Socket Programming for Inter-process Communication

CPEN333 - 2023 W1

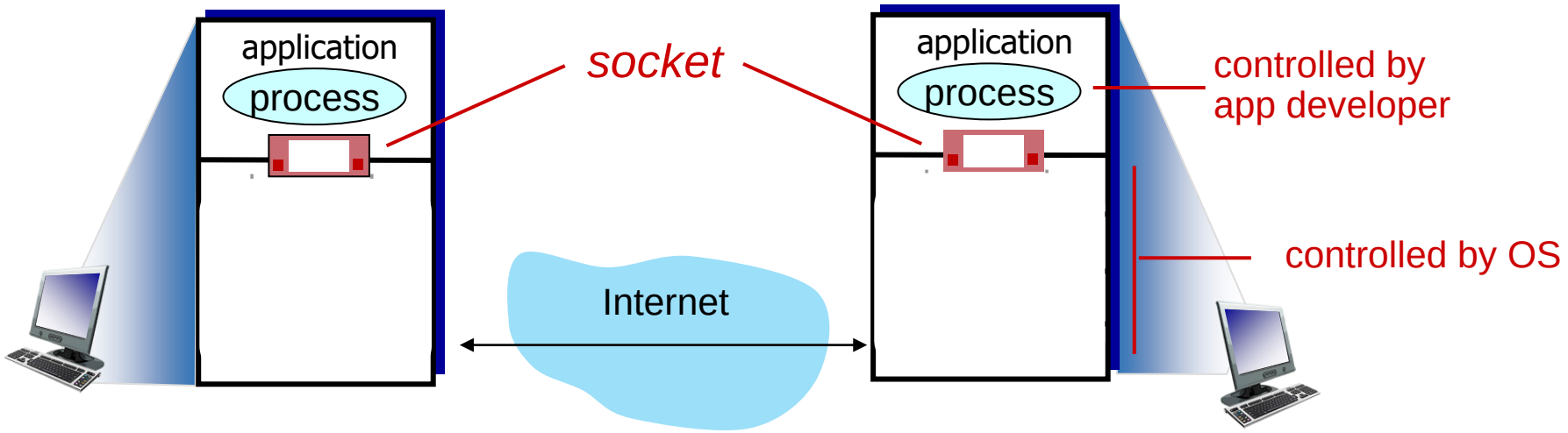
University of British Columbia

Introduction

- Sockets are popular and capable mechanism for inter-process communication.
  - A socket is an endpoint for communication and a pair of sockets are usually used in a client-server paradigm.
- Sockets allow two processes to communicate.
  - Those processes could be on the same machine (locally) or they could be on two machines on either side of the world (remotely, e.g. Internet).
- There are different types of sockets.
  - We focus on the two widely-used stream (TCP) and datagram (UTP) sockets.
- We will use Python's `socket` module in this set of slides.

Objectives

- To examine socket programming as an inter-process communications mechanism
- To implement client/server communication using python's `socket`



Socket programming

- There are different types of sockets. We focus on stream (TCP) and datagram (UTP) sockets.
  - TCP: reliable, byte stream-oriented
  - UDP: unreliable datagram

- A simple application Example:
  1. client reads a line of characters (data) from its keyboard and sends data to server
  2. server receives the data and converts characters to uppercase
  3. server sends modified data to client
  4. client receives modified data and displays line on its screen

Python's **socket** module

- Python's socket module is a low-level networking interface that provides access to the BSD socket interface.
  - Availability: all modern OS
  - https://docs.python.org/3/library/socket.html

- It is a direct transliteration of the Unix's socket and library interfaces.
  - So by learning this, we get familiarity with most other socket programming libraries.

socket.socket

- `class socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)`
  - Creates a new socket using the given address family, socket type and protocol number.
  - https://docs.python.org/3/library/socket.html#socket.socket

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be `AF_INET` (the default), `AF_INET6`, `AF_UNIX`, `AF_CAN`, `AF_PACKET`, or `AF_RDS`. The socket type should be `SOCK_STREAM` (the default), `SOCK_DGRAM`, `SOCK_RAW` or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted or in the case where the address family is `AF_CAN` the protocol should be one of `CAN_RAW`, `CAN_BCM`, `CAN_ISOTP` or `CAN_J1939`.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of `socket.getpeername()` but not the actual OS resource. Unlike `socket.fromfd()`, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using `socket.close()`.

Socket families

- The socket module defines various socket families.
  - https://docs.python.org/3/library/socket.html#socket-families

- We focus on the `(host, port)` pair used in the `AF_INET` address family.
  - This is the widely-used format in the Internet for the IP version 4 (IPv4) protocol.
  - If we want to work with IPv6 (IP version 6), we can use `AF_INET6` address family.

- Why is this important?
  - This `AF_INET` constant is used at the time of socket creation to specify the type of socket we want to create.

Important methods

- The function we use may depend on whether it is a UDP or a TCP socket.
  - It also depends on whether it is the client socket or the server socket.
- At the simplest form we may have the following:
  - **UDP client uses**: `socket, sendto, recvfrom, close`
  - **UDP server uses**: `socket, bind, recvfrom, sendto`
  - **TCP client uses**: `socket, connect, send, recv, close`
  - **TCP server uses**: `socket, bind, listen, accept, recv, send, close`
- These functions are used to:
  - Create a socket
  - Use the socket to send or/and receive data
  - Close the socket

socket.sendto and socket.recvfrom

- `socket.sendto(bytes, address)`
  - Sends data to the socket.
    - Generally used with the `SOCK_DGRAM` sockets.
    - The format of *address* depends on the address family
    - The destination socket is specified by *address*
    - Return the number of bytes sent
  - There is also the overload: `socket.sendto(bytes, flags, address)`
  - https://docs.python.org/3/library/socket.html#socket.socket.sendto

- `socket.recvfrom(bufsize[, flags])`
  - Receives data from the socket.
    - Generally used with the `SOCK_DGRAM` sockets.
    - The return value is a pair (bytes, address) where bytes is a bytes object representing the data received and address is the address of the socket sending the data.
  - https://docs.python.org/3/library/socket.html#socket.socket.recvfrom

  - `flags` is from the UNIX recv(2), and defaults to zero

socket.recv and socket.send

- `socket.send(bytes[, flags])`
  - Sends data to the socket.
    - Generally used with the `SOCK_STREAM` sockets.
  - https://docs.python.org/3/library/socket.html#socket.socket.send

- `socket.recv(bufsize[, flags])`
  - Receives data from the socket.
    - Generally used with the `SOCK_STREAM` sockets.
    - The maximum amount of data to be received at once is specified by *bufsize*.
    - The return value is a bytes object representing the data received.
  - https://docs.python.org/3/library/socket.html#socket.socket.recv

socket.bind and socket.listen and socket.accept

- `socket.bind(address)`
  - Binds the socket to address (The socket must not already be bound)
    - The format of *address* depends on the address family
  - https://docs.python.org/3/library/socket.html#socket.socket.bind

- `socket.listen([backlog])`
  - Enable a server to accept connections.
    - *backlog* is the number of unaccepted connections that the system will allow before refusing new connections
  - https://docs.python.org/3/library/socket.html#socket.socket.listen

- `socket.accept()`
  - Accepts a connection.
    - There is also socket.connect(address)
  - https://docs.python.org/3/library/socket.html#socket.socket.accept

socket.connect

- socket.connect(address)
  - Connect to a remote socket at address.
  - https://docs.python.org/3/library/socket.html#socket.socket.connect

socket.connect(*address*)

Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a `TimeoutError` on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an `InterruptedError` exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Raises an auditing event `socket.connect` with arguments `self`, `address`.

socket.close

- socket.close()
  - Marks the socket closed.
  - https://docs.python.org/3/library/socket.html#socket.socket.close

`socket.close()`

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from `makefile()` are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly, or to use a `with` statement around them.

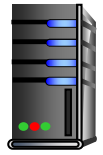
Socket programming with UDP

- We use the `socket.SOCK_DGRAM` constant in the python's socket module to identify this type of socket.
- UDP: no "connection" between client & server
  - no handshaking before sending data
  - sender explicitly attaches IP destination address and port # to each packet
  - receiver extracts sender IP address and port# from received packet
- UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: UDP client

Python UDPClient

include Python's socket library →
```
from socket import *
serverName = "hostname"    #replace with actual name
serverPort = 12000         #use an available port
```
create UDP socket for server →
```
clientSocket = socket(AF_INET,  SOCK_DGRAM)
```
get user keyboard input →
```
message = input("Input lowercase sentence:")
```
attach server name, port to message; send into socket →
```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```
read reply characters from socket into string →
```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```
print out received string and close socket →
```
print (modifiedMessage.decode())
clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

create UDP socket →

bind socket to local port number 12000 →

loop forever →

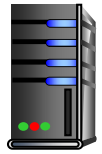
Read from UDP socket into message, getting client's address (client IP and port) →

send upper case string back to this client →

Socket programming with TCP

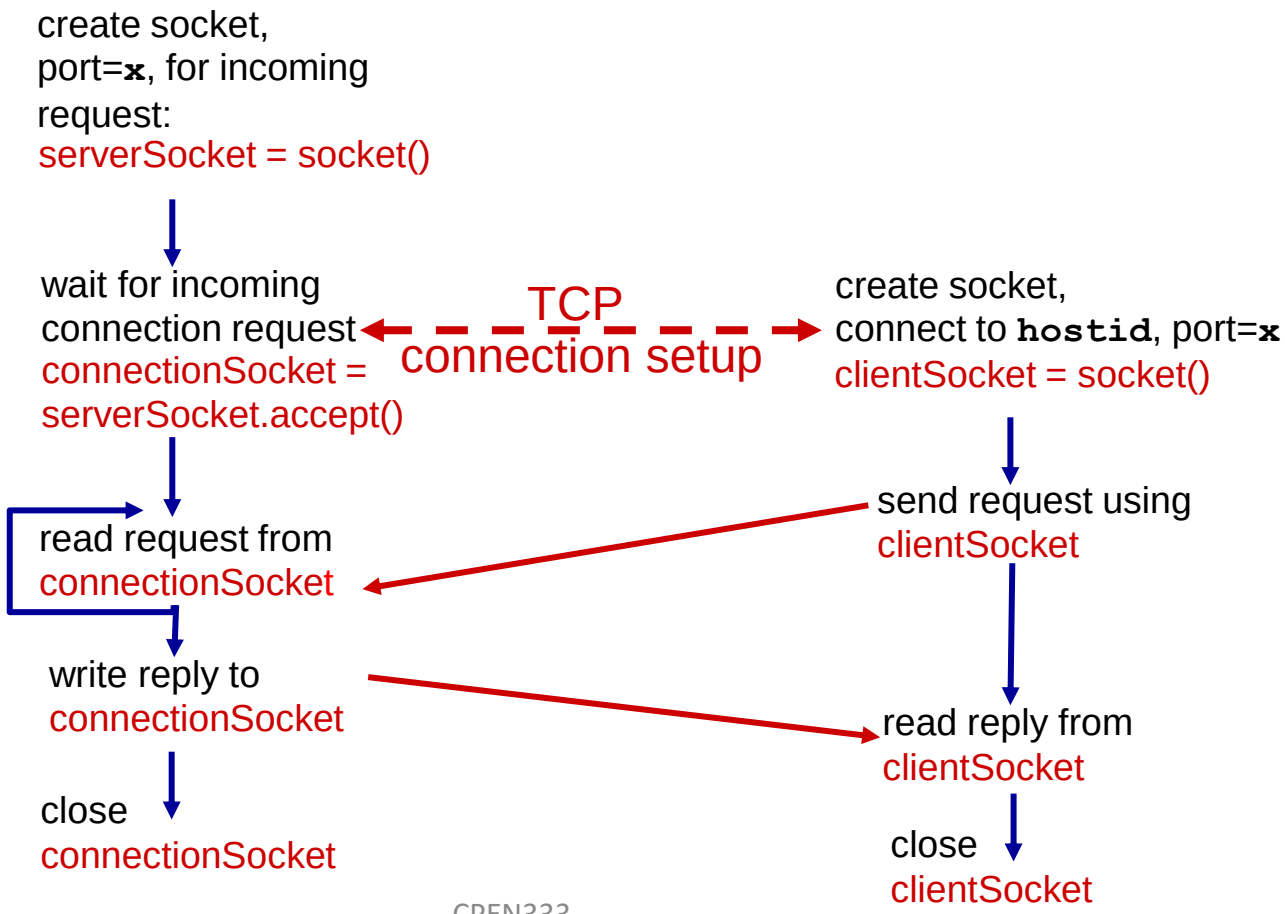
- We use the `socket.SOCK_STREAM` constant in the python's socket module to identify this type of socket.
- Client must contact server
  - server process must first be running
  - server must have created socket (door) that welcomes client's contact
- Client contacts server by:
  - Creating TCP socket, specifying IP address, port number of server process
  - when client creates socket: client TCP establishes connection to server TCP
- when contacted by client, server TCP creates new socket for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)
- Application viewpoint
  - TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



server (running on hostid)

client 



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = "servername"  #replace with actual name
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input("Input lowercase sentence:")
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ("From Server:", modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print ("The server is ready to receive")
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                        encode())
    connectionSocket.close()
```

create TCP welcoming socket →

server begins listening for incoming TCP requests →

loop forever →

server waits on accept() for incoming requests, new socket created on return →

read bytes from socket (but not address as in UDP) →

close connection to this client (but *not* welcoming socket) →

References

- Python documentation:
  - https://docs.python.org/3/library/socket.html
  - https://docs.python.org/3.10/howto/sockets.html

- *Operating System Concepts*, Silberschatz et al, section 3.6

- *Computer Networking*, by Kurose and Ross, Section 2.7