ESS 201 - Programming II (Java)
Mini-Project/Assignment 6
Due: Dec 26, 2021
Version 1.2

*This is an update to the assignment and incorporates changes to the assumptions and workings of the system. Changes relative to earlier versions are underlined for convenience, and also explicitly listed at the end of the document.*

**Fault-tolerant agreement**

A set of autonomous machines are moving on the streets of a city. As the machines move, they try to agree on the next action (change of direction) that they should perform. Let's assume this is a choice between turning left or right (relative to their current direction of motion), at periodic intervals. Machines communicate by sending messages to each other.

Since some of the machines can be faulty, the remaining machines (the correct ones) need to achieve the following objectives:

Objective 1: *All the correct machines agree on the next action*
Objective 2: *If a correct machine proposes an action, then all the correct machines agree on that action*

The simulation proceeds in phases. At the start of each phase, the player (or main):

1. Identifies up to **t** machines as the faulty machines. Note: none of the correct machines know which of the machines, if any, are faulty. They just know that there can be a maximum of **t** faulty machines.
2. selects one of the machines as the leader. Note: the leader can also be faulty.

Once selected, the leader decides on the next action (move right or left), and transmits that decision to all the machines (including itself).

Since the receiver of a message does not know if the sender is correct or faulty, machines run a multi-round protocol as described later to try to arrive at a consensus. At the end of the protocol, all correct machines are expected to have reached agreement as described in Objectives 1 and 2.

Once a machine believes it has reached agreement for that phase, it starts moving in that direction until a new decision is reached in the next phase.

By definition, a correct machine always sends the same (and correct) message to all machines, and specifically, if it is the leader, it sends its initial decision to all the machines. It also sends out its messages within a certain bounded time.

In general, a faulty machine, on the other hand, can be *malicious* and do anything it wants. It can send different messages to different machines, it can avoid sending messages to some machines, it can send multiple messages to a given machine, it can delay the sending of certain messages, etc. However, to simplify our solution, we will assume that the machines are not that malicious, and make the following critical assumptions:

- *The leader can be faulty, but still sends the same message to at least 2t+1 machines. That is, the leader may avoid sending a message to at most t machines. However, all messages sent out by the leader are the same.*
- *A faulty machine either stays silent for a round, or sends the same message to all machines. This message can be incorrect though.*
- *All messages are sent and delivered within a very short time and are delivered correctly (no loss, no tampering in transit, etc)*

To reach agreement, each machine executes the following protocol:

1. messages are sent in rounds, and each phase has rounds starting from 0, and increasing by one each time. To keep things simple, we assume each machine sends a message to itself too in each round.
2. A message is a method invoked by the sender on the recipient, and contains the id of the sender, the phase number, the round number, and the decision value. Even for faulty machines the source ID, phase number and round number should be correct and as per the protocol. Only the decision value may be incorrect.
3. However, it sends the same message to all machines. For simplicity, we will assume that the sender id is always filled in correctly.
4. The protocol is executed by each machine as follows:
    a. Round 0: The leader (and only the leader) sends its decision to all machines. Note that a faulty leader can send different messages to different machines.
    b. Round 1:
        i. each machine transmits the value it received in Round 0, to all machines.
        ii. each machine keeps track of all the messages it receives in this round, and maintains the count of messages of each type.
        iii. At the end of the round, it uses the message with the larger count as its decision for Round 2.
    c. Round 2:
        i. each machine sends to all machines the decision value it chose (as majority value) in Round 1
        ii. each machine confirms that it gets at least 2t+1 identical values in this round, and terminates the protocol, with this value as its decision. If not, it flags an error, by printing out an error message, and does not participate further in this phase.It can take any action it wants after that, including staying stationary

The above is a (highly simplified) example of what is known as the Byzantine Generals problem or Byzantine Fault Tolerance (BFT), a mechanism used in different applications including blockchains!

Model this as a Java program with the following structure:

**class Game:**
This is initialized with the list of machines, and the number of faulty machines - t. We can assume that the number of machines, n > 3t

It runs a series of phases. For each phase:

- it informs each machine whether it is correct or faulty (invoking a method setState on the machine), by randomly tagging t of them as faulty.
- It then chooses one of the machines as the leader, and informs that machine that it is the leader. The leader is then expected to initiate that phase by executing Round 0 as above.

The Game instance decides which are the faulty machines and which is the leader for each phase.

**class Machine** is an abstract class and is expected to implement methods:

- to be informed if it is faulty and if it is the leader.
- a method setMachines that is called before any phases, and provides the list of machines in the game. The position of a machine in this list is to be used as its sourceID when sending messages. The number of faulty machines ***(t)*** can be at most ⅓ of the total number of machines.
- sendMessage: which is invoked by a source machine that wants to send a message to this instance, during the multi-round protocol
- move: which is invoked by the base class at each time step
- other methods to return name, position etc
- Machines are initially at (0,0) and start moving in direction (0,1)

A **Main class** (contains **main**), which instantiates Game, the machines, and initiates each phase.

Each student should implement extensions of class Game and class Machine to conform to the above descriptions. Specifically, implement the derived class of Machine to behave as correct or faulty machines depending on the state set. Replace the “demo” package and “Machine_demo”, “Game_demo” with your implementations.

During the demo, the Game class of each student will be integrated with the Machine class of all the others in that group to create a simulation environment.

The base classes Game and Machine (in package common) are provided in Game.java and Machine.java, along with a Location class for convenience.

You can add any classes you need to support your implementation of derived classes of Game and Machine.

A sample Main is provided. This will be replaced with a main that is driven by a GUI.

Your derived class of Machine should be named as Machine_abcd, where abcd is the last 4 digits of your roll no. The getName method should return this “abcd” as its name

Changes in v1.2

1. The direction of turning - “left” or “right” - is with respect to the current direction of motion. We assume all machines start at (0,0) and are moving in the direction (0,1)
2. A leader who is faulty still sends the same message to at least 2t+1 machines. That is, the leader may avoid sending a message to at most t machines. However, all messages sent out by the leader are the same.
3. Class Machine has a new method setMachines that is called by main to inform the machine about the list of machines.
4. The position of a machine in the list passed in setMachines is it’s ID number (to be used as sourceID in sendMessages)
5. The number of faulty machines is intended to be less than ⅓ of the total number of machines, i..e $n > 3t$. Hence, the value of t can be assumed to be the largest integer that satisfies this relation.
6. Faulty machines can only change the message value. The sourceID, phase number and round number should be as per the protocol.
7. If a correct machine detects a possible error situation - e,g,,when it does not get 2t+1 messages in Round 2 - it prints out an error message, and can take any action it wants after that, including staying stationary
8.