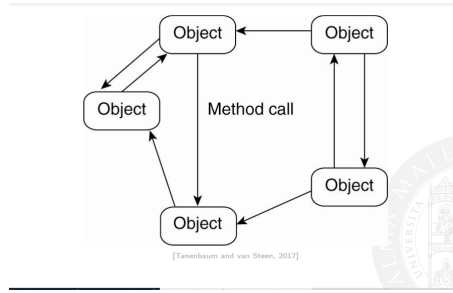


Figure 6: object-based architectural style.

The object-based architectural style, which underlies most common client-server architectures, is based on the idea that system components are objects capable of interacting with each other through RPCs (a concept strongly reminiscent of the method of component communication within Raft).

4.4 Structure / Domain Entities

To better understand which domain entities needed to be modeled, refer to the following UML class diagram:

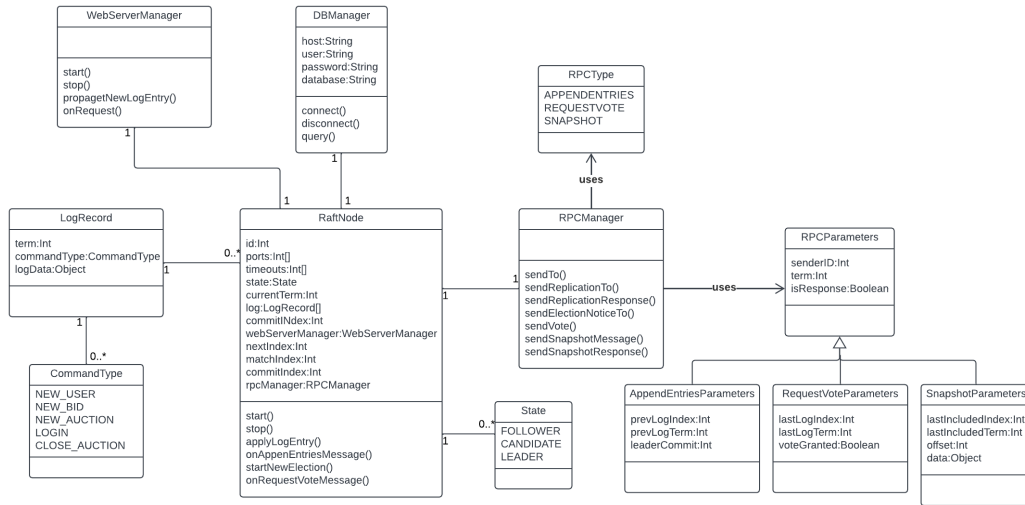


Figure 7: UML class diagram.

Thus, the UML class diagram highlights the following entities:

- **RaftNode** → class for the Raft protocol node
- **State** → enumerator for the possible states that a Raft node can assume

- **LogRecord** → class for the node's log record
- **CommandType** → enumerator for the possible commands contained in a log record
- **WebServerManager** → class that handles communications between Raft nodes and web server
- **DBManager** → class that handles communications between Raft nodes and database
- **RPCManager** → class that handles communications between Raft nodes
- **RPCType** → enumerator for the possible types of RPCs that Raft nodes can exchange with each other
- **RPCParameters** → class for the possible parameters based on the different RPC type.

Finally, note that in the above UML class diagram there are no server and client related classes: this was done for readability and also because they do not have some particular implementation, they simply use sockets to communicate with the Raft cluster.

Instead, the modularization of the various classes can be observed from the following UML Package diagram:



Figure 8: UML package diagram.

As can be seen from the image, the three main modules that make up the application are:

- web client package → containing all the code (including HTML and CSS) client-side
- web server package → containing all server-side code
- Raft node package → containing all entities useful for implementing the consensus algorithm (RaftNode, Log, DBManager...).

4.5 Behaviour

The behavior of the main entities in an activity diagram is shown below:

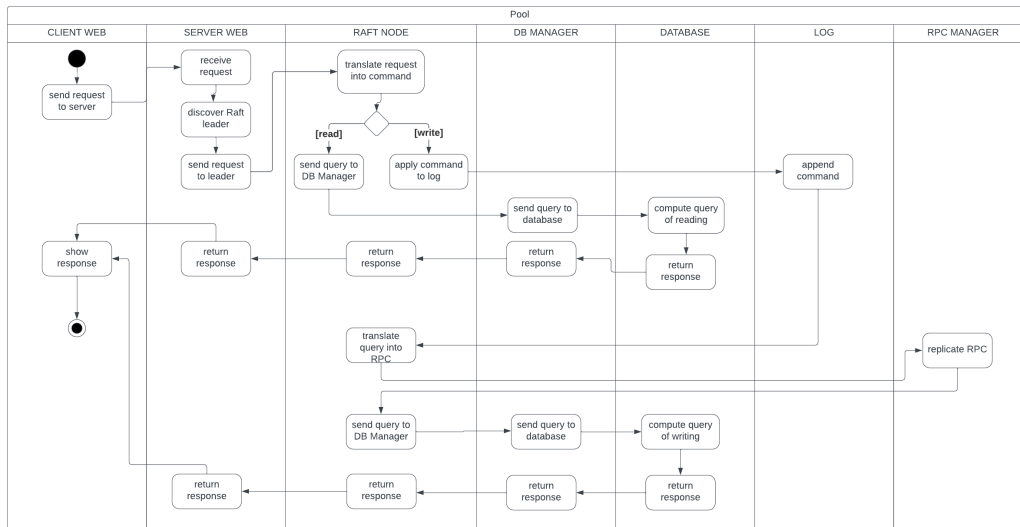


Figure 9: UML activity diagram.

4.6 Interaction

The following UML sequence diagram shows the interactions between the main entities:

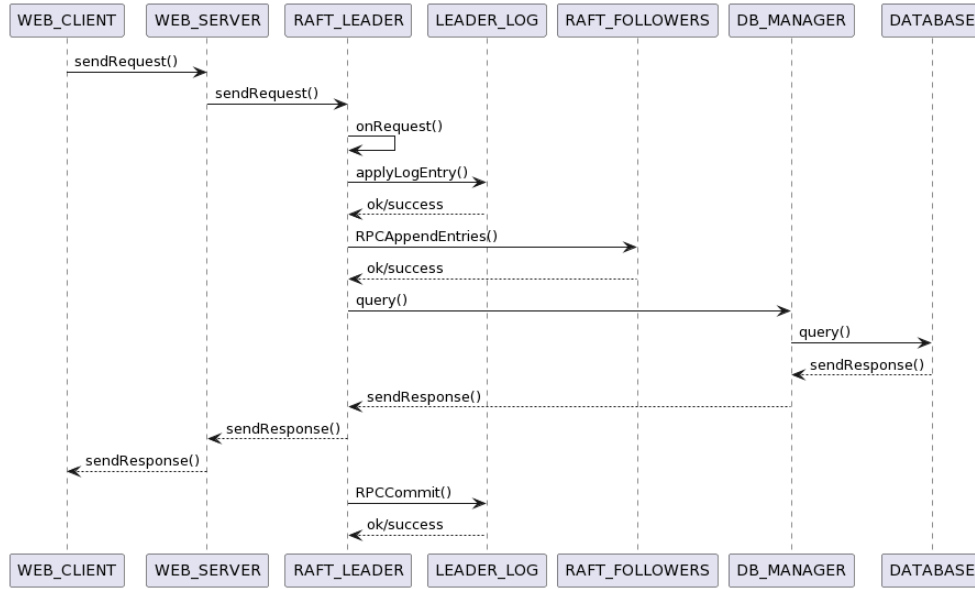


Figure 10: UML sequence diagram.

5 Implementation Details

The following are some interesting and nontrivial implementation details.

The paper on which our project is based does not express any implementation details about the configuration phase of the network infrastructure useful for the Raft consensus protocol, so it was necessary to handle this phase completely independently.

It was decided to create a JSON file called **cluster-config.json** in which it is possible to specify:

- the desired size of the cluster
- the port on which all servers in the cluster will be listening for communications regarding the consensus algorithm
- hostname and id of each node in the cluster.

We chose this strategy since we felt it was the simplest approach, both in terms of understanding and use but also in terms of extensibility.

Aiming to simplify our implementation, it was also decided:

- not to allow changes to the cluster (addition/removal of nodes) while it has already been started

- to not maintain the log following the cluster shutdown → this implies that all database data will be deleted if the Raft cluster is restarted → although it seems like a stretch, this strategy became necessary as it ensures that when the cluster is started all its nodes are aligned to the same content, having no previous data in the log
- when a client makes a request involving a database change, the leader also includes in the record entered in the log a callback necessary to alert the client once a commit is made to the record → if, however, the leader stops working before the commit and subsequent response to the client, the client's request will remain unsatisfied because callbacks cannot be serializable (and thus cannot be transported and handled by sockets).

To see all the implementation details, you can read the documentation produced at the following address: https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/ds-project-bedei-bertuccioli-notaro-ay2324/-/tree/main/docs?ref_type=heads

6 Self-assessment / Validation

Several types of tests, at different levels of detail and focusing on different aspects, were required to verify the correctness of the software produced. The account of the parts tested is available in the subsections that follow.

6.1 Testing of the web application

To verify the correctness of the web application, a series of tests and checks were performed regarding various aspects and behaviors, among which it is worth mentioning:

- check of the correctness of the visualization → we made sure that the interfaces were clean and correct, both with desktop and mobile visualization
- verification of credentials → we made sure that the system allows entry to users with only a valid username and password
- logical correctness of operations → we ensured that the application responds correctly and robustly to potentially risky user behavior, such as creating empty auctions or creating bids of less than acceptable value
- logical correctness in filling fields → we made sure that the system does not allow filling fields with values of the wrong type or content (e.g., words where numeric values are expected).

6.2 Testing of the Raft architecture

Regarding the verification of the correctness of the Raft infrastructure, in the sense of the creation of the components and their correct interaction, tests were performed both

truly distributed and on a single machine, then analyzing the logs of the different nodes line by line in order to make sure that the interaction was taking place correctly. This allowed us to verify, along with the following subsection, the correctness of the implementation of the Raft consensus algorithm.

Below are screenshots to understand what happens when a node asks to replicate a record in its log:

```
[node3 (leader)]: Set commit index to 0
[node3 (leader)]: Received successful "appendentries" response from node2 -> ignored (ok).
[node3 (leader)]: Committed 1 log entries to database.
[node3 (leader)]: Added new user to database.
[node3 (leader)]: Leader COMMIT
[node3 (leader)]: Received successful "appendentries" response from node1 -> ignored (ok).
```

Figure 11: leader's point of view, note the increase in the commit index and the commit of the record.

```
[node2 (follower)]: Committed 1 log entries to database.
[node2 (follower)]: Received "appendentries" request from node3 with term 2 -> responded.
[node2 (follower)]: Received "requestvote" request from node1 -> refuse vote.
[node2 (follower)]: Added new user to database.
[node2 (follower)]: Client COMMIT
```

Figure 12: point of view of the follower, note the receipt of the request to append a record to the log and the correctness of the commit.

6.3 Testing of Raft components

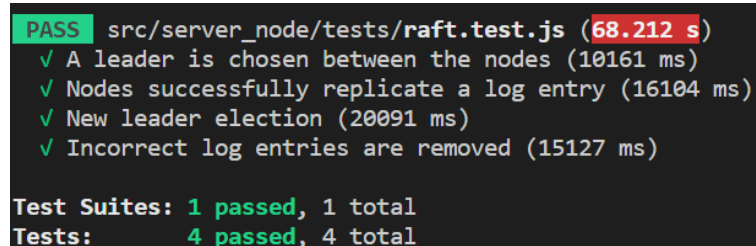
To verify in more detail the correct behavior of the Raft components, the Jest library was used, which is a JavaScript framework designed to ensure code correctness verification and test-driven development.

In particular, the following tests were performed, which can be found within the file `src/server_node/tests/raft.test.js`:

- test "A leader is chosen between the nodes" → instantiates a cluster of nodes and verifies that a leader is elected at some point → this test verifies the correct election of the leader
- test "Node successfully replicates log entry" → a cluster of Raft nodes is instantiated, showing that at the beginning the length of the nodes' logs is 0, it is then verified that a node becomes a leader and that when it orders the replication of a record the length of each log is increased to 1 → such test verifies the correct replication and growth of the logs

- test "New leader election" → the test shows that if the leader stops working a new one is elected
- test "Incorrect log entries are removed" → this test initializes the cluster, waits for a leader to be elected, after which it adds a record to a follower's log and makes sure that after a while that record is removed since it was not committed.

The following screenshot shows that all tests pass correctly:



```
PASS src/server_node/tests/raft.test.js (68.212 s)
  ✓ A leader is chosen between the nodes (10161 ms)
  ✓ Nodes successfully replicate a log entry (16104 ms)
  ✓ New leader election (20091 ms)
  ✓ Incorrect log entries are removed (15127 ms)

Test Suites: 1 passed, 1 total
Tests: 4 passed, 4 total
```

Figure 13: screenshot depicting the tests being passed.

These tests therefore show and demonstrate the correctness of our implementation with respect to the main functionalities that Raft provides.

7 Deployment Instructions

In order to launch the produced software, it is not necessary to install any components: as per agreements made in class and as the realistic node distribution allows, the cluster with 5 nodes owned by the group is in fact always active and ready to receive requests. It is therefore sufficient to open a browser and connect to <http://asteraft.ydns.eu> to evaluate the application produced.

If, on the other hand, you would like to start the entire system from scratch, perhaps even modifying the configuration parameters, keep in mind that this is a very complex operation, since each node should have node.js installed and a MySQL server active and with a database already created.

In this eventuality, the following steps should be followed carefully:

- clone the repository <https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/ds-project-bedei-bertuccioli-notaro-ay2324>
- edit the file **src/server_node/cluster-config.json** specifying the communication ports for the Raft protocol, the credentials and information to connect to the database, and all the information (host, id, and port) of the other nodes in the cluster excluding the current node

```

{
  "port1": 15000,
  "port2": 15001,
  "protocolConfig": {
    "minLeaderTimeout": 2000,
    "maxLeaderTimeout": 3000,
    "minElectionTimeout": 3000,
    "maxElectionTimeout": 5000,
    "minElectionDelay": 1000,
    "heartbeatTimeout": 1000
  },
  "ownId": "node2",
  "dbHost": "127.0.0.1",
  "dbUser": "fabio",
  "dbPassword": "password",
  "dbDatabase": "portale",
  "otherNodes": [
    {
      "host": "abedo.ddns.net",
      "id": "node1",
      "port": 15000
    },
    {
      "host": "79.23.243.113",
      "id": "node3",
      "port": 15000
    }
  ]
}

```

Figure 14: cluster configuration example.

- edit the file `src/server_web/cluster-config.json` specifying the port on which the web server listens and again all the cluster node information (host, id and port)
- locate yourself in the **RAFT_JS_APP** folder and start the Raft cluster via command `node src/server_node/launch.js` → if all the previous steps have been done correctly you can see the Raft nodes will start exchanging messages
- open a new terminal, locate in the same **RAFT_JS_APP** folder and start the web server via command `node src/server_node/web/server.js` → this command will return the address at which the web application can be accessed.

8 Usage Examples

As also mentioned in the previous section, to use the developed software it is sufficient to search in any browser for the address `http://asteraft.ydns.eu` since the availability of the Raft cluster is guaranteed and continuous.

Once the site is opened it is possible to create a new account, log in, and after that all the operations specified in the Goals/requirements section are possible.

Due to the extreme simplicity of the web application we do not report the steps to perform the individual actions since they would be trivial.

9 Conclusions

In conclusion, the group is very satisfied that they were able to complete the project within the agreed time and requirements.

All phases of the project were successfully completed and developed.

To recap what we did:

- there was a very first phase of studying and understanding the paper explaining how the Raft consensus algorithm works
- having finished the study phase, the implementation phase of the algorithm in javascript began → the possibility of taking cues from existing implementations was considered, however no repository present today was considered by us to be adequate as a starting point (due sometimes to lack of clarity and sometimes to incompleteness)
- the implementation phase of the consensus algorithm was accompanied continuously by testing, not only in the final phase → the group therefore tried as much as possible to comply with the test-driven development approach, including by learning and using the Jest framework
- was then planned to make the Raft cluster truly distributed by using 5 heterogeneous nodes from the performance point of view and spatially distributed
- always in parallel with the algorithm implementation phase, the report writing and web application implementation phases were carried out
- finished all the previous phases, testing moved from verifying the correctness of the interactions between the Raft nodes to verifying the correctness and robustness of the web application
- also finished the testing phase of the entire system, the project was delivered.

9.1 Raft advantages and disadvantages

The following is a table specifying the advantages and disadvantages that our implementation of the Raft algorithm has allowed us to notice and which we think it is fair to highlight and report as relevant from a distributional point of view:

Table 1: advantages and disadvantages of Raft

ADVANTAGES	DISADVANTAGES
<ul style="list-style-type: none"> • clear and understandable structure • simple leader election • improved fault tolerance • improved security compared to other algorithms • structure easily modularized 	<ul style="list-style-type: none"> • higher communication overhead compared to simpler algorithms • implementation complexity despite ease of understanding • difficult optimization for networks with very high latencies • possible performance problems in very large network scenarios • absence of valid open source implementations in the literature

9.2 Known vulnerabilities and limitations

This subsection lists a number of aspects of vulnerabilities and limitations of the product system that are of interest from the perspective of distributed systems, each accompanied by the rationale for taking no action and possible solutions:

- vulnerability to byzantine attacks → the Raft algorithm is known to be vulnerable to byzantine attacks, i.e., there is no way to contrast an attempt to tamper with it by the leader, and in such a case its actions could compromise the integrity of the system → however, as mentioned in the implicit assumptions section, such an attack is mitigated by the fact that the cluster nodes are chosen manually and thus assumed to be secure and loyal
- vulnerability to election forcing → the system becomes vulnerable if a malicious node forces a new election by running with a higher term than the current one (assuming the node's log is up to date), effectively cheating to become a leader → this is also a pathological vulnerability inherent in the consensus algorithm itself, in fact countering this type of attack would mean preventing a node from becoming leader if the current one is still active
- implementation choice of reset following cluster shutdown → as anticipated several times throughout the report, we decided to adopt a rather drastic implementation choice following the shutdown of the entire cluster, namely the emptying of all local databases → we opted for this simplification since the alternative would be

to periodically save the log in some way, however the state of the saved log and the state of the database might diverge in the case of unexpected errors and as a result we would arrive at a divergence of states that is not reconcilable and remediable

- limitation of the reset following the shutdown of a node → similarly to the previous case, if a node is shut down or fails its database is emptied → such drastic strategy is purely due to simplicity issues since, even by saving the log, in crash situations that create inconsistency between log and database it may not always be possible to reconcile the two
- deadlock caused by lack of majority → if only one or at any rate a minority of the nodes remain active in the cluster compared to the total, a deadlock situation is reached whereby no new records can be appended on the log until a majority is reached again → this is also an inherent problem of Raft, as well as of all consensus algorithms based on quorum and majority.

9.3 Future Works

Future works about missing or improvable aspects concern:

- implementation of the Snapshot RPC → the consensus algorithm would also provide for a snapshot RPC useful for log compression, but such functionality has only been prepared by us but not implemented
- as also reported in the previous subsection, future work could address improving persistence following shutdown/disconnection of nodes
- possibility to also add photos of the item for sale via auction.

9.4 What did we learned

The project was very useful in addressing and deepening from a practical point of view notions and principles covered in the course.

In particular, among the most relevant concepts that emerged in the development of the system and that were also presented to us in class are worth mentioning:

- learning to work properly in situations where there is no longer a general notion of common and shared system time and location → to mitigate this problem we tried to achieve shared logical time through the use of various timers
- it is possible to conceive and design distributed systems in such a way that if a component fails or becomes unreachable it is possible to replace it to hide the failure from the outside world or otherwise reduce the impact of perceived failure (fault tolerance) → this principle has also been achieved through log replication and redundancy

- one must expect a distributed system to be unreliable, i.e., to have failures, message leaks, and attacks → during the development of each feature and component one must anticipate and assess how it may break or be misused for illicit advantage
- sequential consistency → in a distributed system consistency is guaranteed even if data are not replicated directly but, starting from the same resource, processes perform the same sequence of operations on their copy
- it is necessary to learn how to handle any case and situation in which in a distributed system a party may be interested in lying or taking advantage of register tampering
- needs to think carefully about how to ensure consensus among distributed nodes
- special attention should be paid to the challenges related to system distribution and consensus → lack of global time between machines, temporary inability of a node to communicate, and byzantine or sybil attacks can generate anomalous situations
- architectural styles are a very powerful tool in analysis and design to help design distributed systems architectures correctly.

In short, to sum up, the group is satisfied with the work done and felt very stimulated during the development of the project by the fact that in the span of the project, many of the distribution aspects discussed in class actually emerged.

Thanks to this phenomenon, the concepts learned were also approached from a practical point of view, making them clearer and deepening them so that they could be assimilated more easily and thoroughly.

References

- [1] Diego Ongaro and John Ousterhout, *In Search of an Understandable Consensus Algorithm (Extended Version)*, Stanford University.