TCP Segment Simulation

Matthew Moretz

January 28, 2019

Abstract

This project was created to show characteristics of the Transport Layer as it is used in computer networking and recreate the structure and some of the functions of the Transmission Control Protocol (TCP) segment that is the basis of connection-based network communications. Anytime you connect to a website over HTTP(S), information is transmitted from the website to your local machine, and passes through numerous layers that handle such things as error checking and routing of information.

1 Important Notes on Project Structure

This program is not intended as a replacement or an efficient replication of how the TCP segment is handled and how the transport layer works, but instead takes liberties with functionality that is provided by high-level programming as well as the object-oriented abilities of the Java programming language. The structure of the TCP segment in this program is taken from the internet standard (RFC 793) used by the modern internet.

The project consists of a *JAR* file which can usually be run via command line using `java -jar <jar_name>.jar`

The program will then display some information about the sending and receiving hosts' transport layers, but the main focus is the code and how the TransportLayer object handles data in TCP segments

2 Using the program

The program accepts a long string of hex values starting with the pseudo-header of the TCP segment, immediately followed by the header of the TCP segment, and finally the data of the TCP segment all on the same line. An example is given below:

AAAA995555555555000600E012C45678FFFFFFFFBBBBBBBB603F1B3472345E
551D341234F2345678

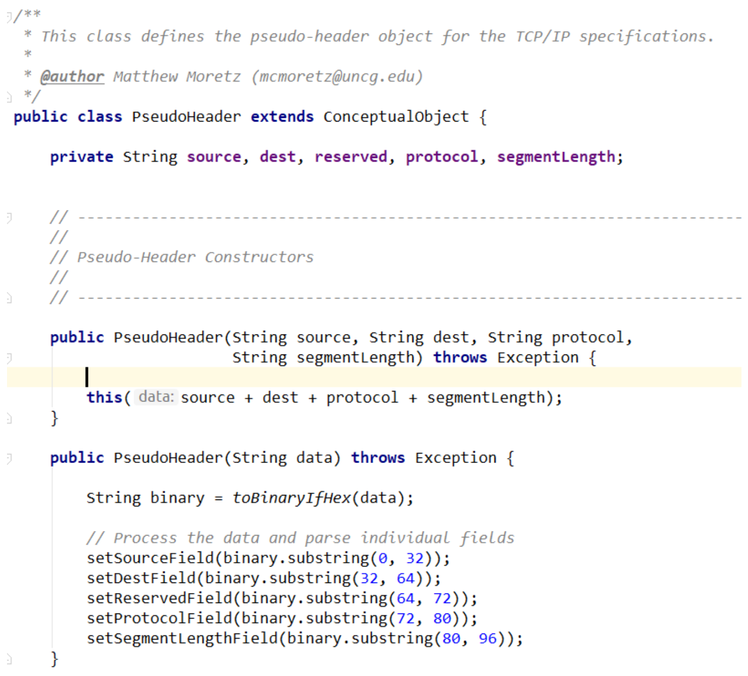
This program was originally designed as part of a class assignment for Computer Networks, including a GUI provided by the instructor's grader to interact with the model, but it was removed so this project would a snapshot of my work alone. Most of the methods are built to be public and safely accessible by other programs, and they are documented as well.

3 Source Code

This project involved taking an input of data (in the form of hexadecimal characters) to fill the fields of a TCP segment then assembling a segment object with fields based on the specifications of the Transmission Control Protocol. For the sake of this project, if there is an error in the input, an exception will be thrown describing what field was invalid or what was wrong and it will be thrown all the way back up to the calling method.

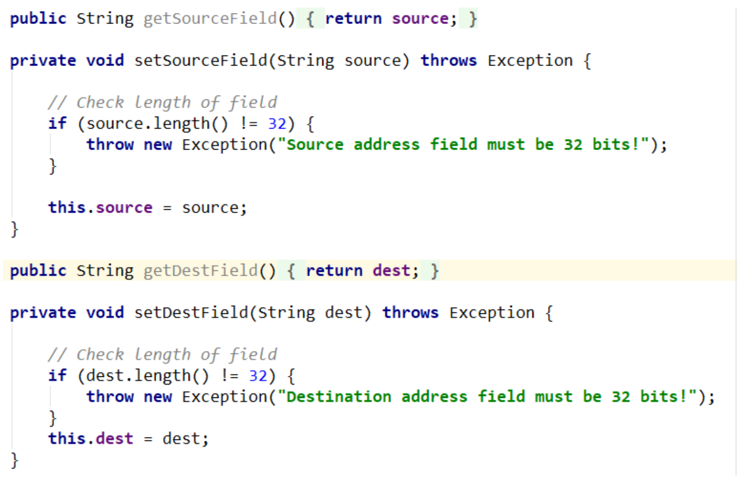
3.1 Pseudo-Header

When the TCP data to be sent is packaged into a TCP segment, it is ultimately passed down from the sender's transport layer to the network layer which places the entire contents of the TCP segment into the IP datagram's payload. In order to understand where the datagram is to be sent, it needs information that is provided by the TCP pseudo-header. The pseudo-header has fields for the source IP, destination IP, protocol (i.e. 6 for TCP), and the length of the TCP segment (not including the pseudo-header), as well as a reserved field which comes before the protocol field. The object I wrote takes in each of these values by parsing them out from a single string of hex values that is passed to it by the Segment object. I should note that the reason I extend a "ConceptualObject" is for demonstration purposes. The "ConceptualObject" class is an empty class that inherits all the properties of the Object class.



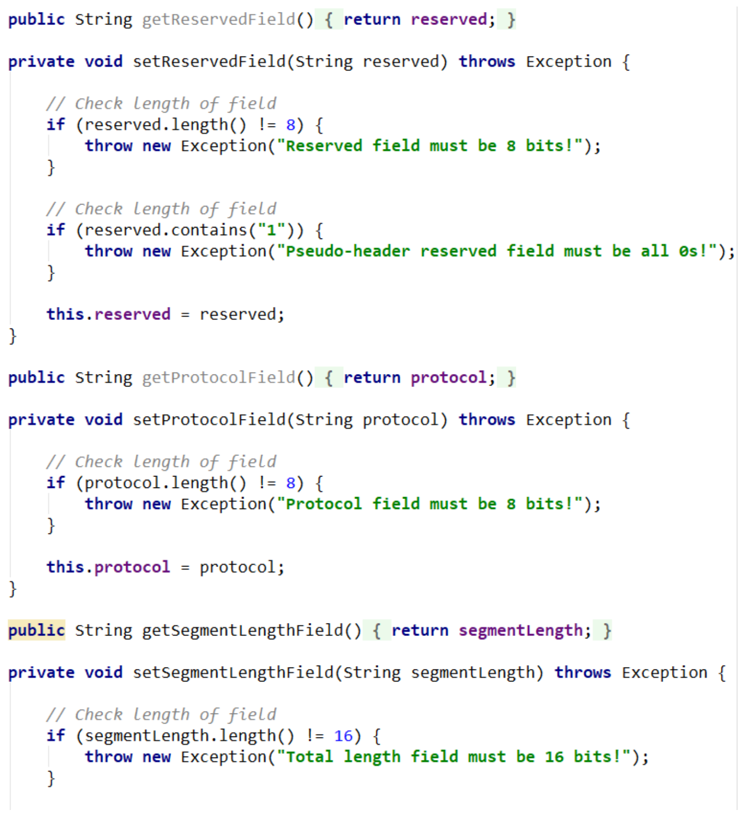
```java
/**
 * This class defines the pseudo-header object for the TCP/IP specifications.
 *
 * @author Matthew Moretz (mcmoretz@uncg.edu)
 */
public class PseudoHeader extends ConceptualObject {

    private String source, dest, reserved, protocol, segmentLength;


    // ------------------------------------------------------------------------
    //
    // Pseudo-Header Constructors
    //
    // ------------------------------------------------------------------------

    public PseudoHeader(String source, String dest, String protocol,
                        String segmentLength) throws Exception {

        this( data: source + dest + protocol + segmentLength);
    }

    public PseudoHeader(String data) throws Exception {

        String binary = toBinaryIfHex(data);

        // Process the data and parse individual fields
        setSourceField(binary.substring(0, 32));
        setDestField(binary.substring(32, 64));
        setReservedField(binary.substring(64, 72));
        setProtocolField(binary.substring(72, 80));
        setSegmentLengthField(binary.substring(80, 96));
    }
```

The source and destination IP fields must each be 32 bits in length.



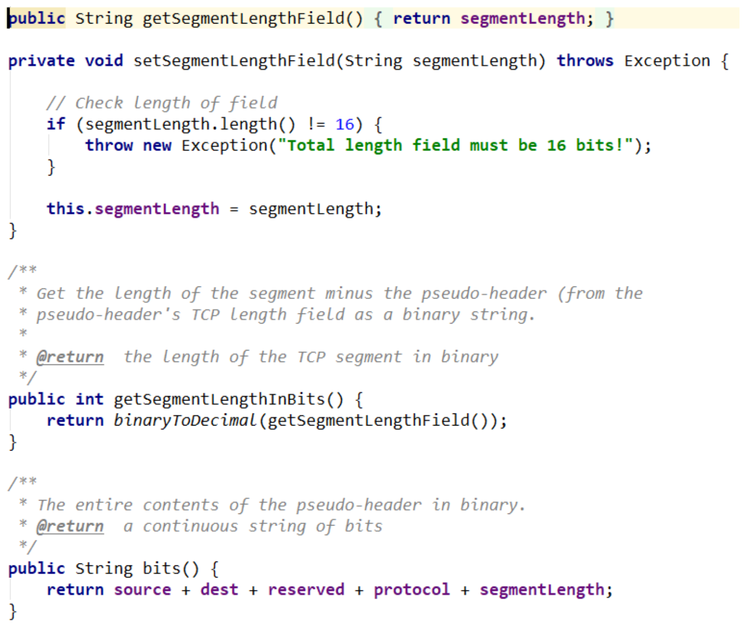
```java
public String getSourceField() { return source; }

private void setSourceField(String source) throws Exception {

    // Check length of field
    if (source.length() != 32) {
        throw new Exception("Source address field must be 32 bits!");
    }

    this.source = source;
}

public String getDestField() { return dest; }

private void setDestField(String dest) throws Exception {

    // Check length of field
    if (dest.length() != 32) {
        throw new Exception("Destination address field must be 32 bits!");
    }
    this.dest = dest;
}
```

The reserved and protocol fields only accept values 8 bits in length, with the requirement that the reserved field is all 0s according to the TCP specifications.



```java
public String getReservedField() { return reserved; }

private void setReservedField(String reserved) throws Exception {

    // Check length of field
    if (reserved.length() != 8) {
        throw new Exception("Reserved field must be 8 bits!");
    }

    // Check length of field
    if (reserved.contains("1")) {
        throw new Exception("Pseudo-header reserved field must be all 0s!");
    }

    this.reserved = reserved;
}

public String getProtocolField() { return protocol; }

private void setProtocolField(String protocol) throws Exception {

    // Check length of field
    if (protocol.length() != 8) {
        throw new Exception("Protocol field must be 8 bits!");
    }

    this.protocol = protocol;
}

public String getSegmentLengthField() { return segmentLength; }

private void setSegmentLengthField(String segmentLength) throws Exception {

    // Check length of field
    if (segmentLength.length() != 16) {
        throw new Exception("Total length field must be 16 bits!");
    }
```

Finally, the TCP segment length field is required to be 16 bits. There are also simple calculation methods used by the Segment and TransportLayer Objects in order to find information about the pseudo-header easily.

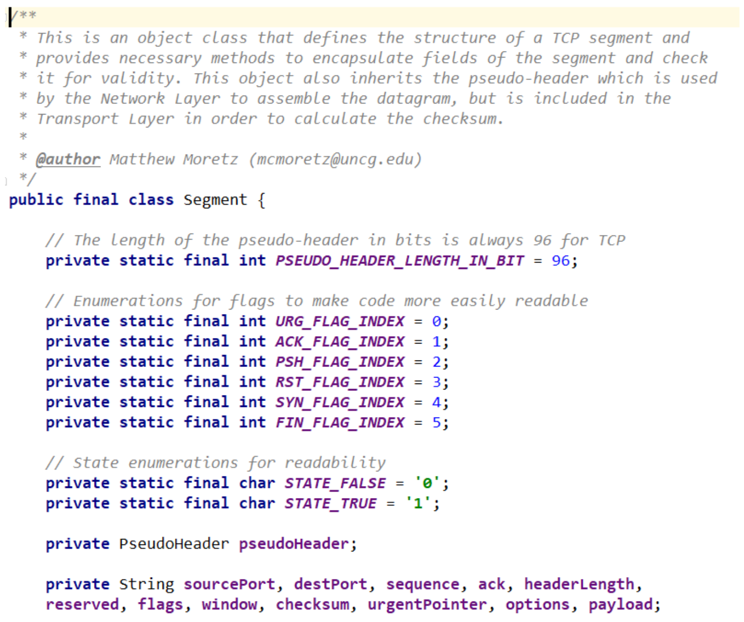


```java
public String getSegmentLengthField() { return segmentLength; }

private void setSegmentLengthField(String segmentLength) throws Exception {

    // Check length of field
    if (segmentLength.length() != 16) {
        throw new Exception("Total length field must be 16 bits!");
    }

    this.segmentLength = segmentLength;
}

/**
 * Get the length of the segment minus the pseudo-header (from the
 * pseudo-header's TCP length field as a binary string.
 *
 * @return  the length of the TCP segment in binary
 */
public int getSegmentLengthInBits() {
    return binaryToDecimal(getSegmentLengthField());
}

/**
 * The entire contents of the pseudo-header in binary.
 * @return  a continuous string of bits
 */
public String bits() {
    return source + dest + reserved + protocol + segmentLength;
}
```

This data is appended to the segment data for this project, but the transport layer only uses it in calculating the checksum (mentioned in detail later).

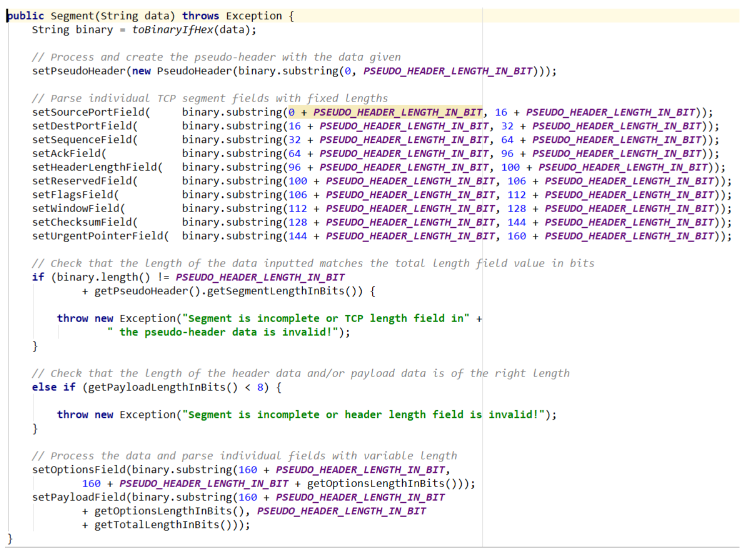
3.2 TCP Segment Header and Data

The TCP segment is assembled in the Segment class, and is parsed in much the same way as the pseudo-header. There are some enumerations in order to make it more readable and easier to follow along. This is the main portion of the code, with functions that handle the creation of the pseudo-header, processing of the checksum, and more.



```java
/**
 * This is an object class that defines the structure of a TCP segment and
 * provides necessary methods to encapsulate fields of the segment and check
 * it for validity. This object also inherits the pseudo-header which is used
 * by the Network Layer to assemble the datagram, but is included in the
 * Transport Layer in order to calculate the checksum.
 *
 * @author Matthew Moretz (mcmoretz@uncg.edu)
 */
public final class Segment {

    // The length of the pseudo-header in bits is always 96 for TCP
    private static final int PSEUDO_HEADER_LENGTH_IN_BIT = 96;

    // Enumerations for flags to make code more easily readable
    private static final int URG_FLAG_INDEX = 0;
    private static final int ACK_FLAG_INDEX = 1;
    private static final int PSH_FLAG_INDEX = 2;
    private static final int RST_FLAG_INDEX = 3;
    private static final int SYN_FLAG_INDEX = 4;
    private static final int FIN_FLAG_INDEX = 5;

    // State enumerations for readability
    private static final char STATE_FALSE = '0';
    private static final char STATE_TRUE = '1';

    private PseudoHeader pseudoHeader;

    private String sourcePort, destPort, sequence, ack, headerLength,
    reserved, flags, window, checksum, urgentPointer, options, payload;
```

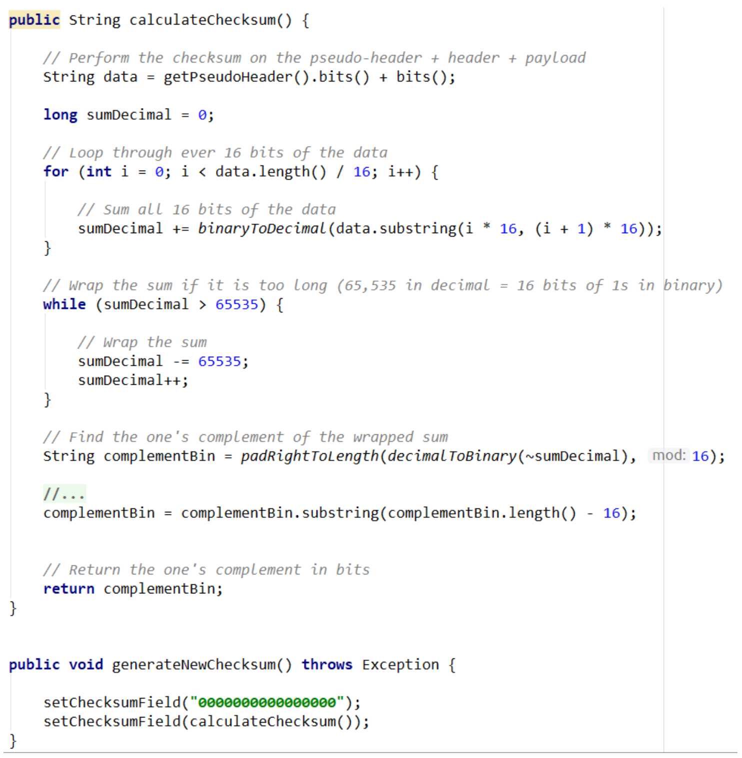
When the TransportLayer object is passed the data for the TCP segment, it first makes a new pseudo-header object and inputs the first 96 bits (constant PSEUDO_HEADER_LENGTH_IN_BIT) as an argument of the constructor, as mentioned above, the pseudo-header is assembled and each field is checked for validity according to the specifications of RFC 793. The remaining data is parsed out according to the requirements of the specifications of the actual TCP segment if the pseudo-header is valid (does not throw any exceptions). Notice that the object automatically checks that the TCP segment length field from the pseudo-header matches the length of the remaining bits for the TCP segment and that the data payload is at least 8 bits. Using the fields previously set, it calculates how long the length of the options must be using helper methods.



```java
public Segment(String data) throws Exception {
    String binary = toBinaryIfHex(data);

    // Process and create the pseudo-header with the data given
    setPseudoHeader(new PseudoHeader(binary.substring(0, PSEUDO_HEADER_LENGTH_IN_BIT)));

    // Parse individual TCP segment fields with fixed lengths
    setSourcePortField(     binary.substring(0 + PSEUDO_HEADER_LENGTH_IN_BIT, 16 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setDestPortField(       binary.substring(16 + PSEUDO_HEADER_LENGTH_IN_BIT, 32 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setSequenceField(       binary.substring(32 + PSEUDO_HEADER_LENGTH_IN_BIT, 64 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setAckField(            binary.substring(64 + PSEUDO_HEADER_LENGTH_IN_BIT, 96 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setHeaderLengthField(   binary.substring(96 + PSEUDO_HEADER_LENGTH_IN_BIT, 100 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setReservedField(       binary.substring(100 + PSEUDO_HEADER_LENGTH_IN_BIT, 106 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setFlagsField(          binary.substring(106 + PSEUDO_HEADER_LENGTH_IN_BIT, 112 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setWindowField(         binary.substring(112 + PSEUDO_HEADER_LENGTH_IN_BIT, 128 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setChecksumField(       binary.substring(128 + PSEUDO_HEADER_LENGTH_IN_BIT, 144 + PSEUDO_HEADER_LENGTH_IN_BIT));
    setUrgentPointerField(  binary.substring(144 + PSEUDO_HEADER_LENGTH_IN_BIT, 160 + PSEUDO_HEADER_LENGTH_IN_BIT));

    // Check that the length of the data inputted matches the total length field value in bits
    if (binary.length() != PSEUDO_HEADER_LENGTH_IN_BIT
            + getPseudoHeader().getSegmentLengthInBits()) {

        throw new Exception("Segment is incomplete or TCP length field in" +
                " the pseudo-header data is invalid!");
    }

    // Check that the length of the header data and/or payload data is of the right length
    else if (getPayloadLengthInBits() < 8) {

        throw new Exception("Segment is incomplete or header length field is invalid!");
    }

    // Process the data and parse individual fields with variable length
    setOptionsField(binary.substring(160 + PSEUDO_HEADER_LENGTH_IN_BIT,
            160 + PSEUDO_HEADER_LENGTH_IN_BIT + getOptionsLengthInBits()));
    setPayloadField(binary.substring(160 + PSEUDO_HEADER_LENGTH_IN_BIT
            + getOptionsLengthInBits(), PSEUDO_HEADER_LENGTH_IN_BIT
            + getTotalLengthInBits()));
}
```

Each of the fields of the TCP is validated according to its proper length similarly to how the pseudo-header fields were checked, throwing an error if something is inputted incorrectly. The source code is commented and written in long form to allow you to explore each field in the source code provided.

One important field is the checksum. The actual checksum field just checks for the proper length, but the methods provided below are very important. The way the object generates a checksum is by looping through the entire contents of the pseudo-header and TCP segment (header and data) and splits it up into sections of 16 bits. Each time, it converts the string of bits into a decimal number, then adds them to a running sum. Once the sum is calculate, it wraps the sum if it exceeds 16 bits (max of 65535 in decimal) and adds the carried bit to the sum. Finally, it takes the resulting sum and flips the bits using Java's bit-wise operation ( ) to do so and returns only the last 16 bits, since Java automatically flips all the bits of the long number, including the leading 0 bits, and we are only interested in the least significant 16 bit portion of the long number. If the resulting bits are all 0s, then the checksum is valid for the purpose of the checksum validity test. When making a new checksum, the checksum field is set to all 0s before computing the checksum and the resulting value becomes the checksum field's value.



```java
public String calculateChecksum() {

    // Perform the checksum on the pseudo-header + header + payload
    String data = getPseudoHeader().bits() + bits();

    long sumDecimal = 0;

    // Loop through ever 16 bits of the data
    for (int i = 0; i < data.length() / 16; i++) {

        // Sum all 16 bits of the data
        sumDecimal += binaryToDecimal(data.substring(i * 16, (i + 1) * 16));
    }

    // Wrap the sum if it is too long (65,535 in decimal = 16 bits of 1s in binary)
    while (sumDecimal > 65535) {

        // Wrap the sum
        sumDecimal -= 65535;
        sumDecimal++;
    }

    // Find the one's complement of the wrapped sum
    String complementBin = padRightToLength(decimalToBinary(~sumDecimal),  mod: 16);

    //...
    complementBin = complementBin.substring(complementBin.length() - 16);


    // Return the one's complement in bits
    return complementBin;
}


public void generateNewChecksum() throws Exception {

    setChecksumField("0000000000000000");
    setChecksumField(calculateChecksum());
}
```