Sistemas Distribuídos

(ano letivo 2024'25)

Pratical Assigment 3 - Election Day

Today is election day, and `voters` gather at the `polling station` to cast their votes for one of two candidates. `Voters` must wait until the `polling clerk` opens the `polling station` before they can enter. Inside, there is a limited capacity for `voters`; if the station is full, those waiting outside must remain until space is available.

Before voting, each `voter` must present their voting ID, which is validated by the `poll clerk`. If the ID is confirmed as valid and has not been used for voting, the `voter` proceeds to the `e-voting booth` to cast their vote. The vote is randomized but skewed toward one of the candidates. Once voting is complete, the `voter` exits the `polling station`. A `voter` is allowed to vote only once; if the `poll clerk` detects a duplicate voting ID, the `voter` is asked to leave the `polling station`.

At the exit, there is an `exit poll` where some `voters` may be approached by a `pollster` for their opinion. The `pollster` selects a predefined percentage of `voters`—for example, 10%—to inquire about their votes. Responding to the `pollster` is optional, and `voters` are not obligated to disclose the truth about their vote. Their decision is governed by probability. For instance, 60% of approached `voters` may choose to respond, and among them, 20% may provide false information.

Once a `voter` completes their journey through the process (e.g., as illustrated in Figure 1), they may be "reborn" with either a new voting ID or the same ID, depending on probabilistic conditions. A reborn `voter` then re-enters the `polling station` as if they were a new arrival.

Election day terminates when the `polling clerk` announces its end. This can occur after, for example, 500 `voters` have participated or when a set time limit has been reached. The `poll clerk` then closes the `polling station` but allows all `voters` already inside to complete their votes. Once the station is empty, the `poll clerk` informs the `exit poll` that the `polling station` is closed and gathers the votes from the `e-voting booth`.

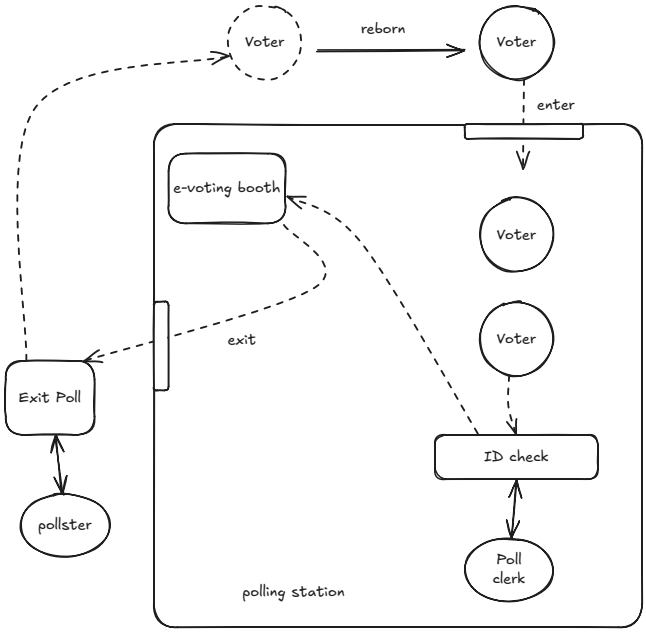

Figure 1: Example workflow of a `voter`.

Objectives and Requirements

Your task is to develop a simulation in Java that models the life cycle of `voters`, `poll clerk`, and the `pollster`. The simulation should be implemented in Java and run on a single platform under Linux using Java RMI.

Requirements

- The minimum number of `voters` is 3 and the maximum is 10. The number of concurrent `voters` should by pass to the program as an argument.
- The waiting queue, inside the `polling station`, has a minimum size of 2 and a maximum of 5. The queue size should by pass to the program as an argument.
- ID check and voting **do not** follow the entry order of the `voter` into the `polling station`; however, once `voters` arrive at the `ID check` or `e-voting booth`, their actions occur in arrival order.
- The ID validation by the `poll clerk` will take a random amount of time between 5 to 10 milliseconds to execute.
- Casting the vote by the `voter` will take a random amount of time between 0 to 15 milliseconds to execute.
- Responding to the `pollster` will take a random amount of time between 5 to 10 milliseconds to execute.

Additionally, you must implement a log file to track and describe the evolution of the system's internal state. A graphical user interface (GUI) is also required, providing a visual representation of the simulation and reflecting the internal state changes. The GUI can also be used to force the end of the simulation. To be able to visualize several stages of the simulation, you can scale the waiting times to an adequate value.

Guidelines for the implementation

1. **Message Structure Definition**
   For each representative server managing an information-sharing region, define the structure of the messages to be exchanged. Ensure that message formats are well-documented and support all required interactions with clients and other servers.
2. **Server Architecture Design**
   Specify the overall architecture of the servers. This should include the internal structure of each server, how it manages state and concurrency, and how it interfaces with other components in the system.
3. **Client Architecture Design**
   Specify the overall architecture of the clients. Describe how clients interact with servers, manage local state (if any), and handle communication and synchronization.
4. **Interaction Diagram**
   Develop an interaction diagram that concisely and accurately illustrates the dynamic behavior of your solution. This diagram should capture the main sequences of message exchanges and control flows between clients and servers. Iterate on steps 1–3 until the interaction model is complete and coherent.
5. **Implementation in Java**
   Implement the system in Java using specific reference data types. Ensure that your code

follows object-oriented design principles and makes appropriate use of Java RMI for remote method invocation between components.

6. **Deployment and Execution Scripts**
   Define the mapping of client and server components onto multiple nodes of the parallel execution environment. Provide shell scripts that automate the deployment and execution of all modules that comprise the application.

7. **Validation and Logging Analysis**
   Validate the correctness of your solution by executing multiple test runs. For each run, analyze the corresponding log file in detail to confirm that the internal state transitions and final outputs are consistent with expected behavior.