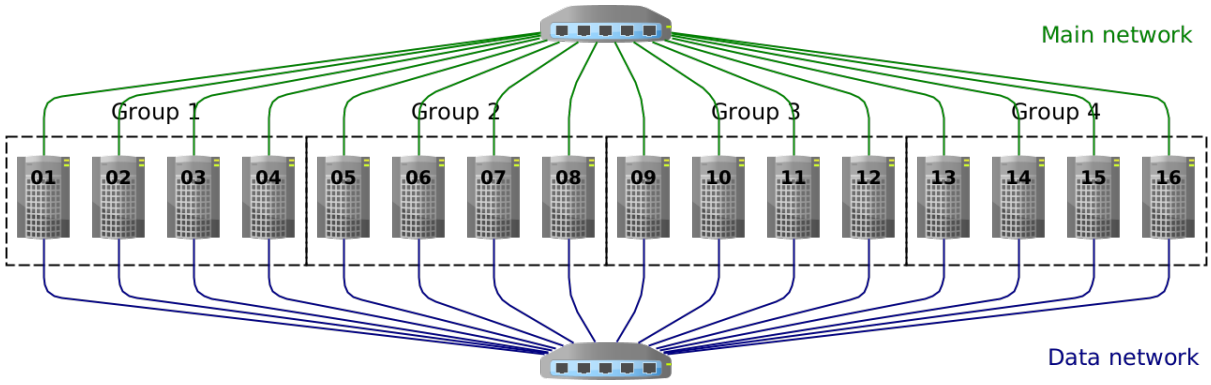
LHCb DAQ LAB

Data acquisition systems require many-to-one communications to collect data "fragments" from the various sensors of an experiment into a single destination where the fragments can be collated and analyzed. Many-to-one communications are challenging from the network point of view. In this lab you will experiment with some aspects of this communication pattern.

You will play with some computers from the LHCb data-acquisition system and you will learn about: network congestion, packet loss, and fair sharing of bandwidth. You will implement your own benchmarking tool, in Python, to learn the basics of network programming.

1. The network you will play on



You will share a cluster of 20 computers (called *tdeb01*, *tdeb02*, …, *tdeb20*) interconnected via two independent networks (called *main* and *data*). All the computers are connected to two network switches: one dedicated to the main network and one dedicated to the data network.

You will use:

- the *main* network for the traffic generated by your tests (hostnames: *tdebXX*),
- the *data* network to control the computers (hostnames: *tdebXX-d*),

Using two separate networks, you can be sure that the control traffic will not interfere with your tests and vice-versa.

2. Connect to your assigned computers

In order not to interfere with the other students, each of you will use a different sub-set of computers for your lab exercises:

- Group 1: tdeb01 to tdeb03
- Group 2: tdeb10 to tdeb12
- Group 3: tdeb13 to tdeb15
- Group 4: tdeb16 to tdeb18

First, you need to connect to the computers using SSH:

```
ssh tdebXX-d
```

Remember: always make sure that hostnames end in "-d" so that the SSH connection is made through the *data* network.

3. Skeleton of our TCP application

You will first build a TCP (Transmission Control Protocol) benchmark in Python. TCP is a connection-oriented protocol; this means it first needs to establish a connection before being able to send or receive data. TCP also manages packet loss (re-transmitting lost packets) and has congestion control to handle saturated links.

A typical network application is composed of a **client** and a **server.** For your tests, you will implement a server that acts as a data consumer and a client that acts as a data producer. In an experiment, the sensors/detectors can be seen as a collection of data producers, while the data-acquisition system can be seen as a collection of data consumers.

You will give the **client** or **server** role from the command line in Python. You can start from this skeleton (name it "tcp_flood"):

```
#!/usr/bin/python3

# For later use:
import socket
import struct
import sys
from datetime import datetime

# Main method in Python:
if __name__ == "__main__":
    mode = sys.argv[1]
    if mode == "client":
        # TODO: do client stuff
        pass
    elif mode == "server":
        # TODO: do server stuff
        pass
```

Do not forget to make it executable:

```
chmod a+x tcp_flood
```

4. Implementation of the client

The client is simple. You need to:

- Get the name or address of the server (given by argument: sys.argv[2]).
- Get the server port (given by argument: sys.argv[3]).

- Open a socket of type SOCK_STREAM (i.e TCP).
- Connect to the destination.
- In an infinite loop, send messages of a constant size (e.g.: 1024 bytes).

Some hints:

- You can find a client server example on https://wiki.python.org/moin/TcpCommunication
- To create a TCP socket:

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- To connect:

```
client_socket.connect((server, port))
```

- You can generate a message of a given size by using:

```
message = b" " * message_size
```

- To send a message:

```
client_socket.send(message)
```

5. Implementation of the server

The server will receive the connection and then loop to receive messages:

- Get the port to use (get it from sys.argv[2]).
- Create you socket of type SOCK_STREAM (see client).
- Associate the socket ("bind") to a local port.
- Listen for incoming connections.
- Start an infinite loop.
- Wait for a client and accept its incoming connection.
- Loop indefinitely to receive messages from the client. To preserve the boundaries between messages, you should receive messages of the same constant size used in the client.
- If you get a message of size 0, the client disconnected. Exit the inner loop to wait for another client.
- In the loop, count the cumulative size of received messages and print the bandwidth in Mbits/s every 10 000 loops.

Some hints:

- To bind the socket to a port for every available network interface:

```
listen_socket.bind(("0.0.0.0", port))
```

- Put the port in listen mode:

```
listen_socket.listen()
```

- Wait for a client and get the socket of incoming connection:

```
server_socket, client_address = listen_socket.accept()
```

- Receive a message:

```
message = server_socket.recv(message_size, socket.MSG_WAITALL)
```

- Get the length of the message:

```
len(message)
```

- Measure time:

```
start = datetime.now()
# do stuff
now = datetime.now()

# compute delta
seconds = (now - start).total_seconds()
```

- Print bandwidth every 100000 iterations:

```
if iterations % 100000 == 0:
    print ......
```

- Handle disconnection: in TCP in case of disconnection the recv() function will return a message of size 0. If you encounter such a message, you need to exit the inner loop (break) to wait another client.

6. Validation

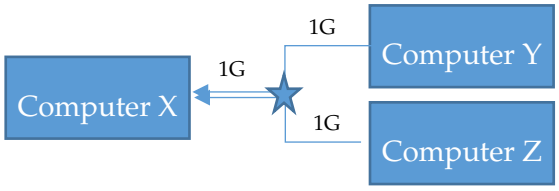
Run a first test locally by using two terminals to launch a client and server on the same computer:

```
# first terminal
./tcp_flood server 6000

# second
./tcp_flood client localhost 6000
```

7. Bandwidth sharing

We will now try to see how the network handles contention if two computers (Y, Z) try to send at the same time to computer X. So we will run 2 servers on X and a client on Y and Z.



Open four terminals and connect two of them to computer X, one to computer Y, and one to Z.

On computer X, launch two servers:

```
# on the first terminal (computer X)
./tcp_flood server 6000

# on the second terminal (computer X)
./tcp_flood server 6001
```

On Y, launch a first client:

```
# on the third terminal (computer Y)
./tcp_flood client X 6000
```

How much bandwidth do you get? Remember that the network in use is 1 Gbit/s.

Now launch the second client on computer Z to create congestion on the link to computer X:

```
# on the fourth terminal (computer Z)
./tcp_flood client X 6001
```

You can measure the bandwidth of your computers by using the command:

```
~/bwm-ng
```

How is the bandwidth distributed among the two clients?

What is the CPU usage of this benchmark? You can use the htop command.

The LHCb data-acquisition system uses a 200Gb/s network, what do you think about CPU usage at this speed?

8. Checking for packet loss

In order to check for potential packet loss in the network we need to identify the individual messages that we are sending. We can do this by writing an incrementing counter into each message. On the client side you can do this this by:

- Declaring a counter initialized to 1 before the infinite loop (you could call it message_num)
- Replacing the message generation by:

```
message = bytearray(b'\0' * MSG_SIZE)
```

- Writing the counter into the 8 first bytes of the message:

```
message[:8] = struct.pack('Q', message_num)
```

- Incrementing the counter into the loop

On the server side you need to check the packet stream integrity, you can do this by:

- Adding a counter to store the message counter from the last received message (you can call it last_num)
- Adding a counter for the missing messages (you can call it missing_messages)
- Extracting the message counter from the message itself using:

```
message_num = struct.unpack('Q', message[0:8])[0]
```

- Counting the missing messages:

```
missing_messages += message_num - last_num – 1
last_num = message_num
```

- Printing the number of lost messages
- You can also compare it to the expected number of received messages by adding a loop iteration counter in your loop and adding:

```
if iterations % 100000 == 0:
    missing_percentage = \
        100 * missing_messages / (iterations + missing_messages)
    iterations = 0
    missing_messages = 0
```

9. Moving to UDP

TCP is extremely popular because it hides a great deal of the complexities of the underlying network from the user. A simpler alternative is the simpler UDP protocol. Its simplicity comes

at a price: UDP is a stateless protocol, it has no message retransmission mechanism and no congestion control.

The advantage of a simple stateless protocol is that its implementation is easier. For example, it can be implemented directly in hardware using few resources. This was the case of the original LHCb data-acquisition system: the data sources were implemented with FPGAs (Field Programmable Gate Arrays). Implementing the full TCP protocol in a FPGA is an extremely challenging task due to the complexity of the protocol and management of the connection status, so UDP was used.

You will now update your tcp_flood program to make it use UDP and observe packet loss on the network in case of contention and unbalanced usage of the network in this scenario.

To proceed we will start from the TCP client, copy the file to another name (udp_flood).

```
cp tcp_flood udp_flood
```

10. UDP client

First we need to change to socket creation moving from SOCKET_STREAM (i.e TCP) to SOCKET_DGRAM (i.e UDP):

```
sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

Since UDP is stateless, there is no need to call socket.connect. You could use the socket.sendto function, providing every time the destination of your message. However, in Python this function accepts host names besides host addresses. When used with hostnames, the hostname is resolved to an address for every call to socket.sendto, which would slow down your benchmark. To side-step it, you can keep the call to socket.connect, but notice that for a UDP socket, this will not negotiate a connection with the server. It will simply store the destination you specified, so you don't have to give it every time you call socket.send.

11. UDP server

As for the server, first change the socket creation to SOCK_DGRAM (see client modification).

In addition, since UDP is a connectionless protocol, there is no need to listen for incoming connections. Rename listen_socket to server_socket, then remove the calls to socket.listen and socket.accept.

The last change concerns the reading; you need to replace the recv call with recvfrom:

```
message, client_address = sock.recvfrom(MSG_SIZE)
```

12. UDP bandwidth sharing

Redo the same measurement described in section 7 using udp_flood this time.

What do you see? What is your message loss percentage?

13. UDP vs TCP

Redo the same measurement described in section 7 with one tcp_flood client/server pair and one udp_flood pair this time.

What do you see? What are the bandwidths?

14. Going further

If you still have time, you can:

- Repeat the measurement with the iperf tool.
- Make the server capable of managing multiple clients at the same time using multi-threading or multi-processing.
- Run the tests on the data network instead of the main network. The data network is 10 times faster: you might need to optimize your benchmarks to reach the network limit.