Institute of Information Systems Engineering

Distributed Systems Group (DSG)
VU Distributed Systems 2024W (194.024)

Assignment 3

Submission Deadline: 18.01.2025, 18:00 CET

Last modified: 30.11.2024

Contents

1 General Remarks

Please read the assignment carefully before you get started. Section 2 explains the application scenario and Section 3 & Section 4 present the specification of individual components. If something is not explicitly specified (e.g., the behavior of the application in a specific error case) you should make **reasonable assumptions** for your implementation that you can justify and discuss during the interviews.

1.1 Learning Objectives

In this assignment, students will:

- Gain experience with distributed algorithms and network communication in Java.
- Extend an existing message broker with leader election capabilities.
- Implement UDP-based monitoring to track statistics.

1.2 Guidelines

1.2.1 Assignment Mode

This assignment is **individual** work only. Therefore, group work is not permitted. While we encourage you to exchange ideas and engage in discussions with your colleagues, the code you submit must be entirely your own.

1.2.2 Code Repository

The repository provided upon joining the GitHub Classroom assignment is set to **private** by default. **Do not** change the repository visibility to **public**. If you choose to use other version control hosting platforms, ensure that your repositories remain private.

1.2.3 Generative AI and Tools

The use of code generation tools is permitted, but you **must fully understand** both the code and the corresponding theoretical background of any code you submit. There will be no room for interpretation, nor will we make any exceptions if the use of generative tools results in plagiarism. Therefore, use these tools at your own risk.

1.3 Grading

The achievable points (25) of this assignment are based on automated tests and a submission interview. You can achieve a score of up to 125 (i.e., 12.5 points) if all tests pass, and an additional 12.5 points during the submission interview. Below is a brief overview. Refer to Section 5 for more details.

- Push your code to the `main` branch of the assignments repository provided by GitHub Classroom.
- The last achieved *Grading Workflow* score, counts towards your final assignment grade. Scores from **re-runs** of the *Grading Workflow* are **not** considered for your final assignment grade.
- No late submissions accepted, submit before the deadline.

Submission Interview In the submission interview, you must be able to explain your solution and answer questions that assess your deeper understanding of the technologies used to implement this assignment. We also manually check your submission for non-functional requirements that are not verified by any automated tests. The assignment description includes remarks referring to non-functional requirements (e.g., correct concurrency handling) that constitute a proper implementation. Points will be deducted from the implementation part, if your implementation fails to meet non-functional requirements.

Note for Theory Questions

You can only get Theory Points for parts that you have implemented. The achievable theory points will correspond to the amount of points of the implementation part.

Example: You achieved 4.2 points in the implementation part, therefore you can achieve a maximum of 4.2 points in the theory part.

If you have questions, the DSLab Handbook[1] and the TUWEL forums[2] are good places to start.

[1] https://tuwel.tuwien.ac.at/mod/book/view.php?id=2388365
[2] https://tuwel.tuwien.ac.at/mod/forum/view.php?id=2426579

2 Implementing Leader Election and UDP Monitoring

In this assignment, you will introduce Leader Election capabilities to your existing Message Broker implementation from Assignment 2. Additionally, a new UDP-based Monitoring Server is implemented, which keeps track of the messages sent over the Message Brokers.

The Leader Election functionalities utilize the TCP-based Leader Election Protocol (`LEP`), which enables Message Brokers of a certain pool to agree on a single leader based on the configured election type/mode. Once a leader has been elected, the newly elected leader registers itself with the DNS Server under a pre-defined domain.

The UDP Monitoring Server tracks the routing of the messages by receiving the UDP packets sent by the Message Broker for each routed message. The UDP Monitoring Server provides statistics, such as the number of messages sent for specific routing keys.

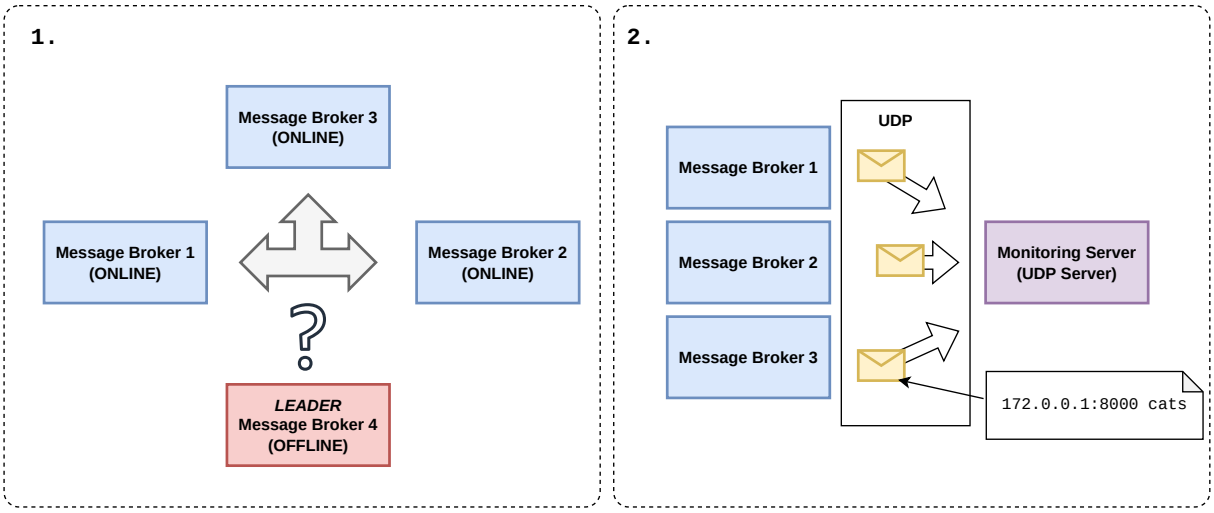

Figure 1: Assignment 3 - Bird's Eye View

2.1 Application Overview

1. **Leader Election**: Multiple Message Brokers can start simultaneously and must determine a leader. The leader registers itself with the DNS (i.e., all Message Brokers have the same domain but only one at a time is registered). The election process is determined by one of four (including *none*) election types. The active type/algorithm is selected on Message Broker startup. Moreover, servers implement a mechanism to detect leader failure and initiate a new election when necessary.
2. **UDP Monitoring Server**: A simple Monitoring Server is to be implemented that uses the UDP protocol[3] to receive packages from Message Brokers. These UDP packages are sent for each routed message and contain the Message Brokers address and the routing key. The Monitoring Server can return the aggregated statistics for printing.

Remarks on Re-use of Code and Plagiarism

We strongly emphasize that you must base your Assignment 3 implementation on your **very own** Assignment 2 implementation.

Do not copy or use your colleagues solutions! Doing so will trigger the plagiarism check and result in a full point deduction for all parties involved.

[3] https://en.wikipedia.org/wiki/User_Datagram_Protocol

3 Leader Election Process

This section is split into three sections:

- **Overview (Section 3.1):** Gives an overview of the Leader Election process.
- **Leader Election Protocol (Section 3.2):** Explains individual commands used to perform the Leader Election.
- **Election Algorithms (Section 3.3):** Lays out details on the three Leader Election algorithms, including examples.

3.1 Overview

The Leader Election process selects a leader from a pool of active Message Brokers. A pool is defined as a set of Message Brokers which fullfills following properties:

1. The pool includes all Message Brokers specified in the `election.peer.hosts` and `election.peer.ports` properties, except for the Message Broker specifying the property.
2. All Message Brokers of a pool specify the same property `election.domain`.

The leaders acts as the primary communication gateway for clients for a specific domain. That means, multiple Message Brokers can be started for one domain (e.g., **election.brokers.at**) and have to collectively elect a single leader. The assignment involves implementing three different algorithms for Leader Election: **Bully**, **Ring**, and **Raft**. Which algorithm should be used by the Message Broker, can be specified inside the corresponding `.properties` of the Message Broker, and it is to be expected that all servers follow the same Leader Election Algorithm. Upon successful election, the leader registers its `IP:Port` with the domain specified under `election.domain` in the leaders `.properties` file, with the DNS Server.

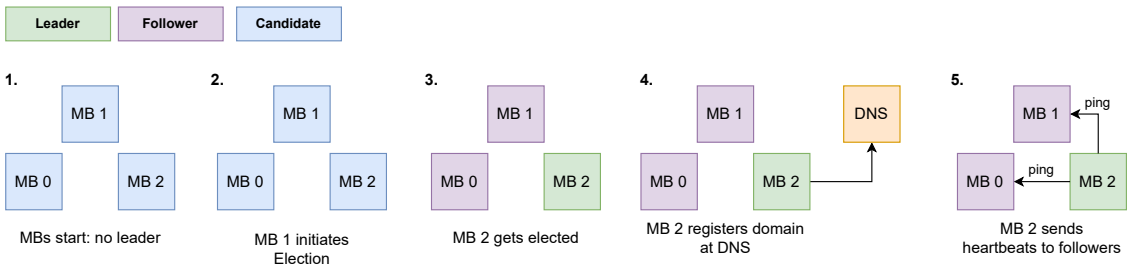

Figure 2: Leader Election Process

Figure 2 depicts such a scenario, where three Message Brokers are started. After the startup of the Message Broker, a Message Broker initiates the election. After successfully performing the election process, the new elected leader registers itself with its `IP:Port` combination at the DNS Server. Each Message Broker can be in one of three states: `candidate`, `follower` or `leader`. During the election process, all Message Brokers initially start as `candidate`. Once a `leader` has been elected, all other Message Brokers transition to the follower state, while the newly elected leader transitions to the `leader` state. Afterwards, any client requesting that domain will communicate exclusively with the leader.

The active leader periodically sends heartbeat (health-check) messages to all followers to indicate it is still alive. In case that an leader becomes unavailable, a new election must be initiated. Therefore, if the leader stops sending heartbeats, the followers detect the failure of the leader and thus initiate a new election. Each follower has an individual timeout duration within which a heartbeat (i.e., `ping`) must be received and will otherwise trigger an election.

Election Timeout: If a follower receives no heartbeat communication over a pre-defined period of time (also referred as the *election timeout*), then it assumes there is no viable leader and initiates an election

to decide on a new leader. The *election timeout* is defined in the `election.heartbeat.timeout.ms` property.

3.1.1 Network Communication Overview

Suppose we have a small setup with three Message Brokers and a single DNS Server. The Message Brokers have following configurations which are defined in the corresponding `*.properties` files:

Property/Server	MB 0	MB 1	MB 2
election.id	0	1	2
election.type	ring	ring	ring
election.port	20001	20011	20021
election.domain	election.brokers.at	election.brokers.at	election.brokers.at
election.peer.hosts	127.0.0.1,127.0.0.1	127.0.0.1,127.0.0.1	127.0.0.1,127.0.0.1
election.peer.ports	20011,20021	20021,20001	20001,20011
election.peer.ids	1,2	2,0	0,1
election.heartbeat.timeout.ms	100	200	300

Step 1: Starting the Servers

The DNS Server and Message Brokers are started with their respective configurations.

1. `DNS 0` starts and listens on the port specified by `dns.port`.
2. `MB 0, MB 1, MB 2` start. The Message Brokers listen for incoming `SMQP` communication on `broker.host` and for `LEP` communication on `election.port`.
3. `MB 0, MB 1, MB 2` start their heartbeat timeout specified in `election.timeout`.

Step 2: broker-0 reaches Heartbeat Timeout

MB 0 has the lowest timeout of 100 milliseconds, so it is the first Message Broker to reach its timeout.

1. 100 milliseconds pass.
2. *MB 0* reaches its timeout.
3. *MB 0* initiates a new election by sending `elect <id>` to the next peer in the ring.

Note: Above election initiation process is specific to the Ring Election.

Step 3: Election Process

1. *MB 0* has initiated a new election by sending `elect <id>` to the next peer in the ring.
2. Go through the full Election Process for Ring Election.

Note: The above election process is specific to the Ring Election and is therefore not directly applicable to other Leader Election types.

Please refer to the Election Types section for a detailed description on each election type.

Step 4: Leader elected

MB 2 wins the election, because `ring` has been configured as the election type and *MB 2* has the highest ID. *MB 2* is now the declared leader and `MB 0, MB 1` become the followers of the new leader *MB 2*.

1. *MB 2* wins the election and becomes the new leader.
2. *MB 2* registers itself at `DNS 0` using the domain specified in `election.domain` property.
3. *MB 2* → `DNS 0`: `register election.brokers.at 127.0.0.1:20020`

Step 5: Heartbeats

Since *MB 2* is the leader, it sends heartbeat message (`ping`) to all of its followers (i.e., *MB 0*, *MB 1*) every `x` milliseconds.

Note: It is recommended to choose an heartbeat interval which satisfies $2 * inteveral \leq lowestTimeout$. In this example we choose 50 milliseconds.

Every 50 milliseconds:

- *MB 2* → *MB 0*: ping
  MB 0 → *MB 2*: pong
- *MB 2* → *MB 1*: ping
  MB 1 → *MB 2*: pong

3.2 Leader Election Protocol (LEP)

The Leader Election Protocol is a TCP-based plaintext application-layer protocol and thus behaves similarly to the protocols `SMQP` and `SDP` you have implemented in Assignment 2. It is used to coordinate the Leader Election process between multiple Message Brokers.

3.2.1 LEP Key Actions

- **Initiate Elections:** Crucial to Leader Election is the initiation of an election. For that the `elect <id>` command is used and following Message Brokers initiate an elecation, once the current leader has missed sending a heartbeat message within the followers heartbeat timeout. To which the sender receives `ok` as response in case of Ring and Bully and `vote` during a Raft election (see further below for more details).
- **Declaring a Leader:** Upon finalizing the election, the newly elected leader will declare itself to others. This is done via the `declare <id>` command. The other Message Brokers (receivers) acknowledge the new leader by replying with an `ack <receiver-id>` command.
- **Heartbeats:** The leader must continuously send heartbeat messages (`ping`) to its followers, which respond with `pong`. Should a follower not receive a hearbeat within its defined timeout-window, then it initiates a new election.
- **Raft Voting**: The Raft algorithm requires an additional `vote <sender-id> <candidate-id>‘` command, in contrast to the Bully and Ring algorithms.

3.2.2 LEP Error Responses:

- Commands `elect`, `vote` and `declare` expect the right arguments and otherwise the Message Brokers responds with an error that contains the usage information (e.g., `error usage: elect <id>`).
- An unknown command is sent: `error protocol error` upon which the client socket should be closed.

3.2.3 Implementation details

- The server sends a `ok LEP` to each connecting client, as a greeting.
- Messages can be safely split by white space.
- We recommend to use a persistent TCP connection to send the heartbeat messages. In this example, the leader should connect to all its followers upon election and re-use this connection to continuously send `ping` and receive the `pong` reply.

Note for Election State Management

The provided automated-tests do not verify whether Message Brokers are always in the correct election state (i.e., `leader`, `follower`, or `candidate`). However, during the submission interview, we will check if the election states are correctly implemented, especially when working with Raft. We recommend using the provided `ElectionState` Enum for this purpose.

The Broker config is extended with the additional properties used for the election.

Relevant properties:

- `election.id` - The Message Brokers unique numerical ID.
- `election.type` - The Leader Election Algorithm the Message Broker should use.
- `election.port` - The network port under which the Leader Election Prototocol is handled.
- `election.domain` - The domain under which the Message Broker registers itself at the DNS Server, when it becomes a leader.
- `election.peer.hosts` - A comma-separated list of IP addresses of peer Message Brokers for the current Message Broker.
- `election.peer.ports` - A comma-separated list of ports of peer Message Brokers for the current Message Broker.
- `election.peer.ids` - A comma-separated list of peer Message Broker IDs for the current Message Broker.
- `election.heartbeat.timeout.ms` - Defines the timeout duration (in milliseconds) within which a leaders heartbeat message must be received.

The properties can be read using the provided `ConfigParser` and can be accessed via the `BrokerConfig` that is passed as an argument at the Message Broker Constructor.

3.3 Election Types

The Message Broker supports several different Leader Election Algorithms to determine which node in a distributed system should act as the leader. The following election algorithms/types are implemented: **None**, **Ring**, **Bully** and **Raft**.

3.3.1 None

In this mode, no Leader Election is performed. This may be used in single-node setups or when external mechanisms manage leader selection. Without a leader, all nodes operate independently, similar to Assignment 2.

3.3.2 Ring

The Ring Election is a simple distributed Leader Election Algorithm where nodes are organized in a logical ring. Each node communicates with its successor in the ring. The `LEP` command `elect <id>` initiates an election. In case that a leader fails, a peer node of the leader starts the election by sending `elect <id>` to its successor node. The node with the highest ID becomes the new leader.

Note: The `election.peer.ids` list is based on the ring order (i.e., `election.peer.ids[0]` is the successor).

Steps:

1. The current leader Message Broker fails.
2. The Message Broker with the lowest heartbeat timeout reaches its timeout first.
3. The Message Broker from the previous step starts an Leader Election by sending a `elect <id>` command to its successor in the ring, proposing itself as the leader.
4. The successor Message Broker forwards the `elect <id>` to its own successor, and so on.
5. Each Message Broker compares its ID to the one in the received `elect <id>` command. It sends `elect <id>` to its successsor containing the higher ID of the two compared IDs. In case the directly following successor is down/has failed, it sends the `elect <id>` command to the next available Message Broker in the ring.
6. The `elect <id>` command is passed until a Message Broker receives an `elect <id>` command that contains its own ID. This indicates that the `elect <id>` command has traveled around the entire ring and it has the highest ID out of all available Message Broker nodes. The node with the highest ID then declares itself among the others as the new leader by sending a `declare <id>` command to its successor node.
7. The `declare <id>` command is propagated around the entire ring until it returns to the leader (i.e. the original sender of the command), completing the election process.

Note on Ring Election

The algorithm you will implement differs slightly from the one presented in the lecture. Instead of passing a list of IDs, only a single ID is circulated among the candidates.

For the Submission Interview: Consider and evaluate the advantages and disadvantages of both approaches. Reflect on aspects such as communication overhead, memory usage, simplicity of implementation, fault tolerance, and the speed of the Leader Election.

Example:

Figure 3 shows an example election flow of the Ring Election Algorithm for this assignment:

1. *MB 4* sends the heartbeat (`ping`) to all its followers.
2. Later, *MB 4* goes offline. *MB 1* does not receive a `ping` in time and initiates the election by sending an `elect 1` command to *MB 2*.
3. *MB 2* compares the ID=1 of the received `elect 1` with its own ID=2. It propagates the higher ID via `elect 2` to its successor (*MB 3*).
4. *MB 3* checks the received `elect 2` command. It comapres the ID=2 with its own ID=3, keeping the higher ID. *MB 3* tries to send `elect 3` to *MB 4*, but fails, proceeding to send the message to *MB 1*, the next successor in the ring.
5. *MB 1* receives `elect 3` from *MB 3*. Because the received ID=3 is higher then its own ID=1, it just forwards the message to *MB 2*,
6. *MB 2* also forwards `elect 3` unchanged to *MB 3*.
7. *MB 3* stops the election, because its ID equals the one in the *elect* command, indicating that it has the highest ID in the ring. *MB 3* declares itself as the new leader by sending `declare 3` to its successor.
8. The successor *MB 4* is offline, so the *declare 3* command is sent to the next available peer *MB 1*.

9. *MB 1* respects the `declare 3` command and accepts *MB 3* as the next leader. It propagates the command to its successor *MB 2*.
10. *MB 2* follows the same procedure as *MB 1* and propagates the `declare 3` command.
11. *MB 3* stops the declaration, because its ID equals the one in the received `declare 3` command.

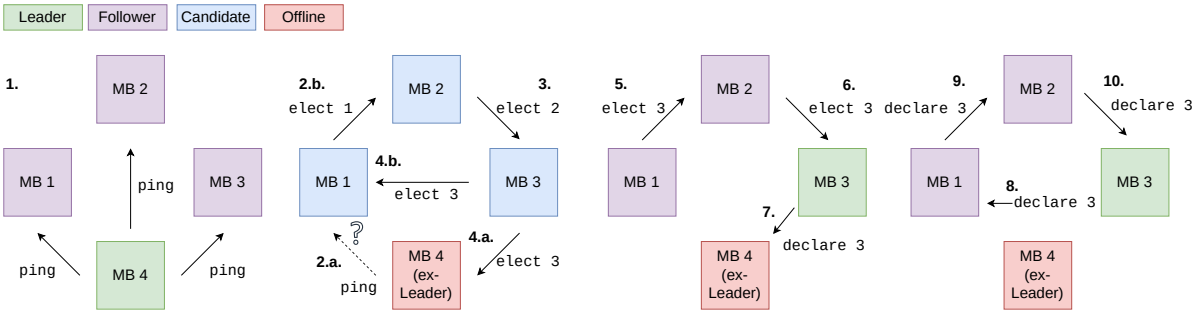

Figure 3: Ring Election

Helpful Links: YouTube Video - Ring Algorithm[4], Blog Post - Geeks for Geeks Ring Algorithm [5].

3.3.3 Bully

The Bully Election is another Leader Election Algorithm where each node can initiate an election and "bully" others with (strictly) lower IDs. When a node runs into a heartbeat timeout (i.e. detects a leader failure), it sends an `elect <id>` command to all nodes with (strictly) higher IDs.

If no node with higher ID responds, the initiating node declares itself as the leader by sending the `declare <id>` command to all peer nodes simultaneously. If a node with higher ID responds, the initiating node defers, allowing the nodes with higher IDs to take over and continue the election process.

Steps:

1. The current leader Message Broker fails.
2. The Message Broker with the shortest heartbeat timeout reaches its timeout first and initiates an election.
3. The election initiating Message Broker sends `elect <id>` to all nodes with higher IDs.
4. If no Message Broker with higher ID responds, the initiating Message Broker declares itself as the new leader by broadcasting `declare <id>` to all peer nodes.
5. If a Message Broker with higher ID responds, the initiating Message Broker defers, allowing the Message Brokers with higher IDs to continue the election process.

Example: Figure 4 shows an example election flow of the Bully Algorithm for this assignment. The scenario starts by the current leader (*MB 4*) suddenly going offline and *MB 1* realizing this as the first broker. Next come the following steps:

1. *MB 1* sends `elect 1` to all Message Brokers with a higher ID (i.e. *MB 2*, *MB 3*, *MB 4*).
2. *MB 2* and *MB 3* respond to the `elect` command with `ok`. Thus, triggering *MB 2* and *MB 3* to send their own `elect <id>` command to Message Brokers with higher IDs.
3. In this step, *MB 2* sends `elect 2` to *MB 3*, which responds with `ok`. *MB 3* sends `elect 3` to *MB 4*, which does not respond, making *MB 3* the only broker to not receive an `ok` as response, indicating that it has the highest ID.

[4] https://www.youtube.com/watch?v=eVs406wSyDo
[5] https://www.geeksforgeeks.org/what-is-ring-election-algorithm/

4. *MB 3* declares itself as the new leader by sending `declare 3` to each broker. To which *MB 1* and *MB 2* respond with `ack 1` and `ack 2` respectively (omitted from Figure 4).

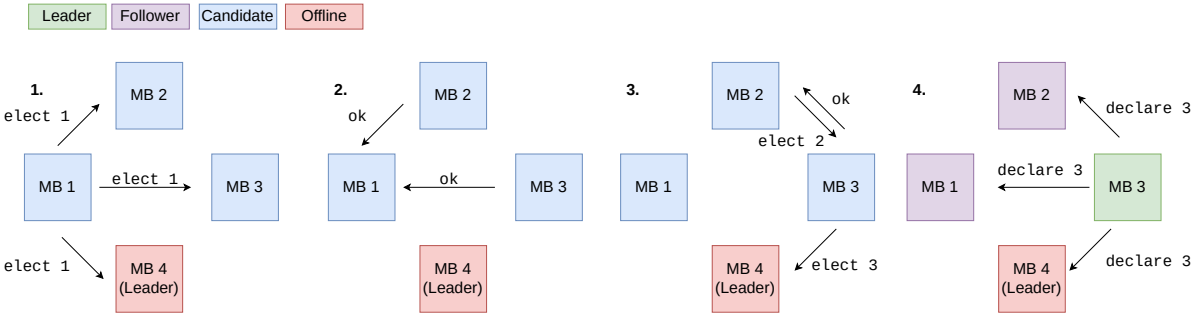

Figure 4: Bully Election

Helpful Links: Geeks for Geeks Bully Algorithm Blog [6]

3.3.4 Raft

The Leader Election approach of the Raft Consensus Algorithm is more complex compared Ring or Bully, but widely used for Leader Election in distributed systems. It ensures that a single leader is elected in a reliable and fault-tolerant manner. At any given time each server is in one of the following three states: `leader`, `follower`, or `candidate`. In normal operation there is exactly one leader and all of the other servers are followers. If the leader fails, followers become candidates and initiate an election. The election is won by the candidate that receives votes from a majority of nodes. Raft also uses a term-based approach for marking re-elections. Specifically, each Message Broker keeps track of how many elections have happened (i.e., terms), and uses this to reject potential votes of Message Brokers that are still in a previous term.

Raft States:

- **Leader:** The leader is always active. It sends sends heartbeats to followers, every x seconds, to let them know that the leader is still alive.
- **Follower:** Followers are passive. They issue no requests on their own but simply respond to requests from leaders and candidates. The expect a heartbeat message within their *election timeout*.
- **Candidate:** A candidate node initiates a new election by incrementing its term, voting for itself, and sending `vote <sender-id> <candidate-id>` commands to other nodes in the cluster. It waits for votes and either becomes the leader if it gains a majority, steps down if another leader is discovered, or restarts the election if no decision is reached.

Steps:

1. The leader continuously sends heartbeat messages to its followers.
2. Once a follower has not received a heartbeat message within the timeout duration, it detects the absence of a leader and transitions to the `candidate` state. This starts a new term, i.e. The Message Broker increments its term.
3. The candidate starts the election by requesting votes from all other Message Brokers, by sending `elect <id>`, causing the peers to also transition into the `candidate` state and increment their term.
4. A message broker casts a vote by responding to the `elect <id>` command via `vote <sender-id> <candidate-id>`, where `<sender-id>` is its own ID and`<candidate-id>` is the ID of the candidate the broker is voting for.
5. Each Message Broker can vote for only one candidate per term. The candidate each broker votes for is based on the first-come first-serve principle. Specifically each broker votes for the candidate

[6] https://www.geeksforgeeks.org/bully-algorithm-in-distributed-system/

that first sends an election request. If later on a another candidate also requests a vote, the Message Broker lets them know that they already voted for someone else. Example:

(a) *MB 1* becomes a candidate sends `elect <1>` to all peers, trying to become leader.

(b) *MB 2* votes for *MB 1* by responding with `vote 2 1`.

(c) *MB 3* who is also a candidate sends `elect <3>` to all peers.

(d) *MB 2* responds to *MB 3* with `vote 2 1`, indicating that it has already voted for *MB 1* this term.

6. A candidate tallies up all the votes it has received. If they received the majority ($totalVotes > numOfPeers/2$), the Message Broker declares itself as the new leader, by sending `declare <id>` to all peers. All other candidates switch to the follower role when receiving the `declare` command and accept the new leader.

Note on Raft Election

The Raft Leader Election Algorithm introduced in this Assignment is a slightly differs slightly from the original Raft Algorithm specifications, to simplify development. Specifically, the relevance of *terms* is reduced by not sending it when declaring a leader or heartbeats. That means, terms are only relevant for each individual candidate to vote for one candidate during the election phase. In contrast to the actual Raft Algorithm, leader declarations are always accepted, even if the internal term count does not match.

Example:

Figure 5 shows an example election flow of the Raft Election Algorithm for this assignment. The scenario starts by the current leader (*MB 4*) suddenly going offline and *MB 1* realizing this as the first broker. Next come the following steps:

1. *MB 1* sends `elect 1` to **all other** Message Brokers.
2. Because *MB 1* is the first Broker to request votes, every **peer** Message Broker responds with `vote <sender-id> 1`.
3. *MB 1* received 2/2 possible votes (a majority), indicating that it won the election and becomes the new leader. Afterwards *MB 1* declares itself as the leader by sending `declare 1` to all peers, to which *MB 2* and *MB 3* respond with `ack 2` and `ack 3` respectively (omitted from Figure 4).

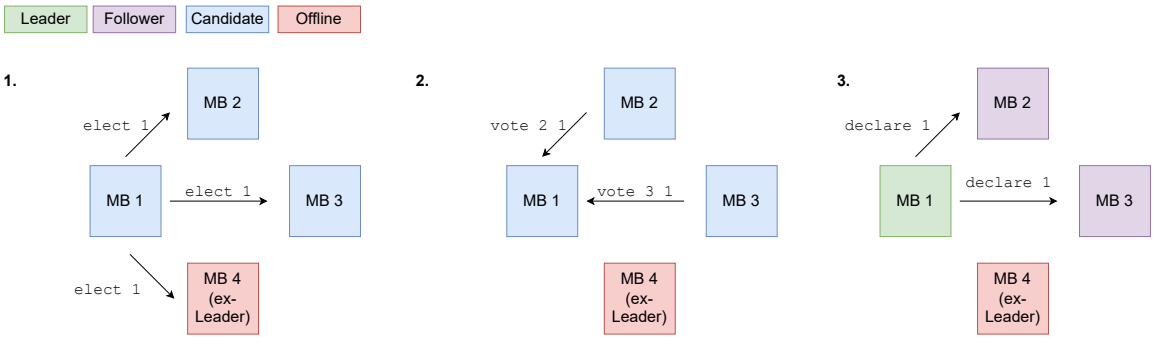

Figure 5: Raft Election

3.4 Implementation Details

- The Message Broker from Assignment 2 is to be expanded with the Leader Election capabilities.
- Use the `BrokerConfig` record passed via the constructor of the Message Broker for the startup configuration. If you start you server using `mvn exec:java@<componentId>` it will utilize the `ConfigParser` and read the config from the matching `<componentId>.properties` file.
- The `.properties` files are not utilize during the testing process. Rather the config is generated during runtime and passed via the `BrokerConfig` record in the constructor.
- The functions defined in the pre-defined interfaces have to be implemented. Namely:
  - `IServer`: `void shutdown()`
  - `IBroker`: `void getId()`, `void initiateElection()`, `int getLeader()`
  - `IDNSServer`: Extends from `IServer`
- The Message Brokers retains as much functionality as possible, even if some dependencies are offline. E.g. if the DNS Server is offline the Message Broker fails to register a DNS-entry, but it retains its other functionalities. Should the leader be unable to register itself with the DNS, it still remains as the active leader.

Helpful Links: Youtube Video - Raft[7], Raft Visualization [8], Blog Post - Raft Algorithm [9], Stanford Raft Basics[10], Raft on GitHub[11].

[7] https://www.youtube.com/watch?v=ro2fU8_mr2w
[8] https://raft.github.io/
[9] https://arorashu.github.io/posts/raft.html
[10] https://web.stanford.edu/~ouster/cgi-bin/cs190-winter20/lecture.php?topic=raft
[11] https://raft.github.io/

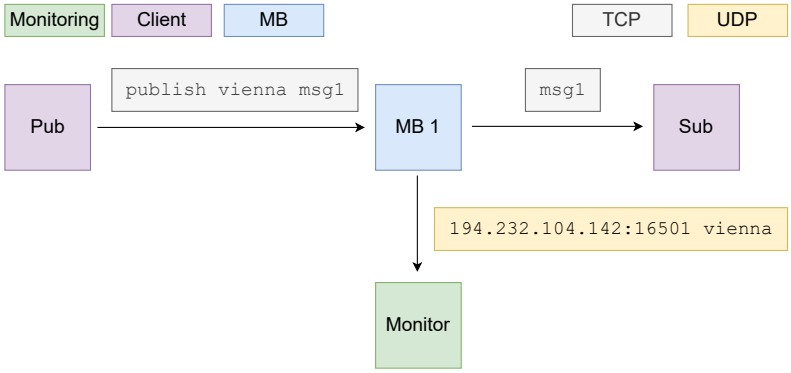

Figure 6: UDP Monitoring Example

4 Monitoring Server (UDP)

Implement a Monitoring Server using UDP. The server is designed to receive UDP packets containing log statistics about message exchanges and routing keys.

Key Requirements:

- Familiarize yourself with Datagrams, which enable UDP communication in Java[12].
- The monitoring server only receives packages, it never sends any.
- The Monitoring Server tracks the number of total received messages and specific statistics for each message-broker.

Process: When the message broker receives a `publish <routing-key> <message>` it sends the following monitoring data to the monitoring server: `<ip>:<port> <routing-key>`.

Consider the example from Figure 6. If the message broker (`194.232.104.142` listening on SMQP port `16501`) receives the message `publish vienna msg1`, it would send `194.232.104.142:16501 vienna` to the Monitoring Server via UDP. The Monitoring Server then stores this information internally.

Retrieving statistics: The Monitoring Server returns the stored statistics via the function `getStatistics()` in the following format:

```
Server <hostname>:<port>                                    1
     <routing-key> <count>                                  2
     <routing-key> <count>                                  3
Server <hostname>:<<port>                                   4
     <routing-key> <count>                                  5
...                                                         6
```

Example:

```
Server 194.232.104.142:16501                                1
     austria.vienna 27                                      2
     austria.linz 3                                         3
Server 142.251.36.238:16503                                 4
     usa.texas.houston 5099                                 5
...                                                         6
```

Additionally the Monitoring server implements the function `receivedMessages()` returning the number of received messages.

[12] https://docs.oracle.com/javase/tutorial/networking/datagrams/index.html

4.1 Implementation Details

- The Monitoring Server only operates via UDP and therefore does not require a TCP ServerSocket.
- Because UDP is a connection-less protocol, the same concurrency mechanisms (i.e., thread-per-connection) are not required. It is sufficient to read datagram packets within a single thread and update the data structures that hold the usage statistics.
- Implement the methods `int receivedMessages()` and `String getStatistics()` defined via the `IMonitoringServer` interface.
- Use the `MonitoringServerConfig` record passed via the constructor of the Message Broker for the startup configuration. If you start you server using `mvn exec:java@monitoring-0` it will utilize the `ConfigParser` and read the config from the matching `monitoring-0.properties` file.
- The `.properties` files are not utilize during the testing process. Rather the config is generated during runtime and passed via `MonitoringServerConfig` to the constructor.

Relevant Properties:

- `monitoring.port`: the port used for instantiating the `DatagramSocket`.

5 Automated Grading, Testing and Protected Files

This section offers a comprehensive guide on how to run the provided tests to verify the correctness of your assignment solution. It also gives an overview regards the grading process, including the criteria used for assessment. Additionally, it denotes "protected files and paths" that must remain unchanged, ensuring you avoid any point deductions due to modifications of restricted files.

5.1 Protected Files and Paths

Certain files and paths are "protected" and must stay untouched at any given time. Altering any of these will result in your repository receiving a permanent **non-removable** flag indicating unallowed changes, leading to point deductions. This restriction extends to creating, renaming, or deleting files or directories in these protected areas. The provided *.gitignore* file is designed to help prevent unintentional commits of unallowed changes.

5.1.1 Files and Paths you must leave unmodified

- .github/**/*
- src/main/resources/**
- src/test/**/*
- pom.xml

If you slightly modify protected files, we may replace your files with the original ones and re-run a Grading Workflow, expect point deduction in this case. In cases of major or severe modifications a full point deduction can be expected. Re-running the Grading Workflow can also be performed, when the code base does not properly reflect the Assignment Templates defaults.

It is your responsibility to ensure that none of the protected files or paths are modified. In the worst-case scenario, unallowed changes may result in a final score of 0 points.

5.2 Grading Workflow with GitHub

Grading is conducted entirely through GitHub via the *Grading Workflow*. You are required to submit your code solution to the assignment's remote repository created by GitHub Classroom using `git push`. Only pushes to the `main` branch will initiate a *Grading Workflow*. **Important:** We cannot accept any late submissions (i.e., hard deadline), so ensure your solution is pushed before the assignment deadline.

To start a *Grading Workflow*, push your (partial) solution to the `main` branch of the above stated GitHub repository. Each unit test is marked with a `@GitHubClassroomGrading` annotation, indicating the score achievable if the test-case result is successful. The score from the last completed *Grading Workflow* is the one that will count toward the final assignment grade.

Do not worry if GitHub displays a red cross - "failed", after the *Grading Workflow* has finished. You will still earn points for any test-cases that passed in the *Grading Workflow*. If all tests pass, you will see a checkmark indicating "success". In this case you will earn the maximum **score of 125**, which is then **translated to 12.5 points** (i.e., dividing by 10). The translated assignment points will be uploaded to TUWEL after the assignment deadline has passed.

Re-runs of the Grading Workflow: Re-runs of the *Grading Workflow* for the same commit do not transmit the results to GitHub Classroom. Therefore, your final assignment grade will only consider scores achieved from "regular" Workflow executions (i.e., triggered by commit and push to `main` branch).

Grading Workflow Limitation: Only one *Grading Workflow* can run at a time per assignment repository (i.e., per student). If you push a new code version to your repository while a *Grading Workflow* is still in progress, the "still in progress" Workflow will be cancelled and a new *Grading Workflow* for the new version will be added to the end of the waiting queue. This could lead to a longer wait time, since you may be queued behind a lot of students already waiting in the queue.

5.3 Test Suite and Local Testing

The JUnit test suite used for the Grading Workflow with GitHub is identical to the one provided in the assignment template. Since the infrastructure hosting the *Grading Workflow* has limited execution capacity, we highly recommend testing your solution on your local machine before pushing any changes to the remote repository's `main` branch.

You can run the JUnit test suite locally to check your progress and estimate the points you may earn for the coding part. Running the entire test suite will generate a **`"Grading Simulation Report"`**, which is printed directly to the end of the output console.

Use the following commands to verify your solution:

- `mvn test` (runs the entire test suite)
- `mvn test -Dtest="<testClassName>#<testMethodName"` (runs a specific test method)

5.4 Summary

The following highlights the necessary steps to verify your solution:

1. Commit and push your code solution to the `main` branch of the repository provided by GitHub Classroom.
2. Wait for the Grading Workflow to complete.
3. Check the score and calculate the points (i.e., by dividing through 10).
4. **Do not modify test files or any others mentioned above.**

6 Submission

Commit and push your code solution to the `main` branch of the repository provided by GitHub Classroom. The score achieved from the last completed "regular" *Grading Workflow* execution, is the one that will count toward the final assignment grade. Refer to Section 5.2 for more details.

6.1 Checklist

- ☐ Ensure that you have not altered any protected resources. Replace them with the original ones when you have accidentally altered them.
- ☐ Ensure that any merge request caused by an update of the Assignment Template is properly reflected in your repository and your final submission.
- ☐ Commit and push your solution ahead of the deadline.
- ☐ Check if the tests are passing as expected in the Grading Workflow.

6.2 Interviews

There is a dedicated Submission Interview for Assignment 3. The interviews will be conducted one-on-one with a tutor online via Zoom. You must register in time for a interview slot via TUWEL. The registration will be available from Tuesday, 14.01.2025 @18:00 and close on Saturday, 18.01.2025@18:00.

For any questions, please e-mail us via dslab@dsg.tuwien.ac.at.

Note for the Submission Interview

The submission interview is mandatory and failing to register in time or miss it without a valid excuse (e.g., illness) leads to a full point deduction for Assignment 3.