In this session, we will be dealing with the implementation of two leader election algorithms for rings of processors. We saw that any wave algorithm can be transformed to compute an infimum over all the nodes in a network. The ring algorithm in particular, can be used to compute the minimum between all process ranks. However, this algorithm is centralized and uses $N$ messages. For leader election, we want that at any time, any process can initiate an election and we want that every process can participate in the election (i.e. be candidate). To this purpose, we will see two decentralized leader election algorithms: LeLann’s algorithm (1977) and Peterson/Dolev-Klawe-Rodeh’s algorithm (1982).

### Part 1

#### LeLann’s algorithm

LeLann’s algorithm is described in Algorithm 1.

This algorithm uses a simple idea: each initiator in the algorithm will send a message with its identity through the whole ring and at the end compare all the identities received to determine a leader. Processes that are not initiators will just transfer received data to their neighbour in the ring. Once the leader has been chosen, we just propagate the information through the ring one more time, with messages with label "end".

**Algorithm 1** LeLann’s algorithm

1: $leader = -1$.
2: $min = p$.
3: if $p$ is an initiator then
4: Send $<tok,p>$ to next process.
5: Receive $<tok,q>$.
6: while $q \neq p$ do
7: $min \leftarrow \min(min, q)$
8: Send $<tok,q>$ to next process.
9: Receive $<tok,q>$.
10: if $p = min$ then
11: $leader = p$.
12: Send $<end, min>$ to next process.
13: else
14: Receive $<end,q>$.
15: if next process is not $q$ then
16: Send $<end,q>$ to next process.
17: $leader = q$.
18: else
19: while true do
20: if Receive $<tok,q>$ then
21: Send $<tok,q>$ to next process.
22: else if Receive $<end,q>$ then
23: if next process is not $q$ then
24: Send $<end,q>$ to next process.
25: $leader = q$.

As usual, we provide two codes `election-skeleton.c` and `election-solution.c` for this exercise. You are sup-
posed to provide your implementations in the given function skeletons in `election-solution.c`. The skeleton will automatically determine some initiators and check that all the returned values for every process are the same.

To compile the code, just use `mpicc election-skeleton.c -o election` as usual and then to run it, type:

```
mpirun -np [nodes] ./election [lelann | peterson] [all | any]
```

The first argument will select the algorithm to run and the last argument, if it is `all`, will make the program run with every process as an initiator, otherwise the last process will always be an initiator and some others could be initiator too with probability $\frac{1}{2}$.

**Question 1**

a) Implement LeLann’s algorithm.
To distinguish the two different types of messages `<tok,q>` and `<end,q>`, you can use MPI tags. Remember that in the prototype of `MPI_Send` and `MPI_Recv`, there is a "tag" argument. You can use different values in your `MPI_Send` to distinguish the two messages. In the `MPI_Recv`, you can use the constant `MPI_ANY_TAG` to receive all the messages and then use the `MPI_TAG` member of the `MPI_Status` object returned by the function to know what kind of message was sent.

Be careful during your implementation as you will need to use `MPI_Isend` at some point!
Make sure that your algorithm does not deadlock when you use the argument `all`.

b) What is the message complexity of LeLann’s algorithm?

**Part 2**

**Peterson’s algorithm**

Peterson’s algorithm (also known as Dolev-Klawe-Rodeh’s algorithm) is described in Algorithm 2 and processes as follow: each process has a current identity (ci) and at each round it will compare its identity to its two active neighbours. If its identity is the minimum among the three, then the process continues, otherwise it becomes passive and can no longer participate in the election. Passive processes only forward the messages and all initiators are active at the beginning of the algorithm.

The problem is that we consider unidirectional rings of processors, thus it is easy to receive the identity of the previous (active) process but difficult to get the identity of the next (active) process. To overcome this problem, we change the virtual identity of the processes: let us consider three active processes p, q and r such that q is the next active neighbour of p and r is the next active neighbour of q. p will send its virtual identity to q and q will send its virtual identity to r in a first message exchange, then q will forward the identity of p to r. This way, r can compare the identity of q to its previous active neighbour (p) and its next active neighbour (r). Then r will change its virtual identity to q and become either passive if q was not the minimum of p, q, r or continue to the next round with its new identity.

The algorithm stops when the identity of the previous active process is the same as the current identity of a process, meaning it is the only active process, and the result of the election is then propagated.

**Question 2**

a) Implement Peterson’s algorithm. Again, you will need to use MPI tags to distinguish the three types of messages. Make sure that your algorithm does not deadlock when you use the argument `all`.

b) What is the message complexity of Peterson’s algorithm?

**Question 3**

a) Use Simgrid to evaluate the efficiency of your algorithms using a unidirectional ring topology.

Make sure to backup all your implementations as they might be useful later on!
Algorithm 2 Peterson’s algorithm

1: \( ci \leftarrow p \)
2: \( acn \leftarrow -1. \)
3: \( leader \leftarrow -1. \)
4: if \( p \) is an initiator then
5: \( state \leftarrow active \)
6: else
7: \( state \leftarrow passive \)
8: while \( leader = -1 \) do
9: if \( state = active \) then
10: Send \( <one,ci> \) to next process.
11: Receive \( <one,q> \).
12: \( acn \leftarrow q. \)
13: if \( ci = acn \) then
14: Send \( <end,ci> \) to next process.
15: Receive \( <end,q> \).
16: \( leader \leftarrow ci. \)
17: else
18: Send \( <two,acn> \) to next process.
19: Receive \( <two,q> \).
20: if \( acn < ci \) and \( acn < q \) then
21: \( ci \leftarrow acn. \)
22: else
23: \( state \leftarrow passive. \)
24: else
25: Receive \( <one,q> \).
26: Send \( <one,q> \) to next process.
27: Receive \( m. \) //\( m \) is \( <two,q> \) or \( <end,q> \).
28: Send \( m \) to next process.
29: if \( m = <end,q> \) then
30: \( leader \leftarrow q. \)
31: return \( leader. \)