```java
import org.apache.hadoop.util.hash.Hash;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.StorageLevels;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.JavaPairDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import scala.Tuple2;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.Semaphore;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DistinctItemsExample {

    // After how many items should we stop?
    // public static final int THRESHOLD = 1000000;

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            throw new IllegalArgumentException("USAGE: port, threshold");
        }
        // IMPORTANT: the master must be set to "local[*]" or "local[n]" with n > 1, otherwise
        // there will be no processor running the streaming computation and your
        // code will crash with an out of memory (because the input keeps accumulating).
        SparkConf conf = new SparkConf(true)
                .setMaster("local[*]") // remove this line if running on the cluster
                .setAppName("DistinctExample");

        // Here, with the duration you can control how large to make your batches.
        // Beware that the data generator we are using is very fast, so the suggestion
        // is to use batches of less than a second, otherwise you might exhaust the
        // JVM memory.
        JavaStreamingContext sc = new JavaStreamingContext(conf, Durations.milliseconds(10));
        sc.sparkContext().setLogLevel("ERROR");

        // TECHNICAL DETAIL:
        // The streaming spark context and our code and the tasks that are spawned all
        // work concurrently. To ensure a clean shut down we use this semaphore. The
        // main thread will first acquire the only permit available, and then it will try
        // to acquire another one right after spinning up the streaming computation.
        // The second attempt at acquiring the semaphore will make the main thread
        // wait on the call. Then, in the `foreachRDD` call, when the stopping condition
        // is met the semaphore is released, basically giving "green light" to the main
        // thread to shut down the computation.

        Semaphore stoppingSemaphore = new Semaphore(1);
        stoppingSemaphore.acquire();

        // &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
        // INPUT READING
        // &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

        int portExp = Integer.parseInt(args[0]);
        System.out.println("Receiving data from port = " + portExp);
        int THRESHOLD = Integer.parseInt(args[1]);
        System.out.println("Threshold = " + THRESHOLD);

        // &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
        // DEFINING THE REQUIRED DATA STRUCTURES TO MAINTAIN THE STATE OF THE STREAM
        // &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

        long[] streamLength = new long[1]; // Stream length (an array to be passed by reference)
        streamLength[0]=0L;
        HashMap<Long, Long> histogram = new HashMap<>(); // Hash Table for the distinct elements

        // CODE TO PROCESS AN UNBOUNDED STREAM OF DATA IN BATCHES
        sc.socketTextStream("algo.dei.unipd.it", portExp, StorageLevels.MEMORY_AND_DISK)
                // For each batch, to the following.
                // BEWARE: the `foreachRDD` method has "at least once semantics", meaning
                // that the same data might be processed multiple times in case of failure.
                .foreachRDD((batch, time) -> {
                    // this is working on the batch at time `time`.
                    if (streamLength[0] < THRESHOLD) {
                        long batchSize = batch.count();
                        streamLength[0] += batchSize;
                        // Extract the distinct items from the batch
                        Map<Long, Long> batchItems = batch
                                .mapToPair(s -> new Tuple2<>(Long.parseLong(s), 1L))
                                .reduceByKey((i1, i2) -> 1L)
                                .collectAsMap();
                        // Update the streaming state
                        for (Map.Entry<Long, Long> pair : batchItems.entrySet()) {
                            if (!histogram.containsKey(pair.getKey())) {
                                histogram.put(pair.getKey(), 1L);
                            }
                        }
                        // If we wanted, here we could run some additional code on the global histogram
                        if (batchSize > 0) {
                            System.out.println("Batch size at time [" + time + "] is: " + batchSize);
                        }
                        if (streamLength[0] >= THRESHOLD) {
                            stoppingSemaphore.release();
                        }
                    }
                });

        // MANAGING STREAMING SPARK CONTEXT
        System.out.println("Starting streaming engine");
        sc.start();
        System.out.println("Waiting for shutdown condition");
        stoppingSemaphore.acquire();
        System.out.println("Stopping the streaming engine");

        // NOTE: You will see some data being processed even after the
        // shutdown command has been issued: This is because we are asking
        // to stop "gracefully", meaning that any outstanding work
        // will be done.
        sc.stop(false, false);
        System.out.println("Streaming engine stopped");

        // COMPUTE AND PRINT FINAL STATISTICS
        System.out.println("Number of items processed = " + streamLength[0]);
        System.out.println("Number of distinct items = " + histogram.size());
        long max = 0L;
        for (Long key : histogram.keySet()) {
            if (key > max) {max = key;}
        }
        System.out.println("Largest item = " + max);
    }
}
```