```scala
package streaming

import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

object WordCount {

  private def updateStateFunc(newValues: Seq[Int], state: Option[Int]): Option[Int] = {
    val stateUpdated = newValues.nonEmpty

    if (stateUpdated) {
      // return new count if there is new values come
      val newCount = state.getOrElse(0) + newValues.sum
      Some(newCount)
    } else {
      // return old value if new value is empty
      Some(state.getOrElse(0))
    }
  }

  def main(args: Array[String]): Unit = {
    if (args.length < 2) {
      System.err.println("Usage: WordCount <input_directory> <output_directory>")
      System.exit(1)
    }

    // set log level to WARN
    StreamingLogs.setStreamingLogLevels()

    val inputPath = args(0)
    val baseOutputPath = args(1)
    val sparkConf = new SparkConf().setAppName("WordCount").setMaster("local[*]") // use as many as core for streaming

    val ssc = new StreamingContext(sparkConf, Seconds(3)) // interval of 3 seconds
    ssc.checkpoint(".")
    val pattern = "[A-Za-z]{3,}".r  // pattern for accepted words
    try {
      val lines = ssc.textFileStream(inputPath)  // monitoring directory from args(0)
      val nonEmptyLines = lines.filter(line => line.nonEmpty) // to filter empty rdd
      val words = nonEmptyLines.flatMap(_.split(" ")) // to split line into words split by space
      // every word is match the pattern in the spec
      // i.e. characters only & <3 characters
      val filteredWords = words.filter { word => pattern.pattern.matcher(word).matches() }

      // Task 1
      // count words with reduceByKey
      val wordCounts = filteredWords.map(x => (x, 1)).reduceByKey(_ + _)
      wordCounts.print()

      var batchCounter = 0  // counter for output folder
      wordCounts.foreachRDD { wordCount =>
          if (!wordCount.isEmpty()) {
            batchCounter += 1
            val counter = f"$batchCounter%03d"
            val outputPath = s"$baseOutputPath/taskA-$counter"
            wordCount.saveAsTextFile(outputPath) // export file to directory from args(1)/taskA-00X
          } else {
            println(s"wordCount empty. Counter: $batchCounter")
          }
      }

      // Task 2
      def coOccurrence(line: String) = {
        val words = line.split(" ")
        val filteredWords = words.filter { word => pattern.pattern.matcher(word).matches() }
        val pairs = for {
          (word1, index1) <- filteredWords.zipWithIndex
          (word2, index2) <- filteredWords.zipWithIndex
          if index1 != index2 // use index instead of values for duplicated words
        } yield (word1, word2)
        pairs // return a pair of word1 and word2 as a tuple
      }
      val coOccurrencePairs = nonEmptyLines.flatMap(coOccurrence)

      val coOccurrenceCounts= coOccurrencePairs.map(pair => (pair, 1))
        .reduceByKey(_ + _)

      coOccurrenceCounts.print()
      coOccurrenceCounts.foreachRDD { coOccurrenceCount =>
        if (!coOccurrenceCount.isEmpty()) {
          val counter = f"$batchCounter%03d"
          val outputPath = s"$baseOutputPath/taskB-$counter"
          coOccurrenceCount.saveAsTextFile(outputPath)  // export file to directory from args(1)/taskB-00X
        }
      }
      // Task 3
      // use updateStateByKey to store the of co-occurrence count
      val coOccurrenceCountsState = coOccurrenceCounts
        .updateStateByKey(updateStateFunc)

      coOccurrenceCountsState.print()
      coOccurrenceCountsState.foreachRDD { coOccurrenceCount =>
        if (!coOccurrenceCount.isEmpty()) {
          val counter = f"$batchCounter%03d"
          val outputPath = s"$baseOutputPath/taskC-$counter"
          coOccurrenceCount.saveAsTextFile(outputPath)  // export file to directory from args(1)/taskC-00X
        }
      }
    } catch {
      case e: Exception =>
        // Handle the exception
        println(s"Error processing batch: ${e.getMessage}")
    }
    ssc.start()
    ssc.awaitTermination()

  }
}

import org.apache.log4j.{Level, Logger}
import org.apache.spark.internal.Logging

/** Utility functions for Spark Streaming examples. */
object StreamingLogs extends Logging {

  /** Set reasonable logging levels for streaming if the user has not configured log4j. */
  def setStreamingLogLevels(): Unit = {
    val log4jInitialized = Logger.getRootLogger.getAllAppenders.hasMoreElements
    if (!log4jInitialized) {
      // We first log something to initialize Spark's default logging, then we override the
      // logging level.
      logInfo("Setting log level to [WARN] for streaming example." +
        " To override add a custom log4j.properties to the classpath.")
      Logger.getRootLogger.setLevel(Level.WARN)
    }
  }
}
```