U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Comunicações Industriais

Industrial Communications

2022/2023

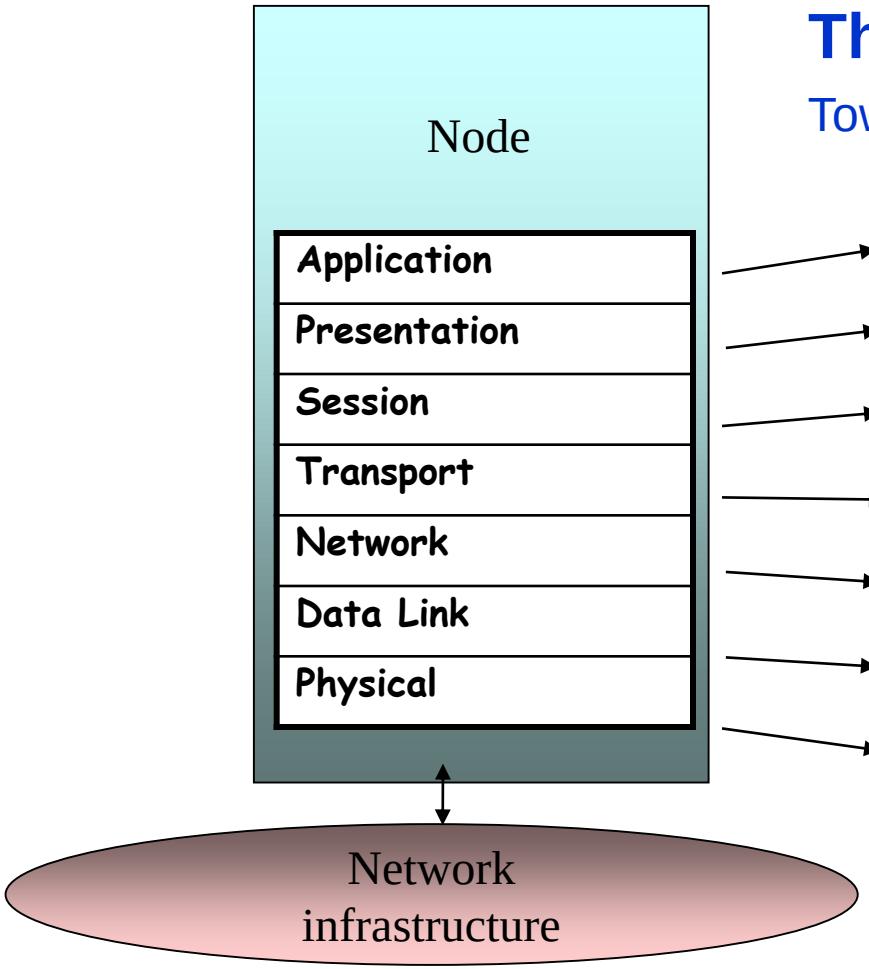
1st lab assingment

Implementing a Modbus TCP protocol stack

Learning objectives

- Understand the concepts of layer, service and protocol in industrial networks.
- Materialize those concepts by implementing a Modbus TCP protocol stack based on sockets, with an API to build Modbus clients and servers.
- Deepen the knowledge on Modbus.

Protocol stack

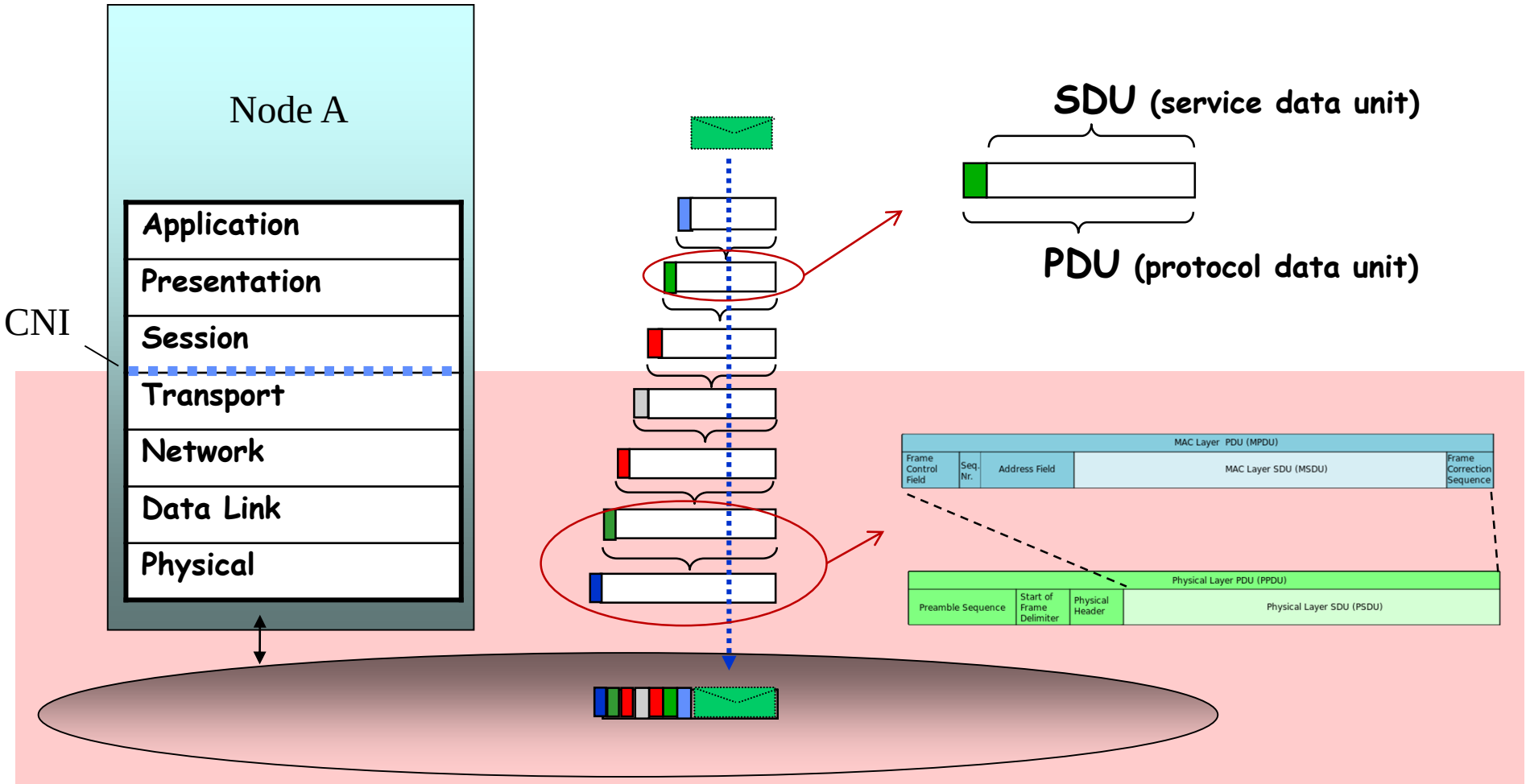


The OSI reference model

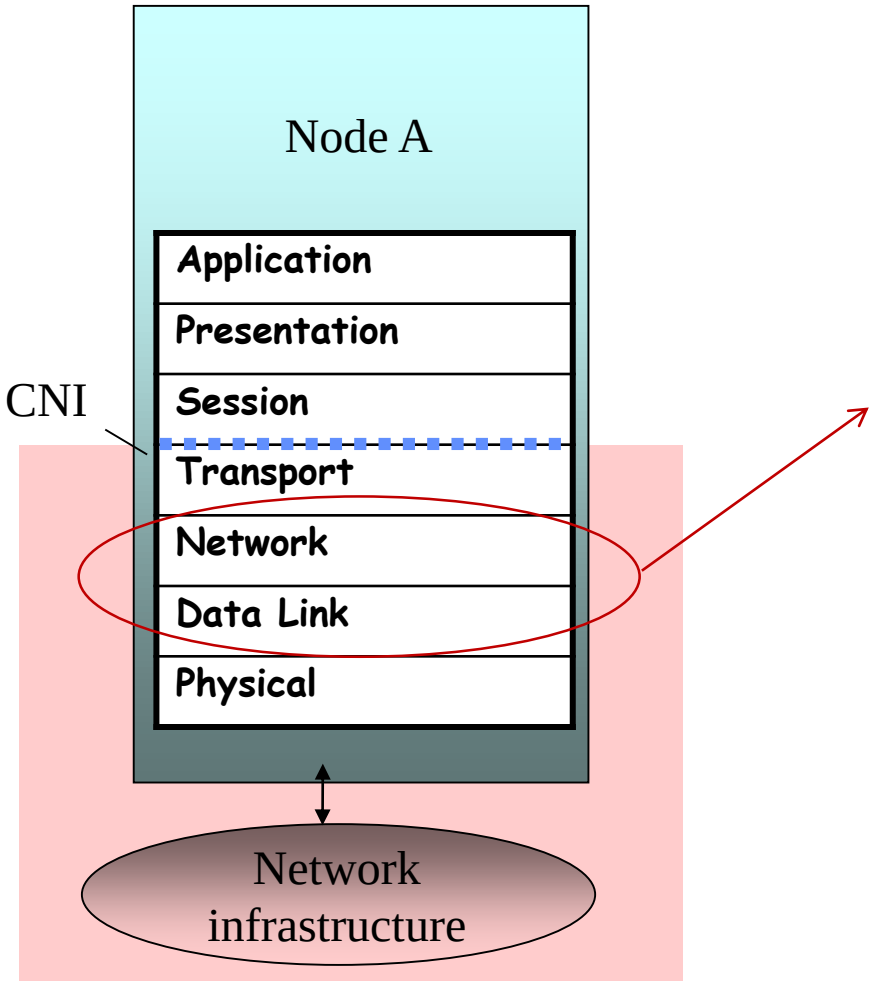
Towards interoperability and open systems

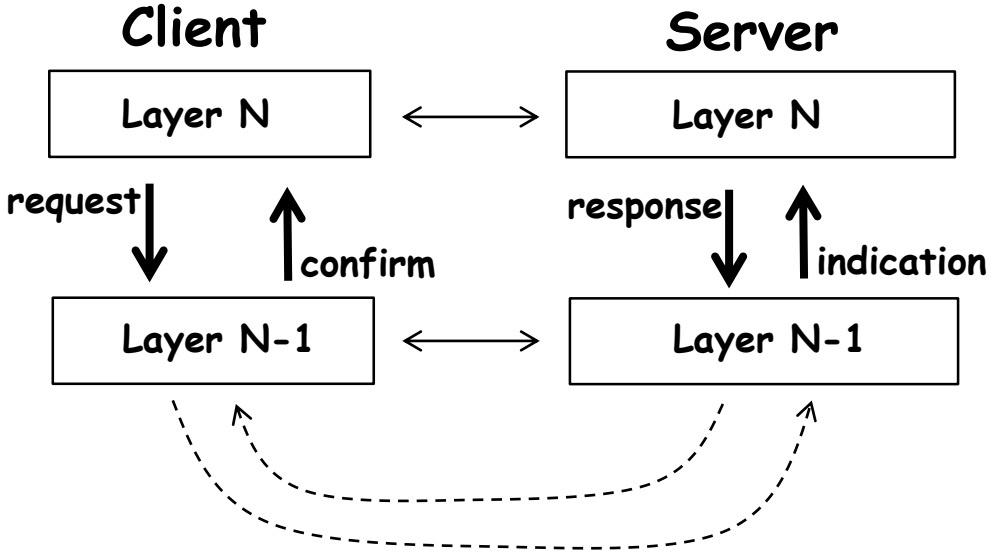
- Common services for class of apps.
- Data semantics
- Remote actions
- End-to-end comm. control
- Routing, logical addressing
- Physical addressing, MAC, LLC
- Topology, medium, bit-encoding

Protocol / Service Data Units



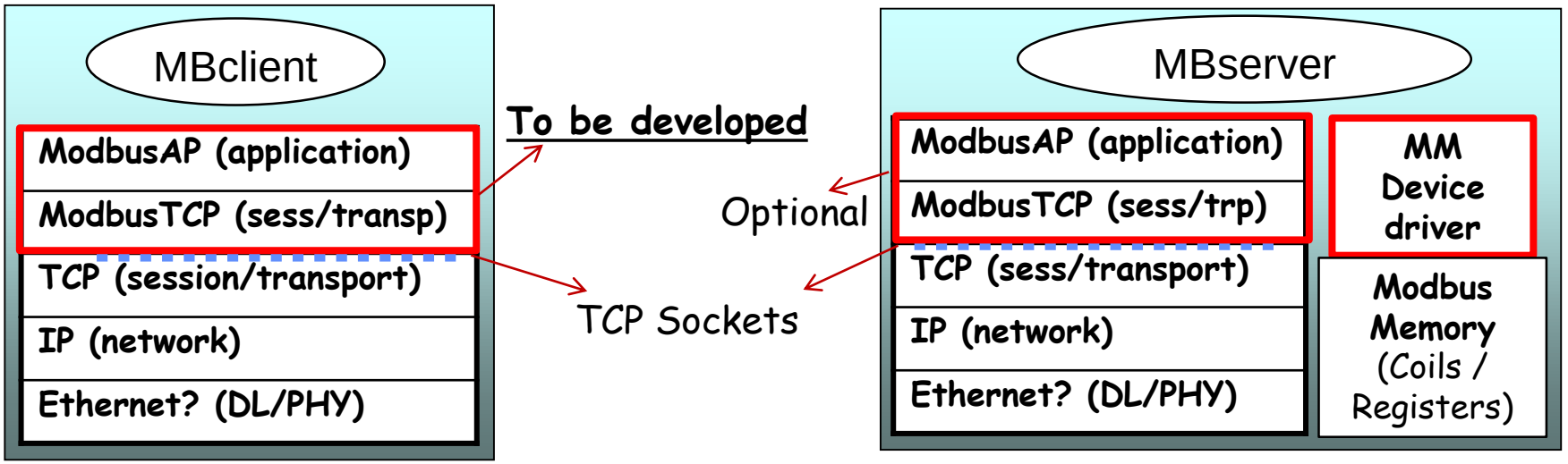
Interaction model between layers





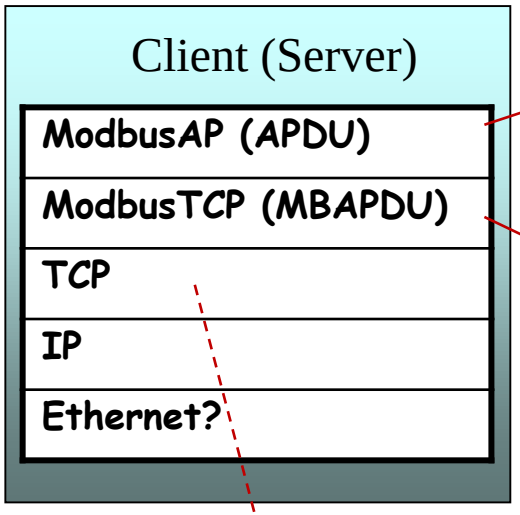
Implementing a Modbus TCP stack

- Modbus functions to be supported (application layer – client)
  - **Read holding registers**
  - **Write multiple registers**



Methods for Modbus TCP protocol stack

MM Dev. Driver (optional)
R_h_regs ( ) / W_regs ( )



ModbusAP.c
Read_h_regs ( ) / Write_multiple_regs ( )
(optional) **Get_Request ( ) / Send_Response ( )**

ModbusTCP.c
Send_Modbus_request ( )
(optional) **Receive_Modbus_request ( )**
(optional) **Send_Modbus_response ( )**

(*Sockets*)
Socket ( ), Bind ( ), Listen ( ) …
Read ( ), Write ( ), …

You may manage the TCP socket in a different way than suggested later on !

ModbusAP layer – client side

```
Write_multiple_regs (server_add, port, st_r, n_r, val)
{
   // check consistency of parameters
   // assembles APDU (MODBUS  PDU)
   Send_Modbus_request (server_add,port,APDU,APDUlen,APDU_R)
   // checks the reponse (APDU_R or error_code)
   // returns:  number of written registers – ok,  <0 – error
}
```

ModbusAP layer – client side

```
Read_h_regs (server_add, port, st_r, n_r, val)
{
   // check consistency of parameters
   // assembles APDU (MODBUS  PDU)
   Send_Modbus_request (server_add,port,APDU,APDUlen,APDU_R)
   // checks the reponse (APDU_R or error_code)
   // returns:  number of read registers – ok,  <0 – error
}
```

ModbusTCP layer – client side

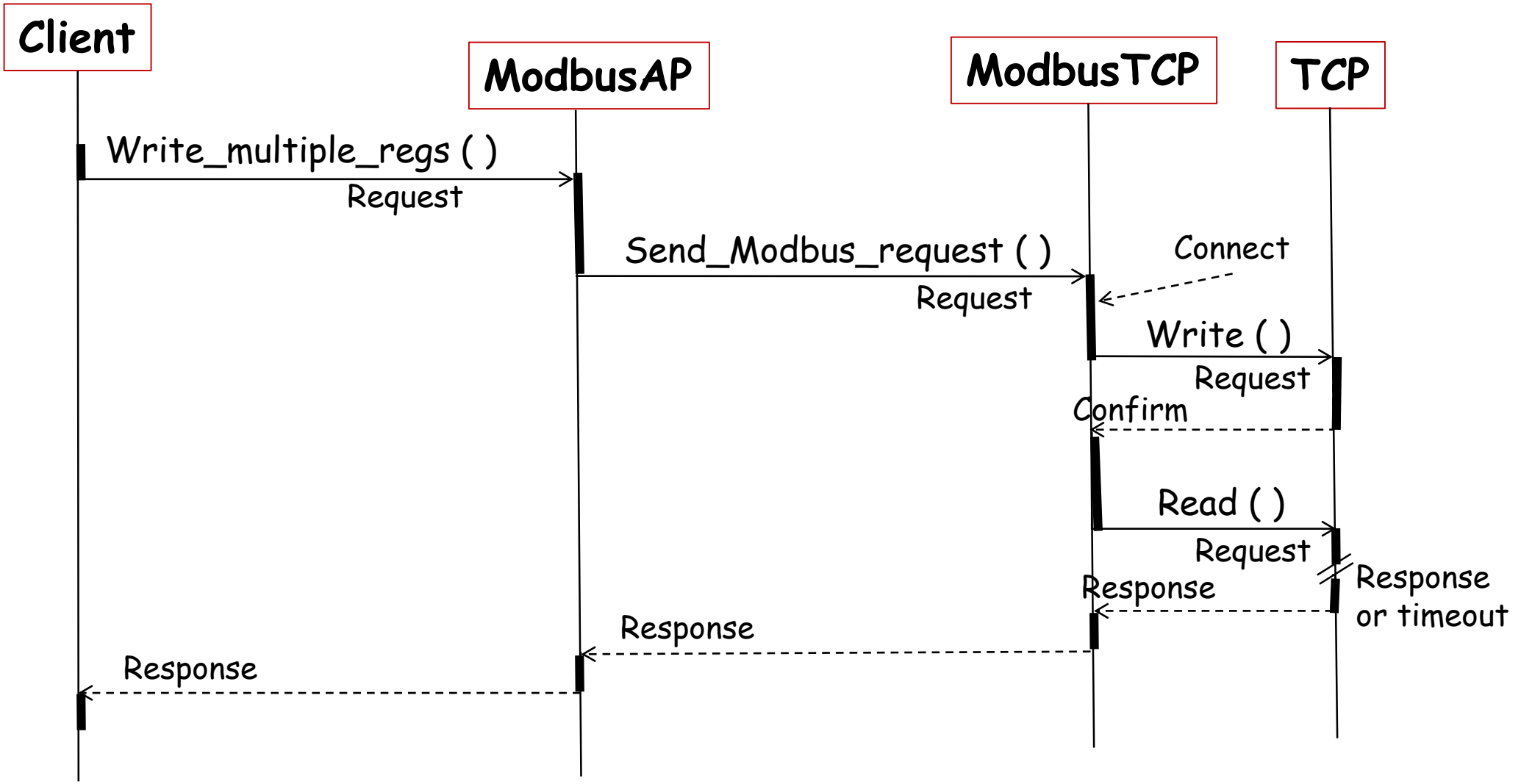
```
Send_Modbus_request (server_add, port, APDU,APDUlen, APDU_R)
{
        // generates TI (trans.ID →sequence number)
        // assembles PDU = APDU(SDU) + MBAP
        // opens TCP client socket and connects to server  (*)
        write (fd, PDU, PDUlen)     // sends Modbus TCP PDU
        read (fd, PDU_R, PDU_Rlen)  // response o timeout
        // if response, remove MBAP, PDU_R → APDU_R
        // closes TCP client socket with server  (*)
        // returns:  APDU_R and 0 – ok,  <0 – error (timeout)
}
```

fd = socket ( )
sockaddr_in…
connect (fd, s_addr, …)

close (fd)

(*) create and destroy a TCP socket every Modbus transaction is not really an efficient solution but it is simple and enough for now

Sequence diagram – client side



Session management – server side (optional)

```
fd = sConnect (server_add, port)
{
        // fd = socket ()
        // binds server address
        // listen – prepares for requests
        // returns  fd – ok,  <0 – error
}

sDisconnect (fd)
{
        // closes / destroys control socket
        // returns  >0 – ok,  <0 – error
}
```

Notes:

- Has to be modified to work with multiple connections

ModbusAP layer – server side (optional)

```
TI = Get_request (fd, op, st, n, val)
{
   TI = Receive_Modbus_request (fd, APDU, APDUlen)
   // extract parameters from APDU,
   // returns:  TI and parameters – ok, <0 – error
}

Send_response (TI, op, st, n, val)
{
   // prepare and send response APDU
   Send_Modbus_response (TI, APDU_R, APDU_Rlen)
   // returns:  >0 – ok, <0 – error
}
```

ModbusTCP layer – server side (optional)

```
TI = Receive_Modbus_request (fd, APDU,APDUlen)
{
   // waits for TCP connection, saves (global) data socket id
   read (fd2, PDU(MBAP), 7 )  // read MBAP of request PDU
   // remove MBAP: TI and length (APDUlen + 1)
   read (fd2, APDU, APDUlen)  // read request APDU
   // returns:  APDU and TI – ok, <0 – erro
}

Send_Modbus_response (TI, APDU_R, APDU_Rlen)
{
   // assembles PDU = APDU_R + MBAP (with TI)
   write (fd2, PDU, PDUlen)//sends ModbusTCP response PDU
   // returns:  >0 – ok, <0 – erro
}
```

fd2 = accept (fd, …)

IO Device Driver – server side (optional)

```
R_h_regs (st, n, val)
{
        // read n registers starting from st and write in val
        // returns:  num registers read, values in val – ok,  <0 – error
}

W_regs (st, n, val)
{
        // write n registers starting from st using values from val
        // returns:  num registers written – ok,    <0 – error
}
```

Sequence diagram – server side (optional)