## Guidelines

This exercise is divided into two tasks. The first task requires an investigation and implementation concerning the consensus problem in distributed systems. The second task focuses on applying Conflict-Free Replicated Data Types (CRDTs) to achieve eventual consistency. Implementations for both parts are required, and you must follow the guidelines outlined below.

1. The corresponding exercise session for this assignment is scheduled for 14.10.2025, 10:15-12:00, in room 05.002, Spiegelgasse 5.

2. Implement your solution using Python 3.

3. Implement your solution in the provided code template attached to the exercise. **Do not change** the file structure.

4. Your implementation will be evaluated by running your scripts with Python 3. Submit all script files and resources you have used. If your code has dependencies, provide a **very brief but clear manual** in a *README.md* file explaining how to execute your code.

5. The use of Large Language Models (LLMs), including ChatGPT, Deepseek, Qwen, Grok or similar tools, is strictly prohibited for solving problems directly. While you may utilize these tools to enhance your understanding or gain knowledge, relying on them to solve problems will be considered plagiarism and will result in a penalty with a scaling factor between 0 and 1 on your grade.

6. This task must be completed in **groups of four** and only **one** should submit the implementation, acting as the team spokesperson. All submissions must be made **exclusively on ADAM**; submissions sent via email or any other medium will not be accepted.

7. Your team, which was already formed and registered for Exercise 1, will be automatically registered for Exercise 2. Therefore, no further action is required. If you encounter any issues with your team registration, please email ali.ajorian@unibas.ch with the subject line **"FDS2025-teammembers-exercise2"**.

8. Sharing your solution with other teams is considered plagiarism for all members of both the sharing and receiving teams. In such cases, all participants involved will receive a **grade of 0** for the exercise.

9. Following grading, an interview will be scheduled and announced via email. All group members are required to attend together. Each member will be interviewed **individually** on questions from **both tasks**, assessing their theoretical understanding of the problem and solution, as well as their practical implementation work. Based on this interview, an individual contribution factor between 0 and 1 will be determined and applied to the team's grade to calculate the final individual grade.

## Task 1: Consensus Algorithm (15 Points)

In this task, you will practice implementing a simple consensus algorithm using a simulated distributed system. Consensus algorithms are crucial in distributed systems to ensure that multiple nodes agree on a single data value, even in the presence of failures. Understanding these algorithms is fundamental for building resilient and reliable systems.

We have provided a framework that simulates a distributed system of $N$ nodes, where each node is simulated by a separate thread. As these threads execute independently, they mimic the behavior of distinct computers in a distributed system. Communication between nodes is simulated using a globally shared buffer, which acts as the network channel. Nodes send messages by writing to this buffer and receive messages by reading from their designated section within it. Although it is necessary to use synchronization mechanisms to prevent race conditions, we have considered small values of $N$ for simplicity, where the likelihood of such issues is low enough for our pedagogical goals. Therefore, we have not incorporated synchronization primitives like locks or semaphores in our template code.

### Template Code

The template code provided with this exercise contains our simulator implementation. As shown in Listing 1, the *initialize* function takes the number of nodes $N$ as input, initializes the global list *nodes* with an instance of the *Node* class for each one, and then calls their *start* method to begin their individual threads. The global *buffer* object is also declared to store the simulated messages exchanged between nodes. It is implemented as a dictionary that maintains a list of **received** messages for each node. The dictionary's keys are the node IDs (integers from 0 to $N-1$). Each message is a tuple containing the data and a message type, allowing you to distinguish between different kinds of received messages.

Listing 1: Template Code: Part I

```python
import threading
import time
nodes = []
buffer = {}

def initialize(N):
    global nodes
    nodes = [Node(i) for i in range(N)]
    for node in nodes:
        node.start()
```

As shown in Listing 2, the class *Node* implements the behavior of each node in the simulator. Its constructor *__init__* takes the node's ID as input and registers its dedicated message queue in the global buffer. The *start* function, as mentioned previously, launches a new thread for the node. This thread executes the *run* function. The *run* function consists of a busy-waiting loop that periodically checks the node's queue in the global buffer. When a message is received, it is popped from the buffer and delivered to the node by calling the *deliver* method. A part of your task is to implement the logic inside the *deliver* function to handle different types of received messages. Regarding the *run* function, the following point is critical:

*All functions called from within run are executed in the node's own thread. Therefore, every* *behavior of a node must be triggered from its thread. This means you can either write the logic directly inside the run function or implement it in a separate function that is called by run.*

Additionally, a helper function named *broadcast* is provided. It takes a message type and its corresponding data, then broadcasts a tuple $(type, data)$ to all nodes by placing it into each node's respective queue in the buffer object.

The framework also supports node crashes, which disconnects a node from the network. The underlying cause, hardware or network failure, is treated as an implementation detail outside the scope of this model. Two functions, *crash* and *recover*, are provided to trigger the failure and subsequent recovery of a node, respectively.

Listing 2: Template Code: Part II

```python
class Node:
    def __init__(self,id):
        buffer[id] = []
        self.id = id
        self.working = True
        self.state = 'unknown'

    def start(self):
        print(f'node {self.id} started')
        threading.Thread(target=self.run).start()

    def run(self):
        while True:
            while buffer[self.id]:
                msg_type, value = buffer[self.id].pop(0)
                if self.working: self.deliver(msg_type,value)
            time.sleep(0.1)

    def broadcast(self, msg_type, value):
        if self.working:
            for node in nodes:
                buffer[node.id].append((msg_type,value))

    def crash(self):
        if self.working:
            self.working = False
            buffer[self.id] = []

    def recover(self):
        if self.working:
            buffer[self.id] = []
            self.working = True

    def deliver(self, msg_type, value):
        pass
```

## Consensus Algorithm Simulation

We implement a simple leader election algorithm as an instance of a consensus problem within our simulator, assuming that all the nodes are honest and do not behave byzentine. In leader election, each node can be in one of three states: follower, candidate, or leader.

- **Leader:** A node that has been elected by a majority during an election process.

- **Follower:** A node that follows the leader's decisions.
- **Candidate:** A temporary role a node assumes when there is no elected leader or when the current leader is presumed to have crashed.

To detect the crash of an elected leader, a *heartbeat* mechanism is used. The leader periodically sends 'heartbeat' messages. By receiving these heartbeats, followers confirm the leader is still active. If a follower does not receive a heartbeat for a specified timeout period, it infers that the leader has crashed.

## Your task

Your task is to implement the leader election process with the following assumptions.

Each node starts in the 'follower' state by default. When a node detects there is no leader or that the current leader has crashed, it decides to become a candidate and start a new election process.

The candidate starts the election process by broadcasting its ID to the network as a candidacy message. Since multiple followers may detect the need for a new election simultaneously, a potential candidate waits randomly for 1 to 3 seconds before broadcasting its candidacy to avoid simultaneous elections. During this waiting period, if the node receives a candidacy message from another node, it resigns its candidacy by not sending its own candidacy message.

If no other candidacy is received during the waiting period, the candidate broadcasts its candidacy message and enters a vote collection period of 2 seconds. After this period, it counts the total received votes. If it has obtained more than half of the votes, it becomes the leader, changes its state to 'leader', and starts broadcasting heartbeats.

When any node receives a candidacy message, regardless of whether it had detected the need for an election, if it has not yet voted, it will vote for the candidate by broadcasting a vote message containing both its voter ID and the candidate's ID. Note that each candidate votes for itself when announcing its candidacy.

During the entire leader election process, if any node receives a heartbeat, it stops the election process and accepts the heartbeat sender as the leader. The leader sends heartbeat messages every 0.5 seconds. If a node does not receive a heartbeat for one second, it infers that the leader has crashed.

In the following box, you will see a scenario involving 3 nodes executing the election algorithm, with node '0' being elected as the leader:

```
node 0 started
node 1 started
node 2 started
node 0 is starting an election.
node 0 voted to node 0
node 1 voted to node 0
node 1 detected node 0 as leader
node 0 detected node 0 as leader
node 2 voted to node 0
node 2 got a heartbeat and followed node 0 as leader
$ state
       node 0: leader
       node 1: follower
       node 2: follower
```

## Task2: CRDT (5 Points)

As you have seen in the lecture, Causal Length Set, CLS, allows adding and removing set elements without any coordination, yet the final state of the set in a decentralized context. Such a CRDT allows us to implement a shared shopping cart **without** servers or consensus algorithm! Image that Alice and Bob have a shared shopping list for the next grocery purchases. While on the road they update their shopping list and intermittently their devices synchronize with each other. The nice property of CLS is that items can be added and be removed several times by any party without coordination, yet all parties eventually agree on the final state.

**Your task** is to implement a CLS class in Python such that the following system trace can be executed.

Listing 3: A simulated shared shopping cart

```
 alice_list = CLS()
 bob_list = CLS()

 alice_list.add('Milk')
 alice_list.add('Potato')
 alice_list.add('Eggs')

bob_list.add('Sausage')
bob_list.add('Mustard')
bob_list.add('Coke')
bob_list.add('Potato')
bob_list.mutual_sync([alice_list])

alice_list.remove('Sausage')
alice_list.add('Tofu')
alice_list.remove('Potato')
alice_list.mutual_sync([bob_list])

print("Bob's list contains 'Potato' ?", bob_list.contains('Potato'))
```

## References and Useful Links

- Consensus: agree on some data value that is needed during computation in a distributed system.
- CRDT: conflict-free replicated data type