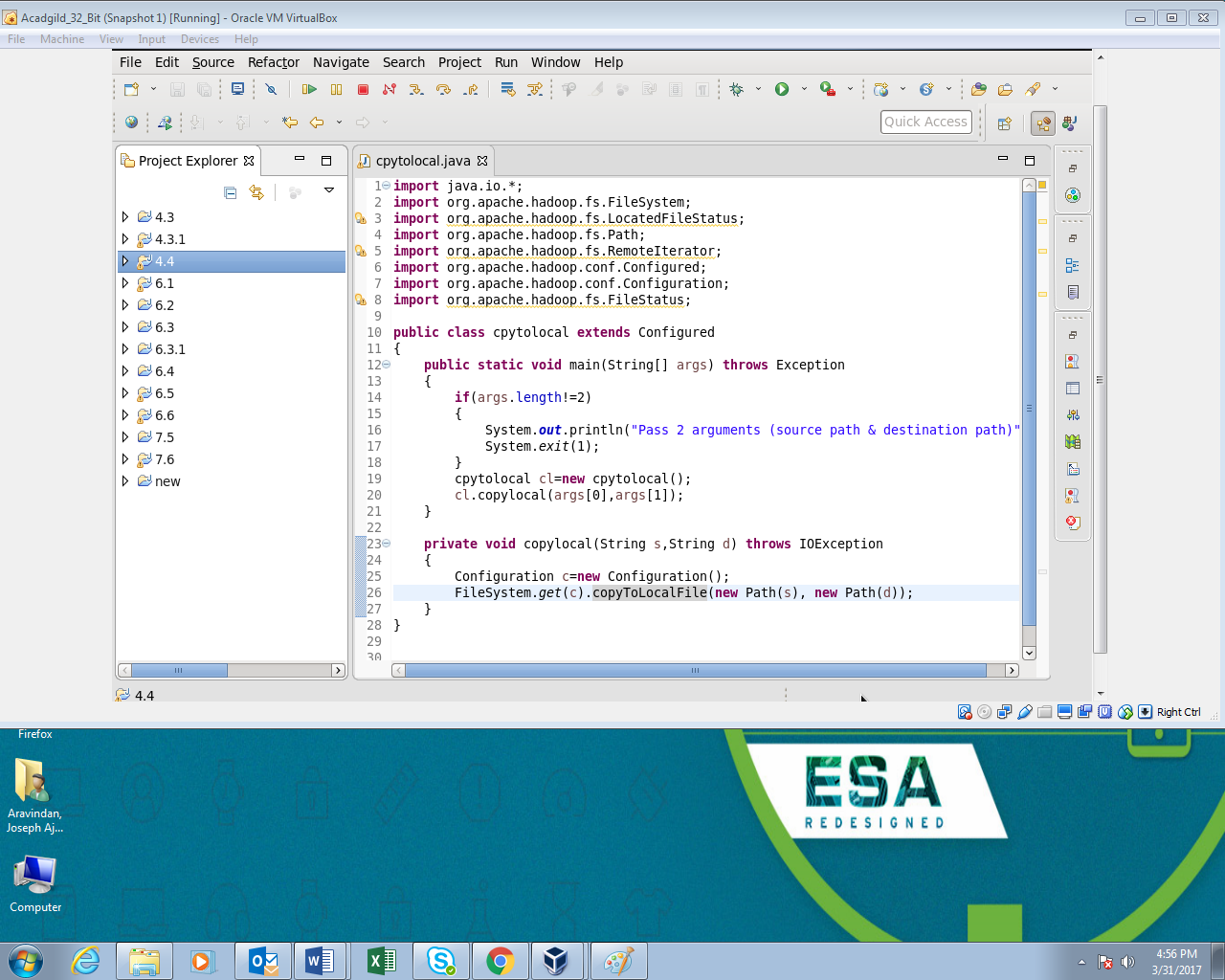
**Assignment 4.4**

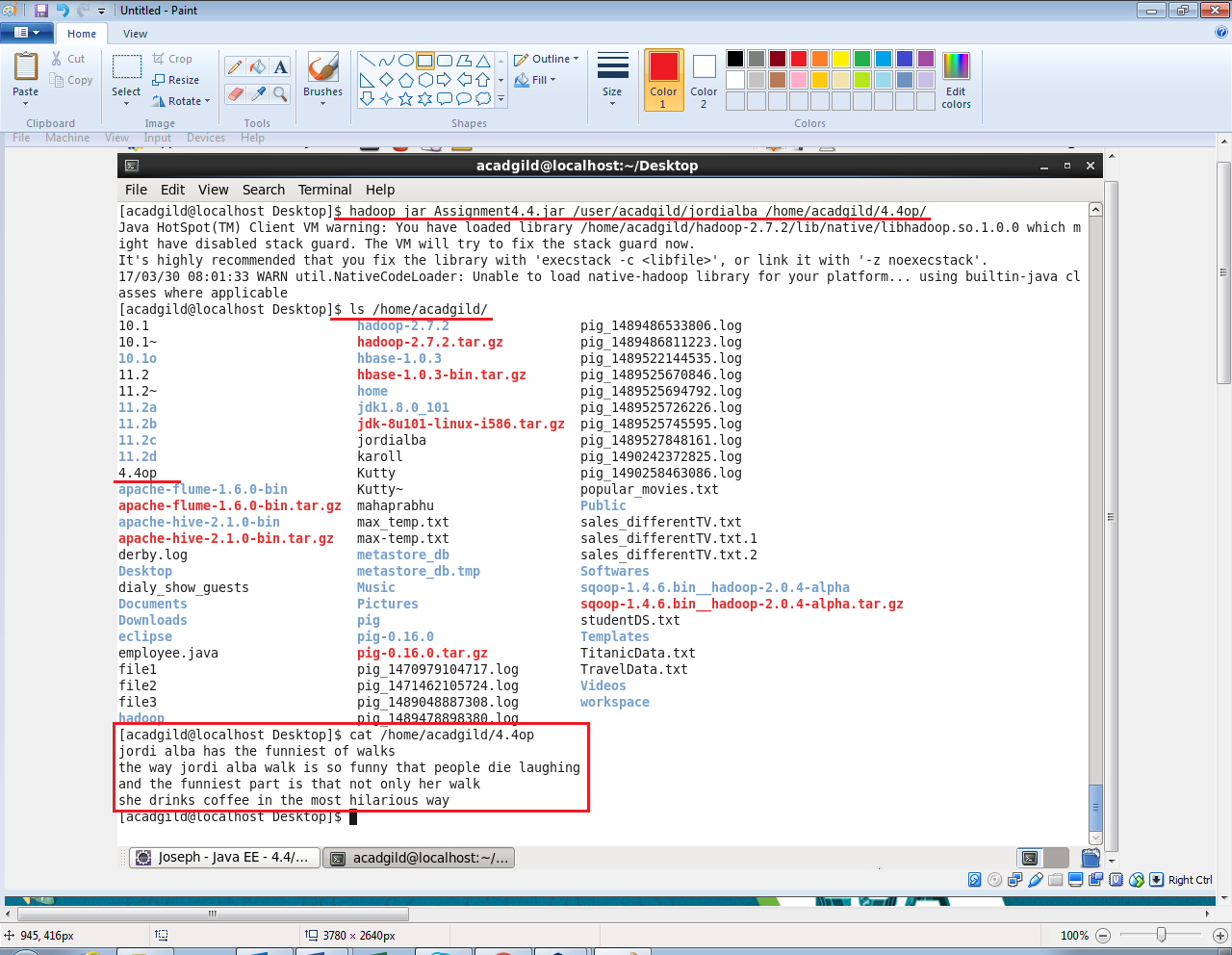
**Problem Statement 1:**

1. Write a Java program to copy a file from HDFS to LFS.

**Program:**



**Output:**



**Problem Statement 2:**

Explain the importance and usage of the below terms in details

● DFSInputStream

● DFSOutputStream

● FSDataInputStream

● FSDataOutputStream

**Answer:**

**1 DFSInputStream:**

DFSInputStream provides bytes from a named file. It handles negotiation of the namenode and various datanodes as necessary.

A stream that 'wraps' another stream it means that any operation done on a wrapper stream is in turn transformed into operations into the wrapped streams. So a DataInputStream can offer data semantics by reading the wrapped InputStream bytes and interpreting them according to the serialization rules for Java primitives.

**2. DFSOutputStream:**

DFSOutputStream creates files from a stream of bytes. The client application writes data that is cached internally by this stream. Data is broken up into packets, each packet is typically 64K in size. A packet comprises of chunks. Each chunk is typically 512 bytes and has an associated checksum with it.

When a client application fills up the currentPacket, it is enqueued into dataQueue. The DataStreamer thread picks up packets from the dataQueue, sends it to the first datanode in the pipeline and moves it from the dataQueue to the ackQueue. The ResponseProcessor receives acks from the datanodes. When a successful ack for a packet is received from all datanodes, the ResponseProcessor removes the corresponding packet from the ackQueue. In case of error, all outstanding packets and moved from ackQueue. A new pipeline is setup by eliminating the bad datanode from the original pipeline. The DataStreamer now starts sending packets from the dataQueue

**3. FSDataInputStream:**

The open() method on FileSystem actually returns an FSDataInputStream rather than a standard java.io class. This class is a specialization of java.io.DataInputStream with support for random access, so you can read from any part of the stream:

package org.apache.hadoop.fs;

public class FSDataInputStream extends DataInputStream

implements Seekable, PositionedReadable {

//implementation elied

}

The Seekable interface permits seeking to a position in the file and provides a query method for the current offset from the start of the file (getPos()):

public interface Seekable {

void seek(long pos) throws IOException;

long getPos() throws IOException;

}

Calling seek() with a position that is greater than the length of the file will result in an IOException. Unlike the skip() method of java.io.InputStream, which positions the stream at a point later than the current position, seek() can move to an arbitrary, absolute position in the file.

**4. FSDataOutputStream:**

The create() method on FileSystem returns an FSDataOutputStream, which, like FSDataInputStream, has a method for querying the current position in the file:

package org.apache.hadoop.fs;

public class FSDataOutputStream extends DataOutputStream implements Syncable {

public long getPos() throws IOException {

// implementation elided

}

// implementation elided

}

However, unlike FSDataInputStream, FSDataOutputStream does not permit seeking. This is because HDFS allows only sequential writes to an open file or appends to an already written file. In other words, there is no support for writing to anywhere other than the end of the file, so there is no value in being able to seek while writing.