Sockets

Helpful References

- Rhodes and Goerzen, Foundations of Python Network Programming, Apress, 2014

Sockets

- Sockets are data structures that abstract communication endpoints
- All POSIX ("Portable Operating System Interface")-compliant systems support sockets
- To receive messages sent to a specific port (1 - 65535), a process must first "bind" to a socket
- Recall that the first 1024 ports require root privileges in order to bind to them

Sample UDP Server

```python
#!/usr/bin/python3.11

from socket import socket, AF_INET, SOCK_DGRAM

BUF_SIZE = 1024
HOST = '127.0.0.1'  # '' would cause the server to listen on all interfaces
PORT = 65432

with socket(AF_INET, SOCK_DGRAM) as sock:  # UDP Socket
    sock.bind((HOST, PORT))  # Claim messages sent to port "PORT"
    data, address = sock.recvfrom(BUF_SIZE)  # Receive up to "BUF_SIZE" bytes
    text = data.decode('utf-8')  # Convert from binary representation
    source_ip, source_port = address  # Extract source IP and source port
    print(f'IP: {source_ip} Port: {source_port} Message: {text}')
```

Sample UDP Client

```
#!/usr/bin/python3.11

from socket import socket, AF_INET, SOCK_DGRAM
from sys import argv

BUF_SIZE = 1024
HOST = '127.0.0.1'
PORT = 65432

if len(argv) != 2:
    print(argv[0] + ' <message>')
    exit()

with socket(AF_INET, SOCK_DGRAM) as sock:  # UDP socket
    data = argv[1].encode('utf-8')  # Convert command line arg to binary
    sock.sendto(data, (HOST, PORT))  # Send message to server
```

Test Run

- Enter
  ./client.py "This is a test"
- Server prints
  IP: 127.0.0.1 Port: 57937 Message: This is a test
- The source port number will vary

Review: Setting Up a UDP Server

- Create a socket data structure:
  - ***with socket(AF_INET, SOCK_DGRAM) as sock:***
- Claim (bind to) a port
  - ***sock.bind((HOST, PORT))***
  - This lets the operating system know that the calling program wants to receive (UDP) messages to the given host and port
  - To receive messages from the network, *HOST* must either be blank or contain the IP address of the network card (assuming there is only one)
  - To receive messages from the local host only, use *localhost* or *127.0.0.1*

Review: Setting Up a UDP Client

- Create a socket data structure:
  - ***with socket(AF_INET, SOCK_DGRAM) as sock:***

Review: Receiving via UDP

- To receive a network message, *recvfrom(BUF_SIZE)* is used:
    - ***data*, *address* = *sock.recvfrom(BUF_SIZE)***
- Note that the *BUF_SIZE* is the maximum acceptable size; the actual size could be less (e.g., sending a 1024-byte message could be received as 2 separate 512-byte messages)
- If exactly *n* bytes are needed, *recvfrom* will have to be called repeatedly until *n* bytes are received
- the address contains the sender's IP and port in the form of a tuple:
    - ***data*, *(source_ip*, *source_port)* = *sock.recvfrom(BUF_SIZE)***

Review: Receiving via UDP

- The received message is in binary format (i.e., it is encoded). Before processing, the message should be decoded first using the 'decode' function.
    - ***text = data.decode('utf-8')***

Review: Sending via UDP

- To send a message, use *sendto(data, (host IP, host port))*
- The transmitted data must be encoded prior to calling this function.
    - ***sock.sendto(text.encode(), (source_ip, source_port))***
- There is a chance that *sendto* won't send all bytes
- *sendto* does return the actual number of bytes sent, so if that does not equal the buffer size, the remaining message will have to be retransmitted.
- Note that UDP messages can be lost or duplicated. It is up to the application to address these issues

Sample TCP Server

```
#!/usr/bin/python3.11

from socket import socket, AF_INET, SOCK_STREAM, SOL_SOCKET, SO_REUSEADDR

BUF_SIZE = 1024
HOST = ''
PORT = 65432

with socket(AF_INET, SOCK_STREAM) as sock:  # TCP socket
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)  # Details later
    sock.bind((HOST, PORT))  # Claim messages sent to port "PORT"
    sock.listen(1)  # Server only supports a single 3-way handshake at a time
    print('Server:', sock.getsockname())  # Server IP and port
    while True:
        sc, _ = sock.accept()  # Wait until a connection is established
        with sc:
            print('Client:', sc.getpeername())  # Client IP and port
            data = sc.recv(BUF_SIZE)  # recvfrom not needed since address known
            print(data)
            sc.sendall(data)  # Client IP and port implicit due to accept call
```

Sample TCP Client

```
#!/usr/bin/python3.11

from socket import socket, AF_INET, SOCK_STREAM
from sys import argv

BUF_SIZE = 1024
HOST = '127.0.0.1'
PORT = 65432

if len(argv) != 2:
    print(argv[0] + ' <message>')
    exit()

with socket(AF_INET, SOCK_STREAM) as sock:  # TCP socket
    sock.connect((HOST, PORT))  # Initiates 3-way handshake
    print('Client:', sock.getsockname())  # Client IP and port
    data = argv[1].encode('utf-8')  # Convert command line arg to binary
    sock.sendall(data)  # Server IP and port implicit due to connect call
    reply = sock.recv(BUF_SIZE)  # recvfrom not needed since address known
    print('Reply:', reply)
```

Timeout Notes

- If you ever get a *Address already in use* error, the socket is still in use, possibly by an old version of your server
- Normally, terminating the server does **not** immediately close the socket, since TCP requires the use of timeouts in its protocol (due to the Two Army Problem)
- This means that terminating the server and then restarting it can cause *bind* to fail because the system is keeping the socket until the timeout has occurred
- *SO_REUSEADDR* is used to make debugging a little easier by overriding this timeout feature

Send and Receive Notes

- While *sendall* will send all data, it may decided to split the message into several parts
- *recv(n)* will read ***up to*** *n* bytes, but it could be less! It is up to the application to check that the correct number of bytes were received

Review: Setting up a TCP Server

- Create a server socket data structure:
    - ***with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:***
- Claim (bind to) a port:
    - ***sock.bind((HOST, PORT))***
- Set up a queue:
    - ***sock.listen(n)***
    - This lets the operating system know that the calling program wants the system to allow up to *n* clients to connect at the same time
    - Any additional clients initiating a 3-way handshake will be ignored, until the calling program has called the *accept* function

Review: Setting up a TCP Server

- Accept a connection:
  - ***sc, _ = sock.accept()***
  - This returns a client socket and information about the connected client
  - Once a client socket is available to the program, it can be used to receive and send messages from and to the client

Review: Setting up a TCP Client

- Set up a socket data structure:
    - ***with socket(AF_INET, SOCK_STREAM) as sock:***
- Connect to the server:
    - ***sock.connect((HOST, PORT))***

Review: Receiving via TCP

- To receive a network message, *recv(BUF_SIZE)* is used:
    - ***data = sc.recv(BUF_SIZE)***
- Note that *BUF_SIZE* is the maximum acceptable size; the actual size could be less
- *recv* only returns the data, since the originating client is already known
- Data must be decoded:
    - ***text = data.decode('utf-8')***

Review: Sending via TCP

- To send a message, use *sendall(data)*
- The transmitted data must be encoded prior to calling this function:
  - ***sc.sendall(text.encode('utf-8'))***
  - Transmissions are buffered, and so sending *n* bytes with *sendall* does not mean that *recv(n)* will receive all those bytes; we may have to call *recv(n)* multiple times until all *n* bytes are collected
  - e.g., if you know *sendall* was called with 1024 bytes, you need to call *data = data + recv(1024 - len(data))* repeatedly at the receiving end until all 1024 bytes have been collected

Read Until Newline

- If we know that each message ends with a newline, we can call *data* = *recv(1)* until a newline is received...for now; in a production environment, we would have to optimize this approach

Read Until Newline

```
BUF_SIZE = 1  # Not efficient


def get_line(current_socket: socket) -> str:
    buffer = b''
    while True:
        data = current_socket.recv(BUF_SIZE)
        if data == b'' or data == b'\n':
            return buffer
        buffer = buffer + data
```

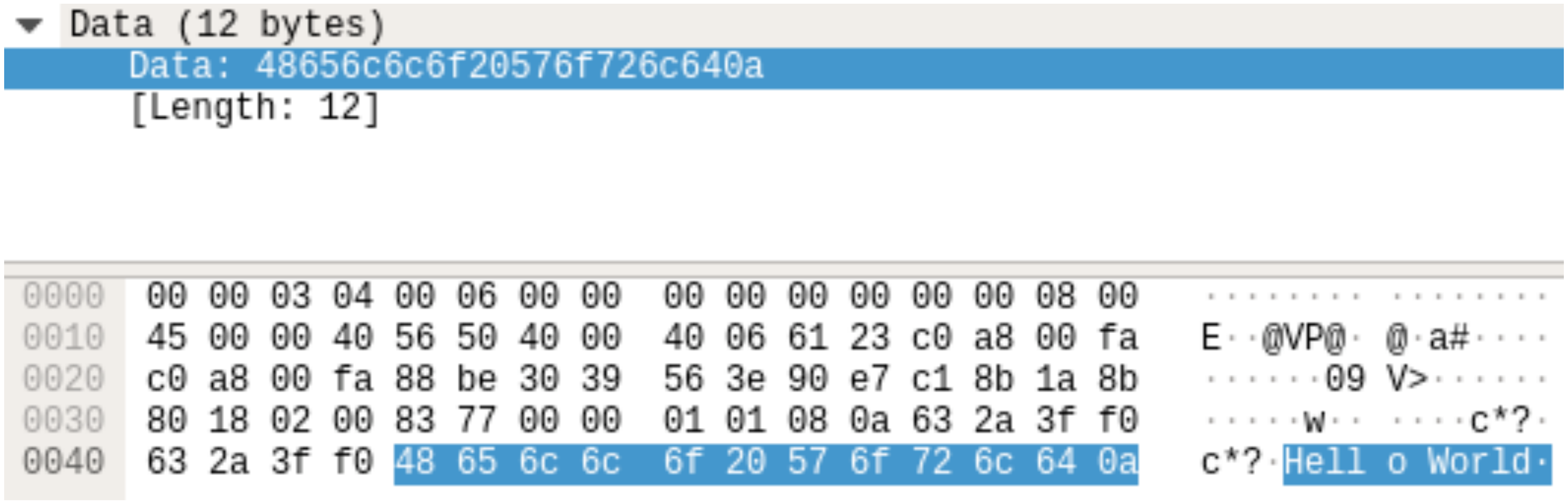
Data Encoding

Transmitting Strings

- When transmitting data over a network, we transmit strings in binary form (e.g., *'Hello World'.encode()* or *b'Hello World'*, as opposed to the normal *'Hello World')*
- b*'ABC'* and *'ABC'* are different: Using the *type(b'ABC')* command, we get <*class 'bytes'*> and using the *type('ABC')* command, we get <*class 'str'*>
- If *x* is a binary string representing *Hello World*, we can view its binary form using *list(x)*
  [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]
- If *x* is a normal string representing *Hello World*, *list(x)* gives us
  ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']

Transmitting Strings

- We can run Wireshark to confirm this (numbers printed in hexadecimal)

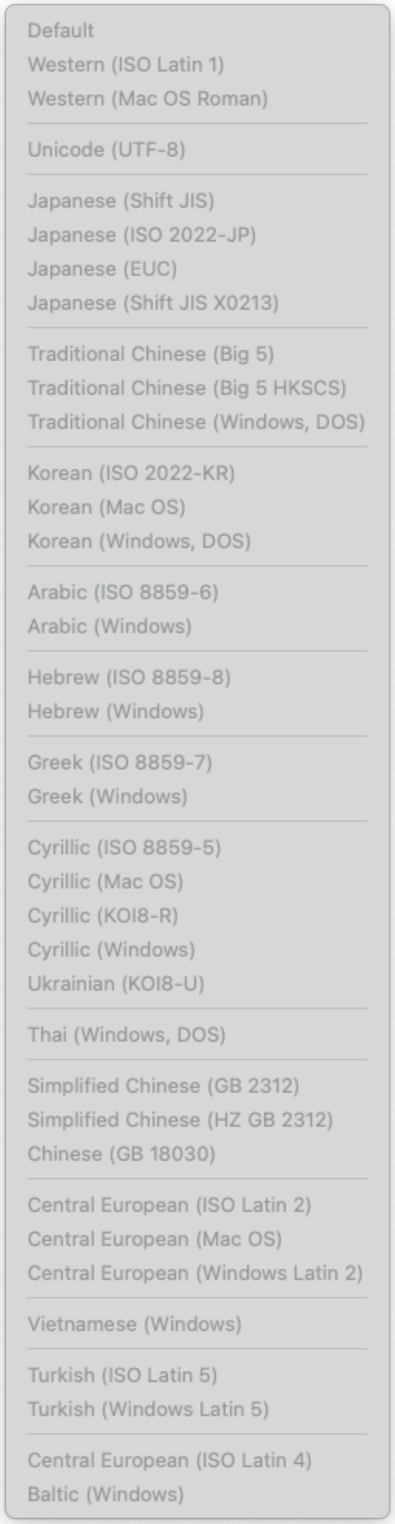


```
▼ Data (12 bytes)
    Data: 48656c6c6f20576f726c640a
    [Length: 12]

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 40 56 50 40 00  40 06 61 23 c0 a8 00 fa   E··@VP@· @·a#····
0020  c0 a8 00 fa 88 be 30 39  56 3e 90 e7 c1 8b 1a 8b   ······09 V>······
0030  80 18 02 00 83 77 00 00  01 01 08 0a 63 2a 3f f0   ·····w·· ····c*?·
0040  63 2a 3f f0 48 65 6c 6c  6f 20 57 6f 72 6c 64 0a   c*?·Hell o World·
```

- The key point is that the sequence 72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100 is sent over the network. It is up to our applications to interpret this data appropriately

Transmitting Strings


Default
Western (ISO Latin 1)
Western (Mac OS Roman)
Unicode (UTF-8)
Japanese (Shift JIS)
Japanese (ISO 2022-JP)
Japanese (EUC)
Japanese (Shift JIS X0213)
Traditional Chinese (Big 5)
Traditional Chinese (Big 5 HKSCS)
Traditional Chinese (Windows, DOS)
Korean (ISO 2022-KR)
Korean (Mac OS)
Korean (Windows, DOS)
Arabic (ISO 8859-6)
Arabic (Windows)
Hebrew (ISO 8859-8)
Hebrew (Windows)
Greek (ISO 8859-7)
Greek (Windows)
Cyrillic (ISO 8859-5)
Cyrillic (Mac OS)
Cyrillic (KOI8-R)
Cyrillic (Windows)
Ukrainian (KOI8-U)
Thai (Windows, DOS)
Simplified Chinese (GB 2312)
Simplified Chinese (HZ GB 2312)
Chinese (GB 18030)
Central European (ISO Latin 2)
Central European (Mac OS)
Central European (Windows Latin 2)
Vietnamese (Windows)
Turkish (ISO Latin 5)
Turkish (Windows Latin 5)
Central European (ISO Latin 4)
Baltic (Windows)

- When receiving binary data, we can use *decode* to get back the original, e.g., *normal = binary.decode()*
- Encoding and decoding strings defaults to UTF-8, but other encodings are possible as well
- If we have a mismatch in encoders/decoders (e.g., encoded in UTF-8 but decoded as a list of 8-bit integers), we garble the data

Transmitting Numbers

- We could transmit number as strings, but this wastes resources, e.g.,
  - an unsigned 8-bit integer takes up 1 byte, and ranges from 0 to 255; the corresponding string '255' takes up 3 bytes
  - an unsigned 64-bit integer takes up 8 bytes, and ranges from 0 to 18,446,744,073,709,551,615, the corresponding string '18446744073709551616' takes up 20 bytes!
- If we write networking code running at Internet scales, we need to transmit integers more intelligently than converting them to strings

Packing Data

- Python provides a module called *struct* that supports the transmission of integers, e.g.,
  - *struct.pack('!I', 1024)* returns *'\x00\x00\x04\x00'*, i.e., hexadecimal
    00 00 04 00,
    which is the equivalent of binary
    0000 0000 0000 0000 0000 0100 0000 0000
  - Confirm that this is decimal 2^{10} or 1024

Endianness Review

- Given a number taking up multiple bytes, how should we store that number, e.g., hexadecimal ABCDEF?
- Big-Endian: Bytes are stored from left to right (historically, SPARC); begins with the most significant component
- Big-Endian: 00 AB CD EF
- Little-Endian: Bytes are stored right to left (historically, Intel); number thus begins with the least significant component
- Little-Endian (8-bit addressable): EF CD AB 00
  Little-Endian (16-bit addressable): CDEF 00AB

Format Flags

- In the pack format string, the first character is a !, which indicates network byte order
- There are other characters that can be used, e.g., < to indicate little-endian and > to indicate big-endian, e.g.,
  - *struct.pack('<I', 1024)* yields '*\x00\x04\x00\x00*'
  - *struct.pack('>I', 1024)* yields '*\x00\x00\x04\x00*'
- Failing to provide this character leads to the machine default, which **<u>may not match</u>** the default on the other machine (e.g., Raspberry Pi OS on ARM vs. Windows on Intel)
- Rule of Thumb: Use !

Format Flags

- The second character describes the data type to use
- Here is an incomplete list:

Flag	C Type	Python Type	Length
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
f	float	float	4
d	double	float	8

Bit Masks

- In addition to encoding strings and numbers, we may also have to encode other types of data, e.g, Up, Left, Right, Down
- Say we want to encode a direction command and the player ID (which can range from, say, 1 to 10)
- Given that there are 4 direction commands, we need 2 bits to represent them (e.g., 00_2 = Up, 01_2 = Left, 10_2 = Right, 11_2 = Down)
- Given 10 players, we need 4 bits for that (e.g., 0000_2 = Player 1, 0001_2 = Player 2, ... 1001_2 = Player 10)
- So we can represent the direction and player number in 6 bits!
- The general rule is that for n possibilities, $\lceil \log_2 n \rceil$ bits are needed

Bit Masks

- Question: How can we read and write bits in Python?

Boolean Logic Operator Review

-
  ```
    1010
  & 1001
  ------
    1000
  ```
- *10 & 9* (AND) yields 8

-
  ```
    1010
  | 1001
  ------
    1011
  ```
- *10* | *9* (OR) yields 11

-
  ```
    1010
  ^ 1001
  ------
    0011
  ```
- *10* ^ *9* (XOR) yields 3

Two's Complement

-
  ```
  ~ 1001
  ------
    0110
  ```

- Oddly, typing *~9* yields -10; this is because computers use two's complement when storing numbers
  - Assuming 8 bits, ~1001_2 really is ~00001001_2, which in turn is 11110110_2
  - To figure out the decimal equivalent of a binary number with a leftmost 1 (e.g., 11110110_2), when stored on a computer, we must
    - flip all the bits (e.g., 00001001_2)
    - add 1 (e.g., 00001010_2)
    - convert to decimal (e.g., 10)
    - multiply by -1 (e.g., -10)
  - If the number starts with a 0, we the apply normal binary to decimal number conversion algorithm

Two's Complement

- To compute the computer representation of a negative decimal number (e.g., -10):
  - The the absolute value of the number (e.g., 10)
  - Convert it to binary (e.g., 00001010_2 assuming 8 bits)
  - Flip all the bits (e.g., 11110101_2)
  - Add 1 (e.g., 11110110_2)
- If the number is non-negative, we convert it using the normal decimal to binary number conversion algorithm

Exercises

- Assume 8 bit numbers for the following:
  - How are the following numbers represented on a computer? -1, -0, -127, 1
  - Compute the decimal equivalent of the following 8-bit 2's complement numbers: 11110000, 00001111
  - Compute ~7
  - Compute ~-7

How to Test and Set Bits

- To test whether the nth bit of *x* has been set (i.e., is 1):
  - ***x & 2 ** n > 0***
  - e.g., to test if bit 3 (counting from the right, starting at 0) of 21 has been set, check if 21 & 2 ** 3 > 0 is true (it is not, since 21 = 1$\underline{0}$101; but bit 2 is set because 21 & 2 ** 2 > 0)
- To set the nth bit of *x*:
  - ***x = x | 2 ** n***
- e.g., to set bit 3 (counting from the right, starting with 0) of 21, run 21| 2 ** 3 (yields 29 since 1$\underline{0}$101 becomes 1$\underline{1}$101)

Exercises

- Given h = 01010011
    - How do we test if bit 2 is set? h & _____
    - How do we test if bit 7 is set? h & _____
    - What will h & 0b10100 return? _____

Exercises

- Say h = 01010011
  - How do I set bit 2? h | ____
  - How do I set bits 2 and 3? h | ____

Exercises

- Say h=01010011
  - How do I clear bit 1? h & _____
  - How do I clear bits 7 and 6? h & _____

Bit Shifting

- `1011 >> 1 is 0101`
  `0101 >> 1 is 0010`
- *11* >> *1* yields 5
- *5* >> *1* yields 2

Decimal to Binary, Using * 2

```
#!/usr/bin/python3.11

num = int(input('Enter a number:\n'))  # Assume valid number
cnt = 1
bit_string = ''
while cnt <= num:
    if num & cnt > 0:
        bit_string = '1' + bit_string
    else:
        bit_string = '0' + bit_string
    cnt = cnt * 2
print(bit_string)
```

- With input *12*, prints out 1100

Decimal to Binary, Using >>

```
#!/usr/bin/python3.11

num = int(input('Enter a number:\n'))  # Assume valid number
bit_string = ''
while num > 0:
    if num & 1 > 0:
        bit_string = '1' + bit_string
    else:
        bit_string = '0' + bit_string
    num = num >> 1
print(bit_string)
```

- With input *12*, prints out 1100

Exercises

- Write a program that prompts for a number and then prints out the number of bits in that number. For example, 10 has 2 bits in it. Assume the number is valid. Use bit shifting and testing; do not use logarithms.

Exercises

- On your own:
  - Work on the questions in the *Sockets* section of *Practice Questions and Solutions*

zlib

- The zlib module is used in Python to compress and decompress data, e.g.,
  - *original = b'This is a test'*
  - *compressed = zlib.compress(original)*
  - *decompressed = zlib.decompress(compressed)*
- Note that just like sockets, compression and decompression works with bytes
- When compressing larger amounts of text, zlib compression is very effective; for formats like JPEG, which are already compressed, zlib is not recommended

Lab 4

- Modify the game so that input and output is transmitted over the network
- Use various techniques to encode/decode data, thereby reducing network bandwidth

Logging

- *print* statements are not usually captured by the log system
- Makes it difficult to determine what happened, especially when doing forensic analysis
- Adds timing delays
- Better to use logs: They are saved to file and can be filtered easily (e.g., only capture errors, not debug messages)
- Remember that *sudo journalctl -f* allows you to monitor logs in real time

Logging

```
from logging import getLogger, DEBUG, debug, error, warning
from logging.handlers import SysLogHandler

logger = getLogger('226_game')

logger.setLevel(DEBUG)  # Debug messages are normally filtered

handler = SysLogHandler(address = '/dev/log')  # Connect to log sys
logger.addHandler(handler)

logger.debug('This is a debug message')
logger.warning('This is a warning message')
logger.error('This is an error message')
```

Key Skills

- Use sockets in Python to send and receive messages using UDP and TCP
- Encode and decode strings, numbers, and other data using a minimum number of bits
- Test, set, clear, and shift bits
- Use zlib to compress and decompress data
- Use the Python logging library