ECE361 Lab 1
Message Transfer

Due Feb. 1, 2020 @ 11:59 PM

1 Introduction

A network address is a logical (e.g., IP address) or physical address (e.g., MAC address) that uniquely distinguishes a network node from other nodes on the network. It is usually a number that is assigned to any device or network interface card that seeks access to the network. A network node such as a personal computer or a mobile phone may also have several applications that use the node's connection to communicate with other applications across the network. To distinguish an application's data from other applications' data, the operating system allows applications to use port numbers. Therefore, each application's data will be mapped to a port number so they can simultaneously use the same address to communicate over the network. By analogy, each unit in an apartment building has its own unit number, but they all share the same building address.

To use a port number, applications are required to use a socket. A socket is one endpoint of a two-way communication link which is bound to a port number so that transport layer protocols can identify the application that the data is destined for. In this lab, you will first learn how to use a socket to transmit data across the network, and then you will use your knowledge to build a simple file transfer application. You will also see how headers are used to carry network addresses, which enable the network to direct data messages to their intended destination.

1.1 Lab Structure

This lab consists of two important tracks: a tutorial track and a challenge track. Each track has an important goal and a set of objectives that must be met. Please read the information below carefully and follow the steps that are involved to complete each goal and its objectives.

1.2 Lab Initialization

This lab includes starter code that uses a custom-built Python 3 library called `ece361`. A Python virtual environment will be created, that will include the library, where you will do all the work in this lab. Virtual environments must be "activated" before you attempt to run the code inside. We have provided an initialization script for you that will setup this virtual environment, along with the starter code, and the custom library.

It's highly recommended to update your VM before starting each lab. Open a terminal (in the VM), and type `ece361-update`

To run the initialization script for lab 1, open a terminal and type `ece361-lab-init 1`. You should see something similar to listing 1.

```
ubuntu@ece361:~$ ece361-lab-init 1
Finding available UG EECG host...
Warning: Permanently added 'ug251.eecg.utoronto.ca,128.100.13.251' (ECDSA) to the list of known
    hosts.
Creating working directory for lab 1 at /home/ubuntu/lab1
Creating Python3 virtual environment...
Installing libraries...

...

Done. Now run source /home/ubuntu/lab1/sourceMe to activate the virtual environment.
```

Listing 1: Initializing lab 1

The initialization script creates a working directory for this lab, located at `~/lab1`. Within that directory, there is a file called `sourceMe` which is used to start the virtual environment. To activate the virtual environment, run the `source` command as shown in listing 1. You should see a `(.venv)` appear in your prompt, confirming the environment is activated.

You will need to activate the virtual environment **in each terminal** that runs code belonging to this lab.

You should also notice two sub-directories under the working directory, named `track1` and `track2`. These provide the tutorial and starter code for the work you will do in sections 2 and 3, respectively.

2 Track 1: Tutorial

This is a tutorial track which explains the use of network addresses in guiding an application's data across the network. The goal of the track is to guide you on a step by step tutorial that allows you to experience the importance of addressing as well as show you how to work with the `ece361` library. You do not need to write code for this track. You only need to follow the steps to complete the track objectives. This track contains four important objectives which are designed in the form of individual modules.

Before continuing, it is highly important to first read the appendix (Section 6) that explains the `ece361` library and its `Socket()` class

2.1 Module 1

This module's objective is to explain the use of broadcasting in exchanging data between several participating network nodes. You will need three terminals with the virtual environment activated in each of them. Please follow the steps below to see an example.

1. Navigate to the `~/lab1/track1` directory and open `module1` on all three terminals.
2. For each of the Python files provided, you need to replace `<your_student_number>` with your student number. This is very important as multiple groups using the same number may cause

conflicts with each other's experiments. If you are working in team of two or more, use one of the team member's student number for all the modules.

3. You need to run an instance of `receiver.py` and an instance of `receiver2.py` in two of the three terminals. Run the instances by typing `python3 receiver.py` and `python3 receiver2.py` in your terminals. Make sure you have replaced your student number in both files.

4. In the third terminal, run an instance of `sender.py` by typing `python3 sender.py`. Make sure you have replaced your student number in the file.

5. Check if the receivers obtained the sender's message and that the sender address seen by both receivers is the same. You can find more information about the methods that are used in this module by examining the code and reading the appendix.

You must make sure you run the receiver applications described in step 3 before you run the sender in step 4.

The `From:` displays the sender address of the broadcast message. It is in the form of a list where the first element indicates the sender's address on the network. It may be a name that was randomly chosen, but it is subject to change at any time. Notice that the sender's address is the same in both receivers. That is an indicator that the same message is received by all the receivers.

The second element of the list indicates the port identifier of the sender node. If "N/A" is shown as the list's second element, that is in indication that the port identifier was not specified by the source (sender) node.

Note that if a message is broadcasted to every node on the network, it does not need to include any information about the message destination. That's why the `sendto()` method only requires the message as its only input argument for this module.

Do you think broadcasting is an efficient way to transmit data between the participant nodes on the network? Think about how scalable this approach can be as the number of participating nodes increases.

2.2 Module 2

This module's objective is to introduce the notion of a destination address when sending data to specific nodes on the network. A major drawback of broadcasting is that it does not scale when the number of nodes on the network increases. Therefore, by specifying the address of a destination node, the network can delivers data messages to that specific node, instead of broadcasting the messages to all the nodes on the network. Please follow the steps below to see an example.

Make sure that the lab virtual environment has been activated as specified in section 1.2.

1. Navigate to the `~/lab1/track1` directory and open `module2` on three terminals.

2. For each of the Python files provided, replace `<your_student_number>` with your student number.
3. In one terminal, run an instance of `receiver.py` by typing `python3 receiver.py`.
4. In another terminal, run an instance of `receiver2.py` by typing `python3 receiver2.py`.
5. In the last terminal, run an instance of `sender.py` by typing `python3 sender.py`.

You must make sure you run the receiver applications described in steps 3 and 4 before you run the sender in step 5.

As you can see, each of the receiver nodes will only receive the message that is destined to it. If you haven't already done so, open the sender and receiver files and examine the differences between them and their counterparts from module 1.

Now imagine that each node in the network will have several applications running on the same node. How can we provide a mechanism so that each application's data can be distinguished from other applications?

2.3 Module 3

This module's objective is to introduce the notion of port identifier to distinguish several applications' data from each other, and to allow them to simultaneously use the node's network address to communicate with each other applications across the network. Please follow the steps below to see an example.

Make sure that the lab virtual environment has been activated as specified in section 1.2.

1. Navigate to the `~/lab1/track1` directory and open `module3` on three terminals.
2. For each of the Python files provided, replace `<your_student_number>` with your student number.
3. In one terminal, run an instance of `receiver.py` by typing `python3 receiver.py`.
4. In another terminal, run an instance of `receiver2.py` by typing `python3 receiver.py`.
5. In the last terminal, run an instance of `sender.py` by typing `python3 sender.py`.

As you can see, both receivers share the same address but they use different port identifiers. Consequently, each receiver application only receives the data messages that is destined for it.

What if an application needs to establish bi-directional communication with another application? How can the receiver reply to the sender after it receives a message from it?

2.4 Module 4

This module's objective is to introduce the notion of source addresses, and how the receiver node can use this information to reply to the sender. Please follow the steps below to see an example.

Make sure that the lab virtual environment has been activated as specified in section 1.2.

1. Navigate to the `~/lab1/track1` directory and open `module4` on two terminals.
2. For each of the Python files provided, replace `<your_student_number>` with your student number.
3. In one terminal, run an instance of `receiver.py` by typing `python3 receiver.py`.
4. In another terminal, run an instance of `sender.py` by typing `python3 sender.py`.

As you can see, the sender node's application uses a port identifier to transmit a message that carries `"hi"` to the destination node application bound to a specific port identifier. The receiving application will reply to the source application by using the sender node's address and the sender's application port identifier. The receiver application can have access to the senders application port number and address via the `recvfrom()` method's return values.

Please see the appendix for further information on how to use `recvfrom()` to write your own applications.

3 Track 2: File Transfer

After showing different forms of transmitting data messages across the network, this track's goal is to complete a programming challenge that allows a sender application to open and read a file, and send the file's contents to a destination receiver application, where the file will be reassembled and written on the disk. To complete this assignment, you must navigate to the `~/lab1/track2` directory where the starter code is located.

The maximum amount of data that the sender can include in a message is 100 bytes. Therefore, the sender must divide the file into several chunks of at most 100 bytes, and transfer them one at a time to the destination. The receiver will then receive each chunk and reassemble the file.

Please do not forget to replace the `<your_student_number>` with your own student number, and to make sure that the lab virtual environment has been activated as specified in section 1.2.

3.1 Objective 1

Write the sender application by filling in the incomplete parts of `sender.py`. You need to complete the missing parts of the `send_file()` function and replace the input arguments with your own information. Even though the source code is commented to guide you, the following checkpoints are important to keep in mind.

- You can find the sender application in the `sender.py` file.
- A text file named `iso.txt` is provided to you with the starter code. The `send_file()` function must be able to open and read this file.
- The sender must first get the size of the `iso.txt` file and send that information to the receiver.
- The receiver will then acknowledge the successful receipt of the file size by sending an "ACK" message. See the comments in the starter code on how to handle acknowledgement messages.
- Upon successful transmission of the file size, the sender will read chunks of the file and send them sequentially to the receiver, until it reads to whole file.
- Each time the sender transmits a file chunk, the receiver must acknowledge receipt of the message and send back an acknowledgement to the sender. The sender can only transmit the next file chunk when it receives an acknowledgement.
- After a successful execution of the sender application (`sender.py`), the application displays the file size that was transmitted as well as the number of file chunks that were sent to the receiver application.

It is very important not to change the input argument for the `file_name` on both the sender and receiver applications.

3.2 Objective 2

Write the receiver's application by filling in the incomplete parts of `receiver.py`. You need to complete the missing parts of the `receive_file()` function and replace the input arguments with your own information. Even though the source code is commented to guide you, the following checkpoints are important to keep in mind.

- You can find the receiver application in the `receiver.py` file.
- The receiver will create a file named `iso_copy.txt` and write the incoming chunks to this file. This file name is used as an input argument when you call the `receive_from` function. Please do not alter this name as we want everybody to see the same sequence of events.
- The receiver is sending an acknowledgement message to the sender's application upon successful receipt of each file chunk. It also sends a "NACK" message which signals the sender to resend the same file chunk if it wasn't successfully received. See the comments in the starter code on how to handle the acknowledgement messages.
- The receiver must first receive the file size from the sender's application and use that information to know how much data to expect, and more importantly, when the file transmission has finished.

- After successfully completing the starter-code, a successful execution of the receiver application (`receiver.py`) will write a new file in the same directory as the original file. The new file should have the same size and content as the original file. The receiver application will also display the file size and the number of chunks that it received.

You must run the receiver application before running the sender application. Otherwise, it may not be successful in receiving all the file chunks.

You may experience a small delay in the data transmission between the sender and receiver applications. If you see that, do not worry and do not stop your code. This is due to the fact that the network is shared between all the students, which may cause some delay. Please be patient as it will resume transmission.

The following images demonstrate an example output after a successful execution of the receiver and sender applications.

```
(venv) ece361@ece361-VirtualBox:~/PycharmProjects/network-layering/solution/track2$ python3 sender.py
The file size: 5842
Segment: 1|100 Bytes is sent!
Segment: 2|100 Bytes is sent!
Segment: 3|100 Bytes is sent!
Segment: 4|100 Bytes is sent!
Segment: 5|100 Bytes is sent!
Segment: 6|100 Bytes is sent!
Segment: 7|100 Bytes is sent!
Segment: 8|100 Bytes is sent!
Segment: 9|100 Bytes is sent!
Segment: 10|100 Bytes is sent!
Segment: 11|100 Bytes is sent!
Segment: 12|100 Bytes is sent!
Segment: 13|100 Bytes is sent!
Segment: 14|100 Bytes is sent!
Segment: 15|100 Bytes is sent!
Segment: 16|100 Bytes is sent!
Segment: 17|100 Bytes is sent!
Segment: 18|100 Bytes is sent!
Segment: 19|100 Bytes is sent!
Segment: 20|100 Bytes is sent!
Segment: 21|100 Bytes is sent!
Segment: 22|100 Bytes is sent!
Segment: 23|100 Bytes is sent!
Segment: 24|100 Bytes is sent!
Segment: 25|100 Bytes is sent!
Segment: 26|100 Bytes is sent!
Segment: 27|100 Bytes is sent!
Segment: 28|100 Bytes is sent!
Segment: 29|100 Bytes is sent!
Segment: 30|100 Bytes is sent!
Segment: 31|100 Bytes is sent!
Segment: 32|100 Bytes is sent!
Segment: 33|100 Bytes is sent!
Segment: 34|100 Bytes is sent!
Segment: 35|100 Bytes is sent!
Segment: 36|100 Bytes is sent!
Segment: 37|100 Bytes is sent!
Segment: 38|100 Bytes is sent!
Segment: 39|100 Bytes is sent!
Segment: 40|100 Bytes is sent!
Segment: 41|100 Bytes is sent!
Segment: 42|100 Bytes is sent!
Segment: 43|100 Bytes is sent!
Segment: 44|100 Bytes is sent!
Segment: 45|100 Bytes is sent!
Segment: 46|100 Bytes is sent!
Segment: 47|100 Bytes is sent!
Segment: 48|100 Bytes is sent!
Segment: 49|100 Bytes is sent!
Segment: 50|100 Bytes is sent!
Segment: 51|100 Bytes is sent!
Segment: 52|100 Bytes is sent!
Segment: 53|100 Bytes is sent!
Segment: 54|100 Bytes is sent!
Segment: 55|100 Bytes is sent!
Segment: 56|100 Bytes is sent!
Segment: 57|100 Bytes is sent!
Segment: 58|100 Bytes is sent!
Segment: 59|42 Bytes is sent!
File transmission is completed!
[5842, 60]
```

Figure 1: The output of the sender application (`sender.py`)

```
(venv) ece361@ece361-VirtualBox:~/PycharmProjects/network-layering/solution/track2$ python3 receiver.py
5842
Acknowledged! --> Segment: 1|Amount of data received so far: 100
Acknowledged! --> Segment: 2|Amount of data received so far: 200
Acknowledged! --> Segment: 3|Amount of data received so far: 300
Acknowledged! --> Segment: 4|Amount of data received so far: 400
Acknowledged! --> Segment: 5|Amount of data received so far: 500
Acknowledged! --> Segment: 6|Amount of data received so far: 600
Acknowledged! --> Segment: 7|Amount of data received so far: 700
Acknowledged! --> Segment: 8|Amount of data received so far: 800
Acknowledged! --> Segment: 9|Amount of data received so far: 900
Acknowledged! --> Segment: 10|Amount of data received so far: 1000
Acknowledged! --> Segment: 11|Amount of data received so far: 1100
Acknowledged! --> Segment: 12|Amount of data received so far: 1200
Acknowledged! --> Segment: 13|Amount of data received so far: 1300
Acknowledged! --> Segment: 14|Amount of data received so far: 1400
Acknowledged! --> Segment: 15|Amount of data received so far: 1500
Acknowledged! --> Segment: 16|Amount of data received so far: 1600
Acknowledged! --> Segment: 17|Amount of data received so far: 1700
Acknowledged! --> Segment: 18|Amount of data received so far: 1800
Acknowledged! --> Segment: 19|Amount of data received so far: 1900
Acknowledged! --> Segment: 20|Amount of data received so far: 2000
Acknowledged! --> Segment: 21|Amount of data received so far: 2100
Acknowledged! --> Segment: 22|Amount of data received so far: 2200
Acknowledged! --> Segment: 23|Amount of data received so far: 2300
Acknowledged! --> Segment: 24|Amount of data received so far: 2400
Acknowledged! --> Segment: 25|Amount of data received so far: 2500
Acknowledged! --> Segment: 26|Amount of data received so far: 2600
Acknowledged! --> Segment: 27|Amount of data received so far: 2700
Acknowledged! --> Segment: 28|Amount of data received so far: 2800
Acknowledged! --> Segment: 29|Amount of data received so far: 2900
Acknowledged! --> Segment: 30|Amount of data received so far: 3000
Acknowledged! --> Segment: 31|Amount of data received so far: 3100
Acknowledged! --> Segment: 32|Amount of data received so far: 3200
Acknowledged! --> Segment: 33|Amount of data received so far: 3300
Acknowledged! --> Segment: 34|Amount of data received so far: 3400
Acknowledged! --> Segment: 35|Amount of data received so far: 3500
Acknowledged! --> Segment: 36|Amount of data received so far: 3600
Acknowledged! --> Segment: 37|Amount of data received so far: 3700
Acknowledged! --> Segment: 38|Amount of data received so far: 3800
Acknowledged! --> Segment: 39|Amount of data received so far: 3900
Acknowledged! --> Segment: 40|Amount of data received so far: 4000
Acknowledged! --> Segment: 41|Amount of data received so far: 4100
Acknowledged! --> Segment: 42|Amount of data received so far: 4200
Acknowledged! --> Segment: 43|Amount of data received so far: 4300
Acknowledged! --> Segment: 44|Amount of data received so far: 4400
Acknowledged! --> Segment: 45|Amount of data received so far: 4500
Acknowledged! --> Segment: 46|Amount of data received so far: 4600
Acknowledged! --> Segment: 47|Amount of data received so far: 4700
Acknowledged! --> Segment: 48|Amount of data received so far: 4800
Acknowledged! --> Segment: 49|Amount of data received so far: 4900
Acknowledged! --> Segment: 50|Amount of data received so far: 5000
Acknowledged! --> Segment: 51|Amount of data received so far: 5100
Acknowledged! --> Segment: 52|Amount of data received so far: 5200
Acknowledged! --> Segment: 53|Amount of data received so far: 5300
Acknowledged! --> Segment: 54|Amount of data received so far: 5400
Acknowledged! --> Segment: 55|Amount of data received so far: 5500
Acknowledged! --> Segment: 56|Amount of data received so far: 5600
Acknowledged! --> Segment: 57|Amount of data received so far: 5700
Acknowledged! --> Segment: 58|Amount of data received so far: 5800
Acknowledged! --> Segment: 59|Amount of data received so far: 5842
The file transmission is now completed!
The file transmission is now completed!
[5842, 60]
```

Figure 2: The output of the receiver application (`receiver.py`)

4 Exerciser

You can use the exerciser to help test the correctness of your implementation. The exerciser will run a set of public test cases against your code. Ensure you have the following files all within the same directory:

- **sender.py**: Your completed sender code from section 3.1.
- **receiver.py**: Your completed receiver code from section 3.2.
- **student_number.txt**: A text file containing the student number of **one person** in the group.

In terminal, browse to the directory containing the files and type `ece361-exercise 1`.

5 Submission

Once you are confident of your implementation, you can run the submission process (which will invoke the exerciser before submitting). Only one person in the group needs to submit.

From the same directory as where you ran the exerciser, type `ece361-submit submit 1`.

You can then verify the submission by typing `ece361-submit list 1`.

At some point after the lab's due date, private test cases will be run against your submission to calculate your final mark.

6 Appendix A: The `ece361` Python Library

The `ece361` library includes a networking package which implements an enhanced environment that allows the developers (students) to experiment with various forms of end-to-end communication by using network addresses. While the networking package offers the same programming features that you normally see within real production environments, it offers more flexibility to support the teaching objectives of the course. As a result, you may want to seek out more information on network programming after you have completed this lab assignment.

The networking package offered by `ece361` library provides an important class called `Socket`. You will learn how to use its methods to create network applications in both track 1 and track 2 of this lab assignments. The following sections will explain the `Socket` class and its methods in more detail.

6.1 The `Socket` Class

A network socket is one endpoint of a two-way communication link. The endpoint can be created by a software application via an application programming interface (API) that provides the required features. Therefore, to enable an application to communicate over the network, the application developer must first create a socket to build an endpoint. The endpoint can be used to send and receive data from the network. In this lab environment, you can easily build a socket by importing the `ece361` networking package and create objects of class `Socket`. The following example demonstrates how you can do this. In addition, you can find informative examples by completing the track 1 tutorial modules.

```
1 from ece361.netowrk.socket import Socket
2
3 # In your code
4 socket_object_name = Socket(<Your Student Number>)
```

Listing 2: Importing the `Socket` class

The `Socket` class used in the lab environment requires you to provide your student number. This is due to the fact that the lab environment will create an isolated personal network for you, in which makes your data messages are invisible to others. Therefore, it is of utmost importance to provide your student number correctly. Students in a team are required to use one of the team members' student number.

If you use a particular student number to create a socket object, you must use the same number in every single instance of your socket objects to allow the endpoints to be part of the same network. For instance, if you have a sender and a receiver that need to exchange data with each other, they must both instantiate their socket objects using the same student number.

After instantiating a socket object, you can use the following class methods to build your applications. Please read each method's manual to learn more about the function they provide.

6.1.1 `Socket.change_source_address(address, port)`

This method allows you to change the socket address and port number. The input parameters are as follows:

- **address**: An IP address or any name in the format of string characters.
- **port**: A number or name in the format of string characters.

```
my_socket = Socket('1234567890')
my_socket.change_source_address(address='130.14.45.88', port='1221')
...
my_socket.change_source_address(address='ece361', port='app1')
```

Listing 3: Example for using `change_source_address()`

Note: Be advised that in the enhanced environment provided by the `ece361` library, when you create a socket object, it chooses a random address and port number as the source address. The initial port number is 'N/A' which indicates the socket does not use any port number. Therefore, the following items are important to keep in mind.

- If you create a socket, the best practice is to change the address and port numbers to your preferred ones to avoid confusion.
- Both address and port must be provided as a string. Please see the examples.
- The flexibility that you have to use a socket without specifying a port number is due to this specific lab environment. In real production environments, you must specify both the address and port number at the time of socket creation.

6.1.2 `Socket.sendto(application_data, dst_address, dst_port)`

This method enables the application developers (students) to send data messages to destination endpoints. The input parameters are as follows:

- **application_data**: This parameter accepts string characters arguments.
- **dst_address**: An IP address or any name in the format of string characters.
- **dst_port**: A number or name in the format of string characters.

Note: The `sendto()` function can carry application data which is less than or equal to 100 bytes. It returns -1 if the data is larger than the maximum segment size, or it returns the length of the data that was successfully transmitted.

The `sendto()` method can be used in three ways.

1. Without specifying a destination address and port: This way the `sendto()` method broadcasts its data to every node on the network.

```
my_socket.sendto(application_data)
```

Listing 4: Not specifying any destination (i.e. broadcasting)

2. Using a destination address without specifying a port: This approach can be used to send the data to a specific node on the network. However, if the name is be shared by several applications, they all will receive the message.

```
my_socket.sendto(application_data='hi', dst_address='130.45.36.78')
```

Listing 5: Specifying just the destination address

3. Specifying both the destination address and port: This approach can be used to send the data to a specific application endpoint.

```
my_socket.sendto(application_data='hi', dst_address='130.45.36.78', dst_port='app1')
```

Listing 6: Specifying both destination address and port

Note: To allow the receiver endpoint to receive data messages, the `dst_address` and `dst_port` must match those of the receiver socket object. Therefore, you must first change the receiver's source address and port number to any values of your choice before you send the data to the receiver. The new values can be used as the destination address on the `sendto()` method. Please complete the track 1 modules to see various examples of how to use your knowledge about the `sendto()` method in building your own applications.

6.1.3 `Socket.recvfrom()`

This method can be used to obtain data messages from the network. The `recvfrom()` method does not require any input arguments. If the method is used, it only receives broadcast messages or the messages that are destined for this receiver.

Note: The receiver endpoint is identifiable on the network by either its source address, or its source address and port number. Therefore, if a message is sent to a receiver by only specifying the destination address, the receiver can obtain the message if the destination address matches the one on the receiver. The receiver will also check the port only if it is provided as a part of the destination address on the `sendto()` method. Please complete the track 1 modules to see various examples of how the `recvfrom()` method can be used to build your own applications.

Note: The `recvfrom()` method will return the source (sender) address of the message as well as the message data. The address will be returned as a list where the first element is the sender's address and the second element is the sender's port number. Please note that if you send a broadcast message or a message without a destination port, you may still receive the source address and port. These values are either the sender's initial address and port that are chosen at the time of socket creation, or are the values that have been set by the sender's application developer. In any case, you can use this information to reply to the sender. Track 1 modules will show you how you can take the advantage of this information in building your own applications.

```
a, b = my_socket.recvfrom()
print(a) # [source address, source port]
print(b) # Message Data
```

Listing 7: Example for using `recvfrom()`

6.1.4 `Socket.display_last_header()`

This method can be used to show the header of the last sent message. When you use the `sendto()` method to send a message, the `sendto()` will create two data structures.

1. **Message Header**: The header contains four pieces of data and is used to allow the network to direct messages to their destinations. The header contains `[source_address, source_port, destination_address, destination_port]`. Therefore, by using this method, you can always see the header of the last sent message.
2. **Segment**: The segment includes both the header and the application data.

Note: The information in the header depends on the values of the source address and how the developer specified the destination receiver for the `sendto()` method. You can see an example of how to use this method by completing the track 1 module 4.

- A message header:

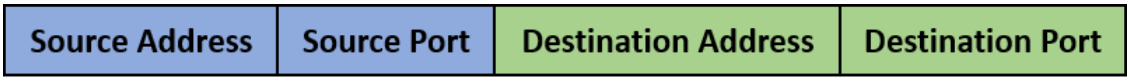

Figure 3: A message header.

- A message segment:

Figure 4: A message segment, containing both the header and message data.

```
1 my_socket.display_last_header()
```

Listing 8: Example for using `display_last_header()`

6.1.5 `Socket.display_last_segment()`

Similar to above, this method can be used to see the last segment's message data.

Note: You can see an example of how to use this method by completing track 1 module 4.

```
1 my_socket.display_last_segment()
```

Listing 9: Example for using `display_last_segment()`