Socket Programming: Section 1 Transcript

Introduction

Welcome to the Socket Programming module.

Sockets are the foundation for network communication in all modern operating systems. First introduced in the Berkeley Software Distribution (BSD) version of UNIX in 1983, the idea of sockets was subsequently adopted by most operating systems known today. Sockets make it possible for applications to listen to incoming connections, send and receive data, and terminate the connection once data exchange has been deemed completed.

Throughout this module, you'll be presented with opportunities to assess and apply what you've learned.

At the end of this module, you will be able to:

- Describe networking sockets,
- Develop basic TCP applications using socket programming in Python,
- Develop basic UDP applications using socket programming in Python,
- Generate raw network packets using socket programming in Python, and
- Demonstrate the ability to manipulate packets using Scapy.

Bypass Exam Introduction

If you are already familiar with the subject matter presented in this module, you can choose to take a Bypass Exam to skip this module.

The Bypass Exam option provides a single opportunity to successfully demonstrate your competence with the material presented within the module. If you pass, you'll receive credit for completing the module, unlocking the content within, and you will be free to proceed to the next module. If you do not pass, you will need to successfully complete the module, including all exercises and the Module Exam, to receive credit.

Click the Next Section button to continue.

Socket Programming: Section 2 Transcript

Networking Sockets

Let's begin by describing network sockets. A socket is a network communication endpoint available within an operating system. It is mostly made available to programmers and users via the use of dedicated Application Programming Interfaces, or APIs, that vary depending on the operating system or the implementation.

Sockets allow user applications to perform networked communication by providing an interface that exposes the networking stack residing within the operating system, or kernel.

The most popular use of sockets is for developing client and server applications using the TCP or UDP protocols. But other uses include the development of raw IPv4 or IPv6 applications without the need for TCP or UDP, or the development of traffic generation applications that are capable of fully crafting network packets at the bit level.

Categorizing Sockets: Socket Domain

Sockets can be categorized using three properties: the socket domain, the socket type, and the protocol type.

In most cases, the domain represents a protocol family, which is virtually what set of protocols will be used for communication. The domain usually relates to the network layer protocol used to carry the traffic; but this is not always the case.

A few examples of domains are the UNIX socket domain, the IPv4 domain, the IPv6 domain, and the low-level sockets domain. Let's discuss each one in more detail.

Categorizing Sockets: Domain

The UNIX sockets domain is used for local communications on the same machine and does not relate to any networking protocol. The communication mechanism used in this domain is somewhat similar to pipes. The main difference with pipes is that these sockets allow two applications to send and/or receive data as if they were communicating over a network. The UNIX sockets domain is only available on UNIX or Linux platforms.

The IPv4 domain is comprised of all protocols that can directly be carried by IPv4 such as ICMP, TCP, UDP, IGMP, and so on.

On the other hand, the IPv6 domain is comprised of protocols that can directly be carried by IPv6 such as ICMP6, TCP, UDP, and so on. You may have noticed that some protocols can belong to more than one domain, which is the case for TCP and UDP.

The low-level sockets domain does not relate to any specific network layer protocol, but it provides the capability to manually build packets from scratch before sending them on the network at the link

layer. The low-level sockets domain is only supported on UNIX and UNIX-like operating systems.

Each domain is identified by a symbol in socket programming. For the UNIX sockets domain, the symbol can be AF_UNIX or AF_LOCAL, which are equivalent. The IPv4 domain has a symbol of AF_INET. For the IPv6 domain, the symbol is AF_INET6. Finally, for low-level sockets, the symbol is AF_PACKET.

Categorizing Sockets: Socket Types

Sockets can also be categorized using the concept of socket type. The socket type usually represents what communication pattern is being used within a domain. It is important to note that not all socket types are available for (or can be used with) all domains. As with domains, the socket types are identified by a symbol in socket programming as well.

For instance, SOCK_STREAM represents a sequenced, reliable, two-way, connection-based communication pattern. In the AF_INET and AF_INET6 domains, this usually corresponds to the TCP protocol.

SOCK_DGRAM represents a connectionless, unreliable, and datagram-based communication pattern. In the AF_INET and AF_INET6 domains, this usually corresponds to UDP.

SOCK_RAW, also known as raw sockets, provides raw network protocol access. In this communication pattern, sections of each packet are built from scratch.

Categorizing Sockets: Protocol Type

The protocol type is used to fully identify what IP protocol is used for communication. The protocol type is identified by a number that is defined within RFC 1700. That number also matches the protocol field in the IPv4 header or the "Next Header" field within the IPv6 header.

In some communication scenarios, the protocol type is ignored because the socket domain and the socket type are enough to fully define what protocol is used. This is the case for sockets of the AF_INET or AF_INET6 domains, and the SOCK_STREAM or SOCK_DGRAM socket type. In that situation, TCP or UDP are clearly the protocols that are used respectively.

Protocol types can also be identified by a symbol. For instance, IPPROTO_TCP represents the TCP protocol; IPPROTO_UDP represents the UDP protocol; and IPPROTO_ICMP represents the ICMP protocol.

Some other generic protocol types can be used as well. For instance, IPPROTO_RAW does not represent any specific protocol, but it is used to indicate that the protocol is already specified in the IP header. The protocol type is sometimes called protocol number or protocol id.

Section Completed

Socket Programming: Section 3 Transcript

Performing TCP Communications with Sockets

In this topic, we will discuss how to perform TCP communications with sockets. Using TCP as a communication protocol follows a very specific pattern. Since TCP is a connection-based protocol, there is an implicit expectation that one endpoint will be listening for incoming connections and that the other will initiate or request the connection. If the connection is authorized, a communication channel is established.

The communication endpoint that listens for incoming connections is the server; whereas, the communication endpoint that initiates the connection is the client. The client and server are the foundation of the renowned client/server model for networked applications.

TCP Socket Communication Steps: Server Side

To gain a better understanding, we will break down the communication steps that must be performed in order to communicate using a TCP socket. On the server side, the following socket function calls are made:

- First, the server creates a socket with the `socket()` system call in order to perform communication over the network. The server must specify the socket domain, the socket type, and the protocol type in the arguments to the system call. In many cases, the protocol type argument can be omitted or set to a default value of zero.
- Then the server uses the `bind()` system call to assign the interface IP address and a port to use for communication. Note that the address and port pair cannot be used twice within the system. Typical interfaces specifications are 127.0.0.1 (localhost), a specific IP address on a multi-homed system, or 0.0.0.0 (any interface).
- The server continues with the `listen()` system call to start listening for incoming connections and placing them in a queue.
- The server can now use the `accept()` system call to pull connections one at a time from that queue. The call to `accept()` will block, that is, perform no other actions, until a client requests a connection to the server.
- For each connection, the server can exchange data by using the `send()` system call to send data or the `recv()` system call to receive data. The call to `recv()` will block until data is available after the client sends it.
- Finally, the server can terminate a connection by closing the socket with the `close()` system call.

TCP Socket Communication Steps: Client Side

On the client side, the following socket function calls are made:

- First, the client creates a socket with the `socket()` system call in order to perform

communication over the network. The client must specify the socket domain, the socket type, and the protocol type in the arguments to the system call. In most cases, the protocol type can be omitted or set to a default value of zero.
- Then the client uses the `connect()` system call to initiate a connection to the server. The client must specify the IP address and port used by the server. The network stack automatically allocates an ephemeral port to the client locally. The connection is established after the three-way handshake is performed.
- The client can now exchange data by using the `send()` system call to send data or the `recv()` system call to receive data. The call to `recv()` will be blocked until data is available after the server sends it.
- Finally, the client can terminate a connection by closing the socket with the `close()` system call.

TCP Socket in Python: Server Side

In this topic, we learn how to interpret basic TCP applications that use socket programming in Python. We will begin by examining the server side. Let's take a look at the Python script of a TCP server endpoint.

The first line is a Python directive that tells the interpreter to make all the socket library symbols and functions available to the script.

On the second line, the server is creating a socket of the AF_INET domain. The type is SOCK_STREAM, which is a TCP socket. Notice that the protocol type does not need to be specified in this case.

On the third line, the server binds to IP address 192.168.11.11 and port 12345. This assumes that the node from which this script will run actually has 192.168.11.11 assigned as an IP address and that the port 12345 is not being used by another TCP server application; otherwise, the call to bind will fail. Notice that `bind()` has a single argument which is a pair composed of an IP address and a port number.

On the fourth line, the server starts listening for an incoming connection and places them on a processing queue of 10 connections. As connections are being processed with `accept()`, the queue size will decrease. However, if the queue size has been reached at a given point, additional connections will be dropped.

On the fifth line, the server processes a connection with the call to `accept()`. This server in particular calls `accept()` only once; therefore, one and only one connection will be processed. The `accept()` call will return a pair including the client socket to use to send or receive data to or from that connected client and the address of that client. That client address is also another pair containing the IP address and port number of the client.

On the sixth line, the server waits for data from the client to become available. The server will pull a maximum of 100 bytes from the network buffer based on the argument of the `recv()` call.

The server prints the result of the seventh line. Printing the data only makes sense if text would be received.

Then the server sends a string to the client on line 8 and closes the connection on line 9.

TCP Socket in Python: Client Side

Now we will consider how things should be done from the client side in Python.

The first line is a Python directive that tells the interpreter to make all the socket library symbols and functions available to the script.

On the second line, the client is creating a socket of the AF_INET domain. The type is SOCK_STREAM, which is a TCP socket. Notice that the protocol type does not need to be specified in this case.

On the third line, the client requests a connection to IP address 192.168.11.11 and port 12345. For the `connect()` call to succeed, IP address 192.168.11.11 must be routable. Furthermore, a server must accept connections to that address on port 12345. Notice that `connect()` has a single argument, which is a pair composed of an IP address and a port number.

On the fourth line, the client sends data to the server.

On the fifth line, the client attempts to pull 100 bytes of data sent to it by the server. The call will block until the server sends some data to the client.

Finally, the client prints the data to the screen and then closes the connection.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Socket Programming: Section 4 Transcript

Performing UDP Communications with Sockets

In this topic, we will discuss how to perform UDP communications with sockets. When comparing the two, UDP communication with sockets follows a different pattern than TCP communication with sockets.

If TCP sockets require a connection to be established before data can be exchanged between two endpoints, UDP sockets do not. The only requirement to receive data is for an application to bind to an address and a port.

The communication endpoint that binds to receive incoming packets is the server. An endpoint that sends UDP packets to the server is a client.

If a client needs to receive data from the server, it also has to bind to a local address and port. In that case, the distinction between the client and the server becomes a little blurred because each endpoint acts as a client or server depending if it's sending or receiving data, respectively.

UDP Socket Communication Steps: Server Side

Let's break down the communication steps that must be performed in order to communicate using a UDP socket. On the server side, the following socket function calls are made:

First, the server creates a socket with the `socket()` system call in order to perform communication over the network. The server must specify the domain, the socket type, and the protocol type in the arguments to the system call. In many cases, the protocol type argument can be omitted or set to a default value of zero.

Then the server uses the `bind()` system call to specify which IP address and port to use for communication. An address and port pair cannot be used twice within the same system.

The server can now receive data by using the `recvfrom()` system call. The call to `recvfrom()` will block until data is available after the client sends it.

Finally, the server can terminate the connection by closing the socket with the `close()` system call.

UDP Socket Communication Steps: Client Side

On the client side, the following socket function calls are made:

First, the client creates a socket with the `socket()` system call in order to perform communication over the network. The client must specify the domain, the socket type, and the protocol type in the arguments to the system call. In many cases, the protocol type argument can be omitted or set to a default value of zero.

The client can now send data by using the `sendto()` system call. The client must specify the address and port of the server. Because of the nature of UDP as an unreliable protocol, data is going to be sent over the network without any verification of the existence of the server endpoint.

Finally, the client can terminate the connection by closing the socket with the `close()` system call.

UDP Socket in Python: Server Side

In this topic, we will learn how to develop basic UDP applications using socket programming in Python. Let's consider the following Python script of a UDP server endpoint from the server side.

As in the other cases, we always start by telling the interpreter to make all the socket library symbols and functions available to the script.

On the second line, the server is creating a socket of the AF_INET domain and of type SOCK_DGRAM, which is exactly a UDP socket. Once again, the protocol type does not need to be specified in this case.

On the third line, the server binds to IP address 192.168.11.11 and port 12345. This assumes that the node from which this script will actually run has 192.168.11.11 assigned as an IP address and that port 12345 is not being used by another UDP server application; otherwise, the call to bind will fail.

On the fourth line, the server awaits for data sent from any client. The argument of `recvfrom()` specifies the maximum size of data that can be received.

The call to `recvfrom()` will block until data becomes available. Then the call returns a pair composed of the data sent by the client and source address and port pair.

The server prints the data received on the fifth line. Once again, printing the data only makes sense if text is received.

Finally, on the last line, the server closes the socket. Any additional data sent to this server from this point will be discarded by the network stack.

UDP Socket in Python: Client Side

Now let's consider how things should be done from the client side in Python.

As usual, first, tell the interpreter to make all the socket library symbols and functions available to the script.

On the second line, the client is creating a socket of the AF_INET domain. The type is SOCK_DGRAM, which is a UDP socket. Once more, the protocol type does not need to be specified.

On the third line, you can start sending data to any UDP server out there as long as you know the IP address and port number it is bound to. In our case we are using IP address 192.168.11.11 and port 12345. Notice that it is not necessary to setup an initial connection to send the data. This behavior is

consistent with the characteristics of the UDP protocol.

When you are done sending data, close the socket using the `close()` command.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Socket Programming: Section 5 Transcript

Performing Raw Communications With Sockets

Raw sockets are the most powerful tool in a socket programmer's arsenal. As a starting point, raw sockets allow you to send data using existing protocols. In addition, you can also experiment with raw sockets and create your own data transfer protocols. This capability is possible because you can use raw sockets to build, from scratch, sections of a protocol data unit, or even a full protocol data unit, if you desire.

For instance, if you use the SOCK_RAW socket type with the AF_INET domain, you can easily develop a new transport layer protocol that will continue to rely on IPv4, and lower level protocols to be carried over the network. That approach, however, has its limitations. In this example you would be forced to use the IP address and MAC address that are assigned to the machine network interface as the source of the packets sent over the network.

To alleviate that limitation, you can use the AF_PACKET domain to build the whole packet from scratch. It is important to note that you will need to calculate and fill some of the fields that would normally be handled by the network stack. Specifically, you will need to calculate and fill the checksum fields, length fields, padding, and flags.

Because of the complexity of this topic, in this section we will focus our discussion on the raw sockets specifically related to the AF_INET and AF_INET6 domains.

Performing Raw Communications With Sockets: Server Side

We will begin by examining the server side. Let's take a look at a script that makes use of a raw socket; and in this example we will skip directly to the most significant lines in the code.

On the second line, we define a new variable called IPPROTO_CUSTOM, and we set the value to 102.

On the third line we create a socket of the AF_INET domain, the SOCK_RAW type, and the protocol type is set to IPPROTO_CUSTOM or 102. According to RFC 1700, which defines protocol types and their corresponding number, the numbers 101 through 254 are currently not assigned to any transport protocol. So it is appropriate to choose 102 for our custom transport protocol.

On the fourth line, we bind our socket to IP address 192.168.11.13. This assumes of course that the machine that will run this code is assigned that IP.

As for the port number, it only has significance for protocols such as UDP and TCP. For our case, it will have no effect. So, in this example we can set the port number to zero.

Performing Raw Communications With Sockets: Client Side

Now we will consider how things should be done from the client perspective. Let's see how to send

data to the server that supports our custom protocol. As we did for the server, we will skip directly to the most significant line in the code.

On line 4, the client performs a `sendto()` call to send data to the server. The arguments used are very similar to those in the case of standard sockets. The only difference is that the port number can be set to zero since this is not significant in our protocol and would have no effect overall.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Socket Programming: Section 6 Transcript

Performing Advanced Socket Communications with Scapy

In this section we will introduce Scapy, an advanced packet manipulation tool for computer networks. This tool allows you to painlessly perform all operations that can be accomplished using sockets.

When properly installed on a system, Scapy can be accessed via the Python interactive shell or via scripting. To access Scapy via the Python interactive shell, the easiest way is to run the `scapy` command. To use Scapy in a script in a similar manner as it would be used in the Python interactive shell, you can use the following instruction: `from scapy.all import *` at the beginning of your script.

First we will introduce a few commands, and later in this topic you will have the opportunity to practice using Scapy to perform socket communications in an Exercise.

Scapy Commands

Let's begin with a few Scapy commands that you can use on the Python interactive shell.

The `ls()` command is used to view all available protocols within Scapy.

`lsc()` is used to list all available functions in Scapy.

Because IP is an available protocol in Scapy, creating an IP packet is easy using the `IP()` packet constructor.

Once the packet is constructed, you can view the fields of the packet using the `show()` function. As you run that command, you will notice the packet is displayed with default values for all fields.

You can change some of the fields by directly assigning them a value. For instance, let's say a packet is assigned a destination IP address of 192.168.11.13. If the destination address is a non-local address, Scapy will automatically assign one of the machine's non-local IP addresses.

Fields can also be set directly during the creation of the packet by passing them as arguments in the construction of the packet.

A payload can be inserted by using the "`/`" operator. When this is done, the resulting packet contains the payload while the original packet is left unchanged. In this example, we created a new packet by inserting a TCP payload using the TCP packet constructor.

Next, raw data is yet again inserted, resulting in a full packet.

Once a payload is inserted, since each packet is actually a hash-table or dictionary of all its subsections, you can access each payload by indexing the dictionary with the name of the payload's

constructor.

After the packet is constructed, you can display a hexdump of the packet by applying the `str()` function to it.

Then you can send the packet using the `send()` function call. The `send()` command can also replay the packet multiple times, but the transmission always happens at layer 3. In other words, the network stack automatically builds the frame by using any means necessary. If you would prefer to send at layer 2, you would use the `sendp()` command.

If packets are expected to be returned from the third party, then you should enable a send-receive loop using the `srloop()` function. Alternatively, you can also capture packets on the wire with calls to the `sniff()` function.

To view the result, you can create a tcpdump-like output by using the `show()` function call. Or, you can access each packet independently since the capture result is an array. Finally, for each captured packet, subsections or fields can also be accessed as it would for a normal packet.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Socket Programming: Section 7 Transcript

Summary

You have completed the Socket Programming module.

In this module we learned about networking sockets. We discussed the categories of sockets. We discussed the steps of basic TCP and UDP communications from both the client and server sides. We performed raw communications using sockets which allow you to send data using existing protocols and even create your own data transfer protocols. We also discussed some common functions in Scapy used to perform advanced packet manipulation.

You should now be able to:

- Describe networking sockets,
- Develop basic TCP applications using socket programming in Python,
- Develop basic UDP applications using socket programming in Python,
- Generate raw network packets using socket programming in Python, and
- Demonstrate the ability to manipulate packets using Scapy.

To receive credit and advance to the next module, you must achieve a passing score on the Module Exam. Click the Next Section button to begin the Module Exam.