An Implementation of a Distributed Queue

We are going to implement a distributed queue based on the Multi-Paxos algorithm by modifying the files from the previous lab. In particular, this requires updating the following classes:

`Message` : use the message types and payloads relevant for Multi-Paxos

`Replica` : the `execute` method has to be modified to handle messages according to a node in Multi-Paxos. For storage, you can still use a HashMap where keys are `Integer` (and denote indices in the log) and values represent enqueue/dequeue invocations, with inputs for enqueues (you may define another class to represent such invocations).

The communication between the DDS and the clients also needs to be updated:

Inputs: In the previous lab, each invocation (except for Stop) is sent to a single replica. Now, every invocation is broadcasted to all replicas. A replica stores the invocations it receives (e.g., in a map from client ids to invocations). *We will assume that every client submits a single invocation.*

Outputs: An output (of a dequeue) is sent by a replica only when all the previous indices in the log are filled. This output is computed by simulating the sequence of methods on a "real" queue, e.g., `java.util.Queue`.

The channels between the DDS and clients should be of type FIFO. The channels between replicas should be of type Bag.

The leader of a round r is chosen as explained in the lecture, i.e. the replica with id $= r \bmod N$ where N is the total number of replicas. After a decision, the leader should continue with proposals until a random integer in the interval [0,1] becomes 0. Use the function `nextInt(1)` of the class `Random` to sample a random integer in the interval [0,1] (see the class `ChannelBag` for an example).

Test your implementations with multiple combinations of clients (as suggested in the Testing file). Be careful to correctness when messages are received in arbitrary orders.

Optional. Modify your implementation so that every client can submit multiple invocations. An invocation should be associated to an identifier that is a pair (`clientId`, `inv_number`) where `inv_number` is an integer that fixes an order between invocations of the same client (it is assumed that clients submit invocations with consecutive numbers). Be careful so that the invocations "proposed" in multi-Paxos respect the order between invocations of the same client (each proposer must check what invocations of a client were already decided before choosing a next one).