**Introduction**

The program is a **minimal real-time word counter**. It listens to a TCP socket on your machine (localhost:9999). Anything you type into that socket (using nc/netcat) is treated as a stream of text lines. Every **2 seconds**, Spark bundles whatever lines arrived during that time into a **micro-batch**, splits them into words, counts the words **within that batch**, and prints the counts.

It’s intentionally simple: no state across batches, no windows, no files—just “read lines → split → count → print,” repeated every 2 seconds.

**Line-by-line explanation**

# task11\_socket\_streaming.py

# Minimal Spark Streaming word count from a TCP socket (Spark 1.6)

Comments: a filename hint and a one-line description.

from \_\_future\_\_ import print\_function

Compatibility for **Python 2** (Spark 1.6 ships with Python 2 on many distros). It makes print() behave like Python 3’s function form. If you run Python 3, it’s harmless.

from pyspark import SparkContext

from pyspark.streaming import StreamingContext

* SparkContext (often called **sc**) is the entry point to core Spark—manages executors, jobs, and RDDs.
* StreamingContext (often called **ssc**) adds **Streaming** on top of SparkContext, creating and managing **DStreams** (Discretized Streams).

# Create a SparkContext (app name shows in logs/UI)

sc = SparkContext(appName="Task11\_SocketStreaming")

Starts a Spark application named **Task11\_SocketStreaming**.

* If you don’t pass a master here, the master is taken from your submit command (e.g., --master local[2]), which is what we want.

# Create a StreamingContext with 2-second micro-batches

ssc = StreamingContext(sc, 2)

Creates a StreamingContext on top of that SparkContext with a **batch interval of 2 seconds**.

* Spark Streaming is micro-batch based: it groups incoming data every 2 seconds, generating one RDD per DStream per batch.

# Define the input stream: connect to TCP server on localhost:9999

lines = ssc.socketTextStream("localhost", 9999)

Defines an **input DStream** named lines that pulls text from a TCP server on port **9999** of the local machine.

* Under the hood, Spark starts a **receiver** that holds an open TCP client connection to localhost:9999.
* Every line you send to that port is appended to the current batch’s lines RDD.

# Split each incoming line into words

words = lines.flatMap(lambda line: line.split())

A **transformation** DStream: for each batch, take the lines RDD and apply flatMap.

* line.split() splits on whitespace, returning a list of words.
* flatMap flattens lists into a stream of individual words.
* Result: words is a DStream where each batch RDD contains all words seen in that 2-second window.

# Map each word to (word, 1)

pairs = words.map(lambda w: (w, 1))

Per-word mapping to a pair key/value shape that’s easy to reduce.

* In each batch, this yields an RDD like: [(hello,1), (spark,1), (spark,1), (streaming,1), ...].

# Reduce by key per batch to get counts

counts = pairs.reduceByKey(lambda a, b: a + b)

**Aggregates within the batch**: sums all the 1s for identical words.

* The reduce happens **separately for each batch**. There’s **no accumulation across batches** (that would require state, e.g., updateStateByKey or a window).
* So the counts reset every 2 seconds.

# Print the top few word counts for each 2-second batch

counts.pprint()

An **output operation** that prints each batch’s results to the driver console.

* Output operations (like pprint, saveAsTextFiles, foreachRDD) **trigger** the streaming computation. If you never call one, nothing runs.

# Start streaming and wait forever (until Ctrl+C)

ssc.start()

ssc.awaitTermination()

* ssc.start() begins the streaming receivers and the recurring 2-second job schedule.
* ssc.awaitTermination() blocks the driver so your program stays alive until you interrupt (Ctrl+C) or it’s stopped elsewhere.

**What happens at runtime (concrete example)**

Assume your socket server is running (nc -lk 9999) and you type:

hello

hello spark

spark streaming spark

Batch 1 (e.g., at Time: 16:09:20) might include the first line:

* lines = ["hello"]
* words = ["hello"]
* pairs = [("hello", 1)]
* counts = [("hello", 1)]
* pprint() prints:
* (u'hello', 1)

Batch 2 (next 2-second window, e.g., Time: 16:09:22), if your next two lines arrived in that window:

* lines = ["hello spark", "spark streaming spark"]
* words = ["hello","spark","spark","streaming","spark"]
* pairs = [("hello",1), ("spark",1), ("spark",1), ("streaming",1), ("spark",1)]
* counts = [("hello",1), ("spark",3), ("streaming",1)]
* pprint() prints:
* (u'hello', 1)
* (u'spark', 3)
* (u'streaming', 1)

If nothing arrives in a given 2-second interval, you’ll see just the time header; there are no rows to print for that batch.

**Key concepts (quick recap)**

* **DStream**: a sequence of RDDs, one RDD per batch interval.
* **Transformations** (flatMap, map, reduceByKey) define how to turn each batch’s input RDD into output RDDs.
* **Output operations** (pprint) make the pipeline actually run.
* **Stateless counting**: counts are per batch; they do **not** accumulate over time in this program.
* **Socket source**: handy for demos—Spark connects as a TCP client to whatever is listening on localhost:9999.
* **Two threads**: when you run, prefer --master local[2] so one thread can **receive** while the other **processes**.