

LABORATORY WORK

Application Layer: Network programming with sockets

1. Objectives

Prerequisite: Use a working software environment for your preferred programming language (Java, C#, Python, C/C++, etc.)

At the end of the activity, students will be able to write software for socket applications and debug network applications using Wireshark.

2. Theoretical considerations

The current practical work focuses on the Transport and Application layers of the ISO/OSI stack (Figure 8.1).

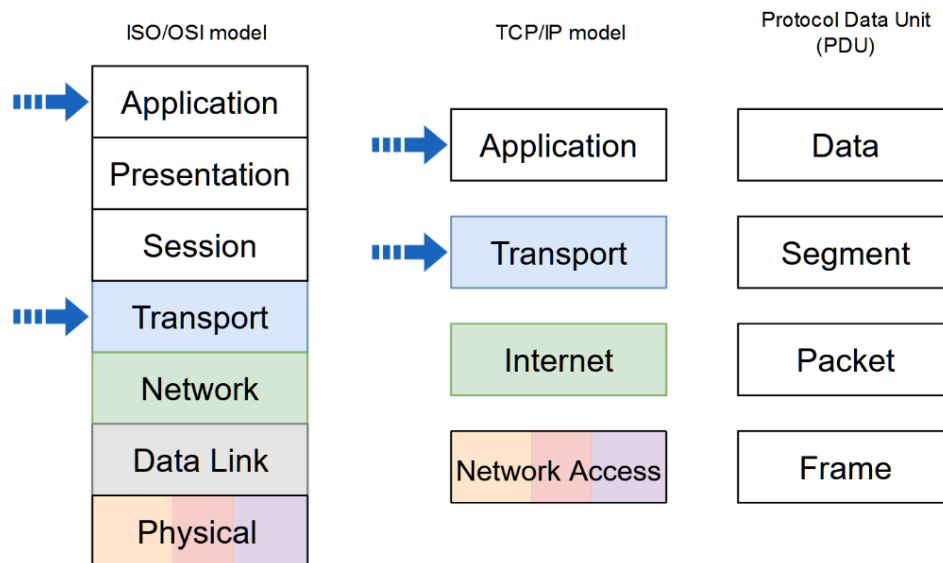


Figure 8.1 Network stack models and PDU naming in each level. The arrows indicate the addressed layers in the current activity

This practical activity addresses the programming side of software engineering and communication offered through the use of network sockets in a desktop environment. Socket programming is available in any high level programming language and sockets are transmitting information at the Application Layer. Sockets are used in different types of applications, such as: Client-Server, peer-2-peer systems, inter-process communication (on the same machine).

Network sockets can be constructed to use both IPv4 and IPv6 addresses. A socket is the combination of an IP address and a port number for use in a network application. A network application provides connectivity between different network devices. It is not possible to bind a socket to a port that is already in use by any other application, however the same port may be used concurrently by TCP and UDP transport layer protocols. The IP addresses identify the

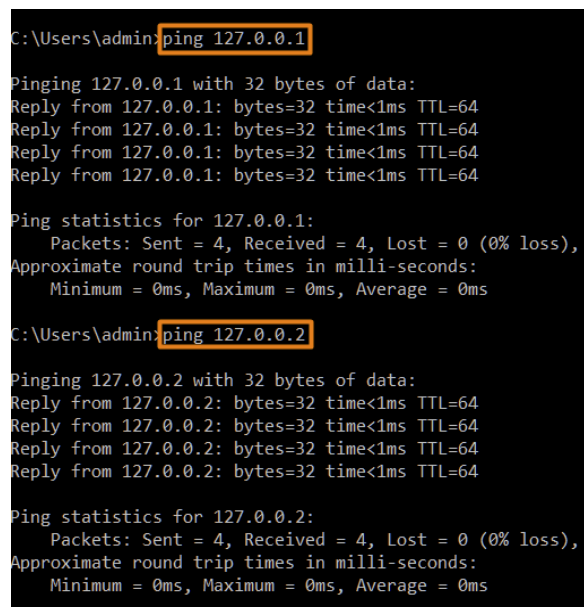
network device, but the port number uniquely identifies each running application on the current network device.

The operations that an application can perform on a socket are the following:

- **Create** - Creation of a socket object
- **Bind** - Configure the socket object to use a local pair of IP address and port number to accept connections
- **Listen** - Program the socket to wait for incoming connections
- **Accept** - Accept the incoming connection
- **Connect** - This operation is used by a client that wants to connect to a server
- **Send** - Used to send data over the socket to the remote destination
- **Receive** - Used to receive data which is sent from a remote location
- **Close** - close the connection between the two sockets

2.1. Working with sockets on the local machine

- In order to simulate a network on the local machine, the entire available loopback range: 127.0.0.0 - 127.255.255.255 can be used. The loopback network interface is available only on the local host and is mainly used for diagnostics and standalone network applications. Therefore, a simulated local network can use these IP addresses for communication. In order to test and confirm that this range can be used, a **ping** command can be run from the local terminal to verify connectivity to said IP addresses (Figure 8.2):

A screenshot of a Windows command prompt window with a black background and white text. The prompt shows the user running the 'ping' command to two local loopback IP addresses. The first command is 'ping 127.0.0.1', which returns four successful replies with 32 bytes of data, a time of less than 1ms, and a TTL of 64. The statistics show 4 packets sent, 4 received, and 0% loss. The second command is 'ping 127.0.0.2', which also returns four successful replies with identical statistics. The user's path is shown as 'C:\Users\admin\'.

```
C:\Users\admin>ping 127.0.0.1

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=64
Reply from 127.0.0.1: bytes=32 time<1ms TTL=64
Reply from 127.0.0.1: bytes=32 time<1ms TTL=64
Reply from 127.0.0.1: bytes=32 time<1ms TTL=64

Ping statistics for 127.0.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Users\admin>ping 127.0.0.2

Pinging 127.0.0.2 with 32 bytes of data:
Reply from 127.0.0.2: bytes=32 time<1ms TTL=64
Reply from 127.0.0.2: bytes=32 time<1ms TTL=64
Reply from 127.0.0.2: bytes=32 time<1ms TTL=64
Reply from 127.0.0.2: bytes=32 time<1ms TTL=64

Ping statistics for 127.0.0.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Figure 8.2 *Loopback addresses testing*

- It is also possible to assign multiple valid IP addresses on the local interface, but this has to be done manually by statically allocating IP addresses to the interface. In this

case, running the **ipconfig** command would show all the IP addresses assigned to the same interface. See an example below (Figure 8.3):

```
Wireless LAN adapter Wi-Fi 2:

Connection-specific DNS Suffix . : 
Link-local IPv6 Address . . . . . : fe80::fce4:5011:a37b:f929%22
IPv4 Address. . . . . : 10.0.0.1
Subnet Mask . . . . . : 255.0.0.0
IPv4 Address. . . . . : 10.0.0.2
Subnet Mask . . . . . : 255.0.0.0
IPv4 Address. . . . . : 20.0.0.1
Subnet Mask . . . . . : 255.0.0.0
IPv4 Address. . . . . : 20.0.0.2
Subnet Mask . . . . . : 255.0.0.0
IPv4 Address. . . . . : 30.0.0.1
Subnet Mask . . . . . : 255.0.0.0
IPv4 Address. . . . . : 30.0.0.2
Subnet Mask . . . . . : 255.0.0.0
IPv4 Address. . . . . : 172.16.0.1
Subnet Mask . . . . . : 255.255.0.0
IPv4 Address. . . . . : 172.16.0.2
Subnet Mask . . . . . : 255.255.0.0
IPv4 Address. . . . . : 172.16.0.3
Subnet Mask . . . . . : 255.255.0.0
IPv4 Address. . . . . : 192.168.0.35
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : fe80::8f3:92ff:fe9b:5d34%22
```

Figure 8.3 *IP configuration view - CLI*

- After having assigned the IP addresses, sockets can now be created to use these IP addresses.

2.2. TCP Sockets

- TCP (Transmission Control Protocol) sockets are connection oriented and represent a reliable data transmission mechanism that allows data to be received and processed in the same order it was transmitted.
- The Figure 8.4 shows a Wireshark traffic capture on the “Adaptor for loopback traffic capture”. The screenshot shows a client-server communication via sockets using the loopback addresses. The applied filter is **tcp.port == 1234**. The server is bound to the 127.0.0.1 address and awaits connections on port number 1234 while the client binds on the 127.0.0.2 address sending a payload of 14 bytes to the server.

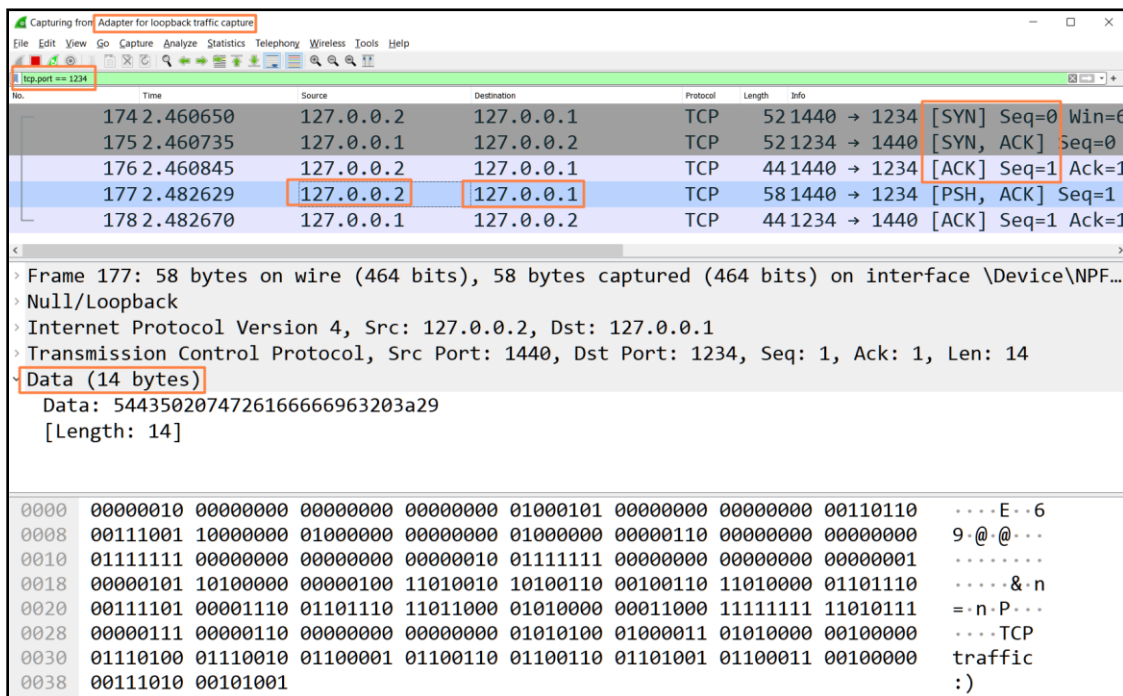


Figure 8.4 Wireshark capture of *TCP socket communication*

- The screenshot highlights the TCP mechanism represented by acknowledgement (ACK) packets. The first three packet exchanges (Figure 8.4) represent the 3-way handshake which is needed to establish the connection for any TCP connection (Figure 8.5) and following that the packet sending the payload is visible. This handshake assures that both hosts want to communicate and acknowledge the other host's intention to communicate.

3-way handshake

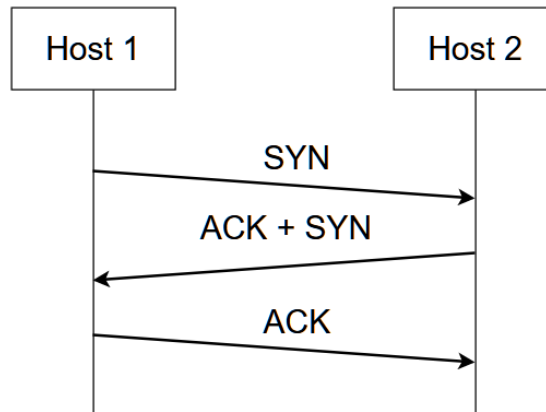


Figure 8.5 *TCP 3-way handshake*

- The Wireshark image shows a socket communication which remains open.
 - Answer the following question while working on the practical activity: If the socket connection is closed, what are the TCP flags that are set in order to close the connection?

2.3. UDP Sockets

- In contrast with the TCP sockets, UDP (User Datagram Protocol) sockets are not connection oriented and they do not provide reliable communication. This means they do not guarantee that network packets are delivered to the destination. The Figure 8.6 represents a Wireshark capture (again on the “Adaptor for loopback traffic capture”) of a UDP communication between two hosts. The applied filter is **udp.port == 1234**. As can be seen, there is no handshake performed and there aren’t any ACK packets being transmitted.

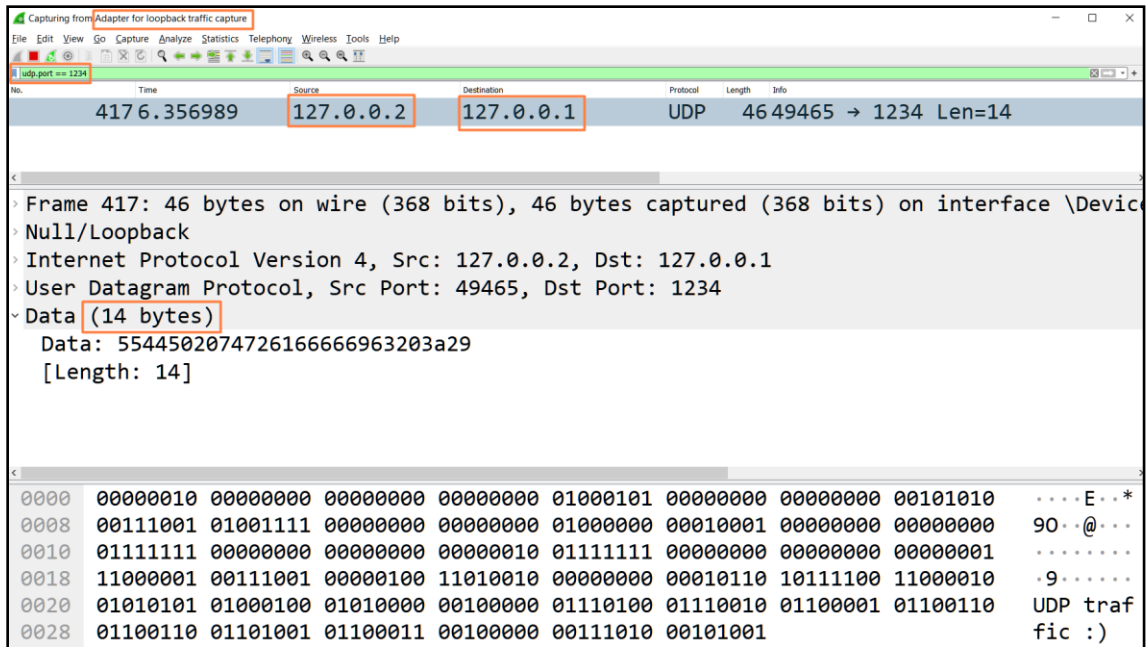


Figure 8.6 Wireshark capture of *UDP socket communication*

TCP and UDP socket communications are presented in Figure 8.7 a and b.

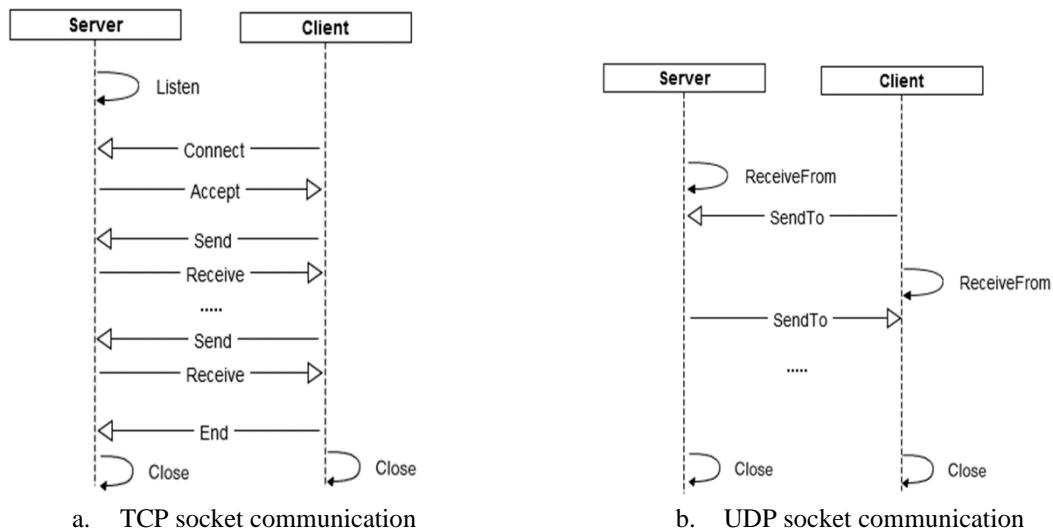


Figure 8.7 *Socket communications*

1. Implementation template

- A network device can function in 3 modes:
 - **Server:** Receiving device
 - **Client:** Sending device
 - **Relay:** Acting as an intermediary node in a communication and acts as both sending and receiving device. This type of network node can be encountered in Wireless Sensor Networks (WSN) where not all sensor nodes are in the wireless range of the collector device, therefore some nodes need to forward the information to the sink node (the collector node).
- This chapter provides a template for implementing the relay communication node using OOP concepts (the template is written in pseudocode, not in a particular programming language). This is not the only possibility to organize the code, students can choose any software design methodology they are comfortable with.

Relay node implementation template

```
class RelayNode {
public:
    RelayNode(IPAddress, serverPortNr) {
        m_server.listen(IPAddress, serverPortNr);
        m_client.bind(IPAddress);

        m_server.onReceive() => {
            ByteArray receivedData = m_server.readData();
            m_client.connectToHost(m_nextHopIpAddress, m_nextHopPortNr);
            m_client.sendData(receivedData);
            m_client.close();
        }
    }
    void setNextHopInformation(nextHopIpAddress, nextHopServerPortNr) {
        m_nextHopIpAddress = nextHopIpAddress;
        m_nextHopPortNr = nextHopServerPortNr;
    }

private:
    Server m_server; // server instance accepting connections
    Client m_client; // client instance sending data to the next hop
    IPAddress m_nextHopIpAddress; // next hop address used by the client instance
    int m_nextHopPortNr; // next hop port nr used by the client instance
}

void main() {
    RelayNode relay(127.0.0.1, 1234);
    relay.setNextHopInformation(127.0.0.2, 2345)
    ...
    // run application event loop
}
```

3. Practical activity

- Each student will be assigned one of the topologies below and the simulation scenario has to be implemented in software.
- Besides the constraints imposed by each simulation scenario, the common tasks for each implementation are the following:
 - Use a programming language of choice to implement the network simulation
 - Use the loopback address range for addressing: 127.0.0.0 – 127.255.255.255
 - Test the implementation using Wireshark
 - Deliver the implementation (source code or link to online code versioning repository)
 - Provide a Wireshark capture to prove the communication between different IP addresses
 - Inspect the ratio between total delivered payload against the relevant application layer traffic / the ratio between the total packet length (headers and data) compared to the length of data sent (use Wireshark statistics or manual packet inspection)
 - Depending on the implemented simulation, research the headers for TCP and/or UDP protocols. Using Wireshark, identify the header elements in the captured traffic

3.1 Ring communication

- Three computers are communicating in a single direction creating a loop (Figure 8.8)
- One of the computers initiates the communication sending the value '1'
- Upon receipt, each network device increments the received value and sends it to the next device
- The communication ends when the delivered payload reaches the value '100'
- Implementation hints:
 - Implement a single class which is instantiated 3 times with different communication parameters (reuse the code and do not duplicate it for each instance)
 - All communication uses TCP sockets (optional)

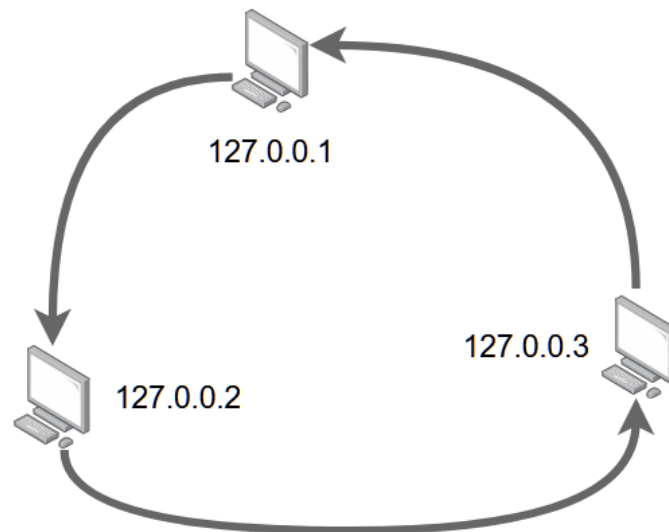


Figure 8.8 *Ring communication network topology*

3.2 Node selector

- There are three nodes in the topology: N1, N2, N3 (Figure 8.9)
- N1 increments a value 100 times and after every increment it sends the value to either N2 or N3 which are selected randomly for transmission
- When N2 receives an integer value which is a multiple of 3 it will send an ACK packet back to N1
- When N3 receives an integer value which is a multiple of 5 it will send an ACK packet back to N1
- Implementation hints:
 - Implement a single class for N2 and N3 which is instantiated with different communication parameters (reuse the code and do not duplicate it for each instance)
 - All communication uses UDP sockets (optional)

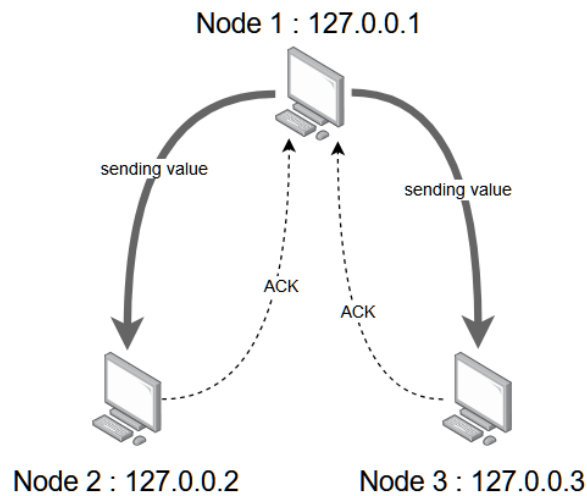


Figure 8.9 *Node selector network topology*

3.3 Relay nodes

- There are four nodes in the topology (Figure 8.10), Sender and three possible destinations (D1, D2, and D3)
- The Sender node is transmitting 100 packets containing an integer number randomly to one of the 3 possible destinations (D1, D2 or D3)
- After each packet transmission the integer number is incremented
- Every node can only send data to the next hop to which it is connected to, therefore a packet from the Sender to D3 must pass through D1 and D2

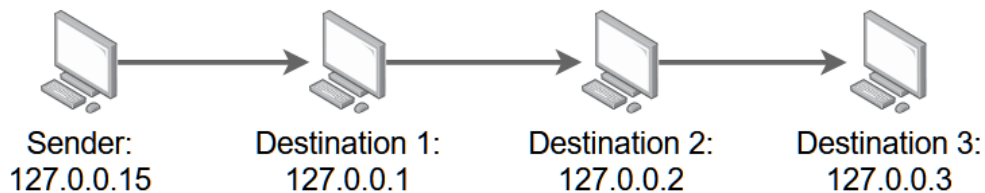


Figure 8.10 *Relay nodes network topology*

- Implementation hints:
 - The data payload that is transmitted via the socket has to contain the target IP address, so the payload has the following format (Figure 8.11):

Target IP address	Value
-------------------	-------

Figure 8.11 *Payload format*

- Every time a node receives a packet it verifies whether the received payload's target IP address is the same as the current node IP address. If it is identical, the communication stops here, otherwise the data is forwarded to the next hop.
- Implement a single class for D1, D2 and D3 which is instantiated with different communication parameters (reuse the code and do not duplicate it for each instance)