universidade de aveiro
theoria poiesis praxis

REDES DE COMUNICAÇÕES 1

LABORATORY GUIDE

Objectives

- Sockets (in Python)
    - UDP Sockets
    - TCP Sockets with textual data, fixed sized binary data packets, and variable sized binary data packets.

Duration

- 1 week

UDP Sockets (Connection-Less data transmission)

1.1. Start the provided UDP server (*serverUDP.py*), which will open an UDP Socket, listen in all available IPv4 interfaces/addresses (using the IPv4 address 0.0.0.0) in port 5005 , and print all messages from clients.

1.2. Start a Wireshark capture on the server machine and analyze the received UDP packets. Start the provided UDP client (*clientUDP.py*) on the same machine or on a remote machine changing the server IPv4 address (variable *ip_addr*). From one (or more clients) send messages to the server. Analyze the code from both server and client. Explain the usage/choice of the source UDP ports by the client(s).

TCP Sockets (Connection Oriented data transmission)

Textual data messages and clients handled with Threads

2.1. Start the provided TCP server (*serverTCP.py*), which will open an TCP Socket, listen in all available IPv4 interfaces/addresses (using the IPv4 address 0.0.0.0) in port 5005, print all messages from clients and send an ECHO message back to the client. This server expects “messages with textual data”.

2.2. Start a new Wireshark capture on the server machine and analyze the received TCP packets. Start the provided TCP client (*clientTCP.py*) on the same machine or on a remote machine changing the server IPv4 address (variable *ip_addr*). From one (or more clients) send messages to the server. Analyze the code from both server and client. Explain the usage/choice of the source TCP ports by the client(s), how the sessions are created, and how different clients are handled by different threads.

Textual data messages and clients handled with Selector

3.1. Start the provided TCP server (*serverTCPsel.py*), which will open an TCP Socket, listen in all available IPv4 interfaces/addresses (using the IPv4 address 0.0.0.0) in port 5005, print all messages from clients and send an ECHO message back to the client. This server expects “messages with textual data”.

3.2. Start a new Wireshark capture on the server machine. Start the provided TCP client (*clientTCP.py*) on the same machine or on a remote machine changing the server IPv4 address (variable *ip_addr*). Analyze the code from both server and client. Explain know how different clients are handled by the server using Selectors and Selector keys.

Binary fixed size data messages and clients handled with threads

4.1. Start the provided TCP server (*serverTCPv2.py*), which will open an TCP Socket, listen in all available IPv4 interfaces/addresses (using the IPv4 address 0.0.0.0) in port 5005, and print all messages from clients. This server expects “messages with binary fixed size data”, where the header/data structure is: 1 byte for the protocol version, two unsigned longs (2x32 bytes) to packet order and original message size, and 20 chars/bytes to carry the message.

Note: the data structure is defined using the package *struct*. See more information: https://docs.python.org/3/library/struct.html

4.2. Start a new Wireshark capture on the server machine. Start the provided TCP client (*clientTCPv2.py*) on the same machine or on a remote machine changing the server IPv4 address (variable *ip_addr*). Analyze the code from both server and client. Explain how data is being sent and decoded.

4.3. Change the server/client code to include a server ECHO response.

Binary variable size data messages and clients handled with threads

5.1. Start the provided TCP server (*serverTCPv3.py*), which will open an TCP Socket, listen in all available IPv4 interfaces/addresses (using the IPv4 address 0.0.0.0) in port 5005, and print all messages from clients. This server expects “messages with binary variable size data”, where the header/data structure is: 1 byte for the protocol version, two unsigned longs (2x32 bytes) to packet order and original message size, and a number of chars/bytes (define by the size field) to carry the message.

Note: the data structure is defined using the package *struct*. See more information: https://docs.python.org/3/library/struct.html

5.2. Start a new Wireshark capture on the server machine. Start the provided TCP client (*clientTCPv3.py*) on the same machine or on a remote machine changing the server IPv4 address (variable *ip_addr*). Analyze the code from both server and client. Explain how data is being sent and decoded.

5.3. Change the server/client code to include a server ECHO response.

1.2 - By not specifying a source port, the client code is simpler and avoids potential conflicts with other applications that might be using specific ports. The OS manages the ephemeral ports, ensuring that they are unique and available, reducing the risk of port conflicts. Each client instance can run on the same machine without worrying about port conflicts, as each will be assigned a different ephemeral port.

2.2 - The source port for the client is chosen dynamically by the operating system. When the connect call is made, the OS assigns an ephemeral port from a range (e.g., 49152–65535) to the client socket as its source port. This ensures a unique client-server pairing. This ephemeral source port enables multiple clients to connect to the same server port without conflict.

The server listens for incoming connections on a specific port (5005). When a client attempts to connect, the server invokes server.accept(). The server accepts the connection and creates a new socket (client_sock) dedicated to communication with the specific client. A session is defined by the unique tuple of (server IP, server port, client IP, client port). This ensures that even if multiple clients are connected, their communication is not mixed up.

The threading mechanism allows the server to manage multiple clients concurrently. Each thread handles:
Receiving data (client_socket.recv()).
Sending responses (client_socket.send()).
The thread continues running as long as the client is actively sending data.

3.2 - Selectors provide an efficient way to monitor multiple sockets without spawning multiple threads. Unlike the thread-based server (serverTCP.py), the selector-based server does not create a new thread for each client. This reduces memory usage and context-switching overhead.
It can efficiently handle thousands of simultaneous connections.

All sockets operate in non-blocking mode, ensuring that no socket operation blocks the entire server.

New sockets (clients) are dynamically registered with the selector, and their callbacks are invoked when events occur.

4.2 - The client packs data into a structure using struct.pack. The format string !BLL20s specifies the expected data types and order:

!: Network byte order (important for compatibility across different systems).

B: Unsigned char (1 byte) for version.

L: Unsigned long (4 bytes) for order.

L: Unsigned long (4 bytes) for size.

20s: String of 20 bytes for the message.

The client constructs the message as follows:
It gets input from the user.
It calculates the size of the message.
It pads the message with '-' characters if it's shorter than 20 characters.
It encodes the message to bytes using .encode().
It packs the version, order, size, and the first 20 characters of the message into a single byte string (pkt).

Data Sending: The client sends the packed data (pkt) to the server using sock.send(pkt).

SERVER:

The server receives data from the client using client_socket.recv(29). Crucially, it receives a maximum of 29 bytes. This is important because the maximum size of the data it expects to receive is 20 bytes for the message plus 4 bytes for the size, 4 bytes for the order, and 1 byte for the version.

The server unpacks the received data using struct.unpack('!BLL20s', request). This unpacks the received bytes into the version, order, size, and message components.

The server prints the unpacked data, including the version, order, size, and the decoded message.