LECTURE 14: UDP AND PROGRAMMING TIPS

Computer Systems and Networks

Dr. Pallipuram
(vpallipuramkrishnamani@pacific.edu)

UDP versus TCP

	TCP	UDP
Reliable?	**Yes** *(Via acknowledgments and retransmitting)*	**No**
Connection-oriented?	**Yes** *(Server has one socket per client)*	**No** *(Server has one socket and all messages from all clients are received on it)*
Programming model?	**Stream** *(continuous flow of data – may get a little bit at a time)*	**Datagram** *(data is sent in its entirety or not at all. Size of each datagram is small)*
Applications	**HTTP (Lab 8)** *Web, email, file transfer*	**DNS (Lab 9)** *Streaming Audio/Video, Gaming*

TCP/IP sends the message as a stream

Create message

Client

Socket

`socket()`

Network

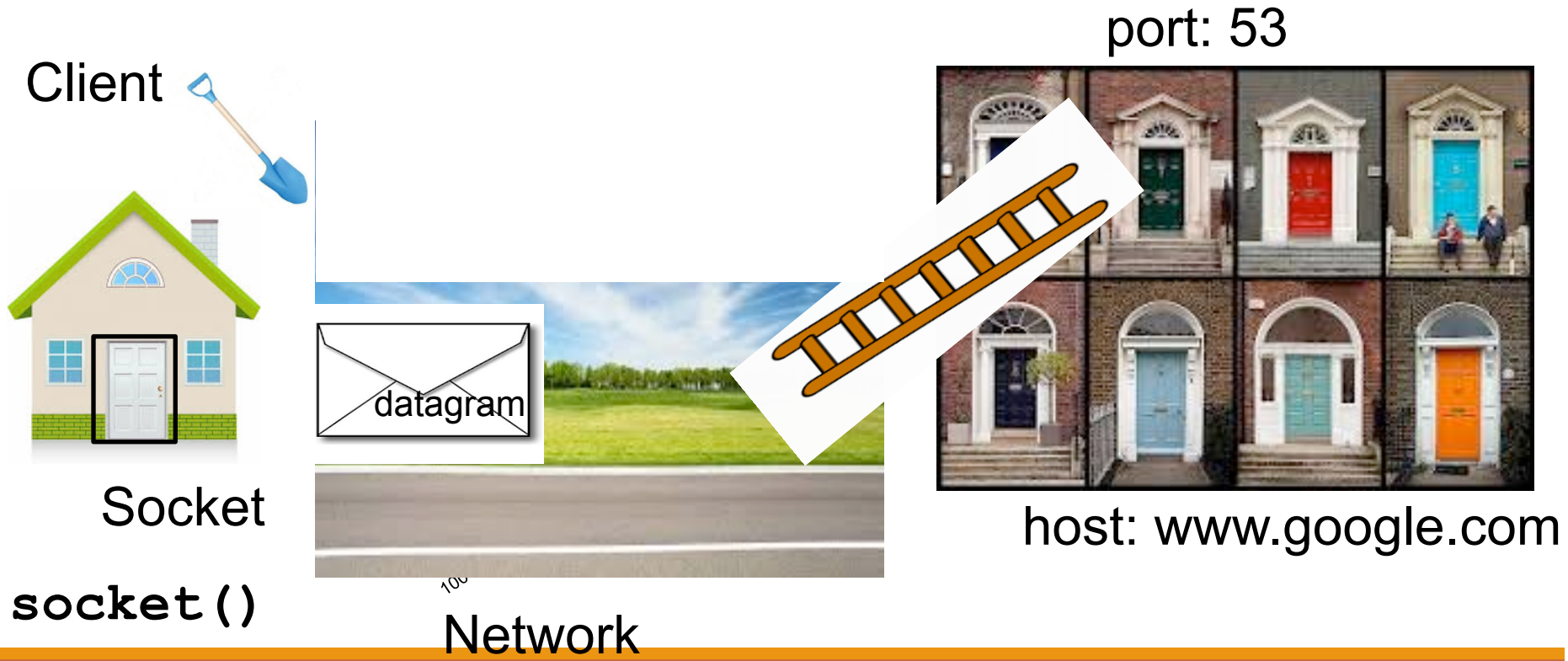
port: 80

host: www.google.com

`connect(host, port)`

UDP sends the message as a whole chunk called Datagram

Create message



User Datagram Protocol (UDP)

Each UDP message is self-contained and complete

Each time you read from a UDP socket, you get the complete message as sent by the sender

- *That is, assuming it wasn't lost in transit!*

Think of UDP sockets as putting a stamp on a letter and sticking it in the mail

- *No need to establish a connection first*
- *Receiver has no idea "letter" is arriving until they look in the mailbox*

Python UDP Programming

Two new functions: `sendto()` and `recvfrom()`

```
server_ip = 1.2.3.4
port = 5678
dest_addr = (server_ip, port)
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
...
...
bytes_sent = s.sendto(raw_bytes, dest_addr)
...
...
max_bytes = 4096
(raw_bytes, src_addr) = s.recvfrom(max_bytes)
```

let's focus on its construction

You receive `raw_bytes` in a single shot. No need to set up a loop!

Warm Up with WireShark

dig www.stanford.edu:

dig www.stanford.edu A @8.8.8.8 +noedns

Monitor the packets using WireShark. Inspect the query packet sent from you and write the values for Transaction ID, flags, Question, Answer RRs, Additional RRs, Query, type, and class.

NOTE: In dns.py, you will create query packets that look very much like the above!

Several ways to create Query packet

#1 – explicitly append bytes to an array of bytes

Query Packet:

Field	Bytes	Value
Transaction ID	2	Random value, you choose!
Flags	2	0x 01 20
Questions	2	0x 00 01
Answer RRs	2	0x 00 00
:		:
Query		variable, but each char is 1B
Type	2	0x 00 01 (A) 0x 00 1c (AAAA)
Class	2	0x 00 01

Create array of bytes using bytearray()

`bytearray()` returns an array of bytes.

```
>>>raw_bytes=bytearray() #try it
```

You can do lot of neat stuff with byte arrays:

- Append hexadecimal byte (for constant message fields):

  ```
  raw_bytes.append(0xfe) #append fe. try it
  ```

- Concatenate strings as bytes (for the query):

  ```
  >>>string_bytes=bytes(string,'ascii')
  >>>raw_bytes+=string_bytes
  ```

Google search Python’s in-built ord function and write what it does. How can you use it in Lab 09?

Exercise 1: Encode qname

Create a string `qname` (query name) and assign it to engineering.pacific.edu. Create a byte array called `raw_string`, which prints as under. Make sure to append 0x00 in the end (why?). Print it. For a query name engineering.pacific.edu, the output should look like this:

```
>>> print(raw_string)
bytearray(b'\x0bengineering\x07pacific\x03edu\
x00')
```

Note the similarity with the query section when you captured the DNS query using WireShark!

It is so straightforward!

In our UDP messages, all of the values are under 255. The lengths of the subdomains were also less than 255

Open the python3 interpreter and do the following:

- declare a byte array called message
- Try appending the value 1024. Successful?

Need something else to be able to append values that take up more than one byte!

#2 -- The `struct` Module

Two main functions in the `struct` module

- `pack`: convert a group of variables into an array of bytes
- `unpack`: convert an array of bytes into a group of variables

Similar to C's `printf` and `scanf`

Each function requires a *format string* to describe how to pack or unpack the arguments

The `struct` Module

Common format string options:

- See https://docs.python.org/3/library/struct.html

```
>>>import struct
```

Format	Python Type	Size (bytes)
B	Integer	1
H	Integer	2
L	Integer	4
Q	Integer	8

```
>>>val1=50
>>>val2=1024
>>>raw_bytes = struct.pack("BH", val1, val2)
>>>print(raw_bytes) #what do you see? What is the endianness?
>>>(val1, val2) = struct.unpack("BH", raw_bytes)
```

The `struct` Module

Endianness must be considered when doing file or network I/O with fields greater than one byte

The first character of the format string determines the endianness

Character	Byte order	Size	Alignment
@	Native	Native	Native
=	Native	Standard	None
<	Little	Standard	None
>	Big	Standard	None
!	Network (Big)	standard	None

DNS Endianness

What endianness is your computer?

- Little endian (x86)

What endianness is the DNS protocol?
(or most network protocols)

- Big endian

What fields in the DNS header does this matter for?

- Two-byte integer fields
  (question count, answer count, etc…)
- You can explicitly append these bytes in big endian format

You can also pack a string

1. Strings need to be packed as bytes, so convert them to bytes first. Use `bytes()`
2. `raw_bytes= struct.pack('!<length of string>s',bytestring)`

Try Packing engineering.pacific.edu by:

- converting it to bytes using `bytes()`
- use `struct.pack` as in 2 above

Receiving UDP message from the server

Import the `dns` class from dns_tools.py (see boilerplate code for lab09). Use `decode_dns` function to decode the UDP message.

One function call and that's it!

In-Class Participation

Pull the boilerplate and find `dummy_query.py` inside the Practice folder. Inspect first few lines to see sample usage. Create a dummy request called `raw_bytes` as in the diagram. Use qname="www.google.com"

A randomly chosen Transaction ID (2 Bytes)	FLAGS (2 Bytes) set to 0x 01 20	Encoded qname with 0x 00 in the end (variable bytes)	TYPE (2 Bytes) 0x 00 01 if qtype is A 0x 00 1c if qtype is AAAA

Get 5 extra points on in-class participation if you can perform it using the struct module alone

Bit Fields

Warning! `struct` only deals with bytes. It cannot handle fields with dimensions less than one byte

Problem – Some of the DNS fields (FLAGS) are composed of 1 bit, 3 bits, or 4 bits fields

How can we handle this?

- Manual bit shifting using **ctypes**

```
QR | OPCODE | AA | TC | RD | RA | Resvd | RCODE
(1)    (4)    (1)  (1)  (1)  (1)  (3)      (4)
```

CTypes

```
import ctypes

# Define a 2-byte structure (equivalent to a 'uint16' variable in C)
class CustomStruct(ctypes.BigEndianStructure):
    _fields_ = [
        ("fieldA", ctypes.c_uint16, 1),  # 1-bit field - Most Sig BIT
        ("fieldB", ctypes.c_uint16, 6),  # 6-bit field
        ("fieldC", ctypes.c_uint16, 4),  # 4-bit field
        ("fieldD", ctypes.c_uint16, 5)   # 5-bit field - Least SIG BIT
    ]

# Create new instance of the 'CustomStruct' data type
special_variable = CustomStruct()

# Access the fields of the structure
special_variable.fieldA = 1
special_variable.fieldB = 18
special_variable.fieldC = 5
special_variable.fieldD = 17
```

CTypes

```
# Print out individual fields
print("Field A = %i" % special_variable.fieldA)
print("Field B = %i" % special_variable.fieldB)
print("Field C = %i" % special_variable.fieldC)
print("Field D = %i" % special_variable.fieldD)

# Convert the structure to a byte array and print it out
print(bytes(special_variable))

# Alternate printing method (won't decode bytes as ASCII)
hex_string = "".join("%02x " % b for b in bytes(special_variable))
print("0x%s" % hex_string)
```