

Mini Project on **Simulation of Modbus Master Slave communication over Virtual Machines**

Manasi Mujumdar

13th July 2020

—

Modbus Protocol Communication

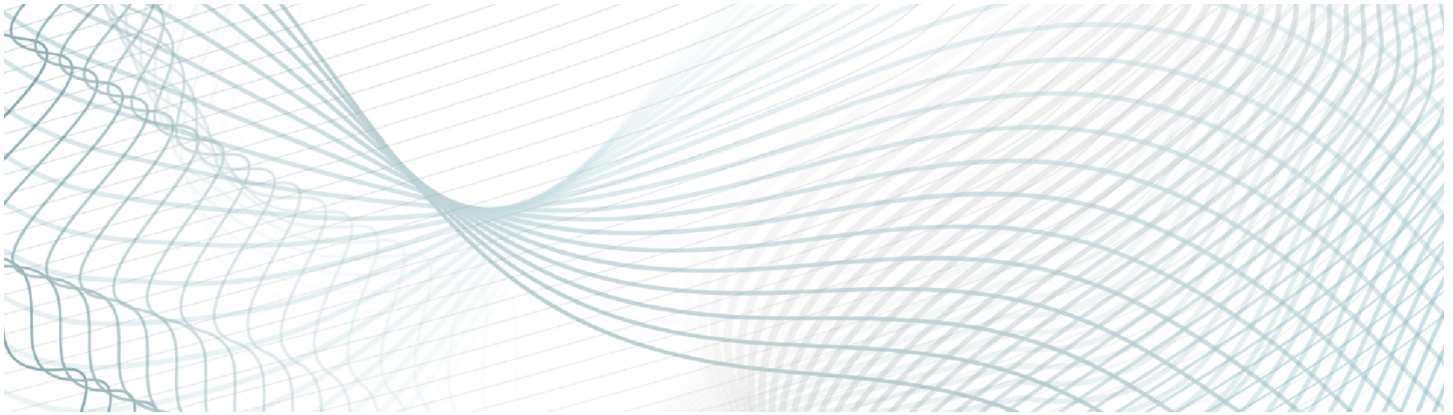
—

INDEX

1. Introduction
2. Technological Overview
3. About Modbus protocol
4. Testing of Implementation
5. Result

Introduction

This is a mini project to implement a modbus protocol between two virtual machines-one as server and other client and establish a communication between the two.



Technological Overview

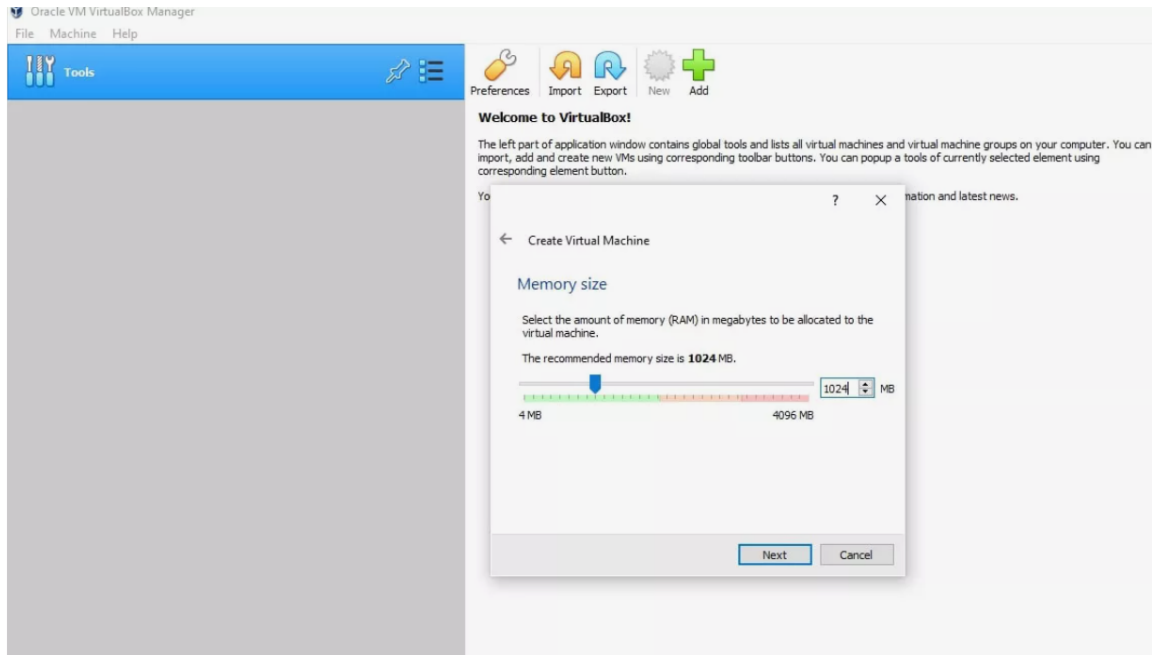
Virtual Box:

Oracle Virtual Box was used during the tenure of this project. It is a free and open-source hosted hypervisor for x86 virtualization, developed by Oracle Corporation.

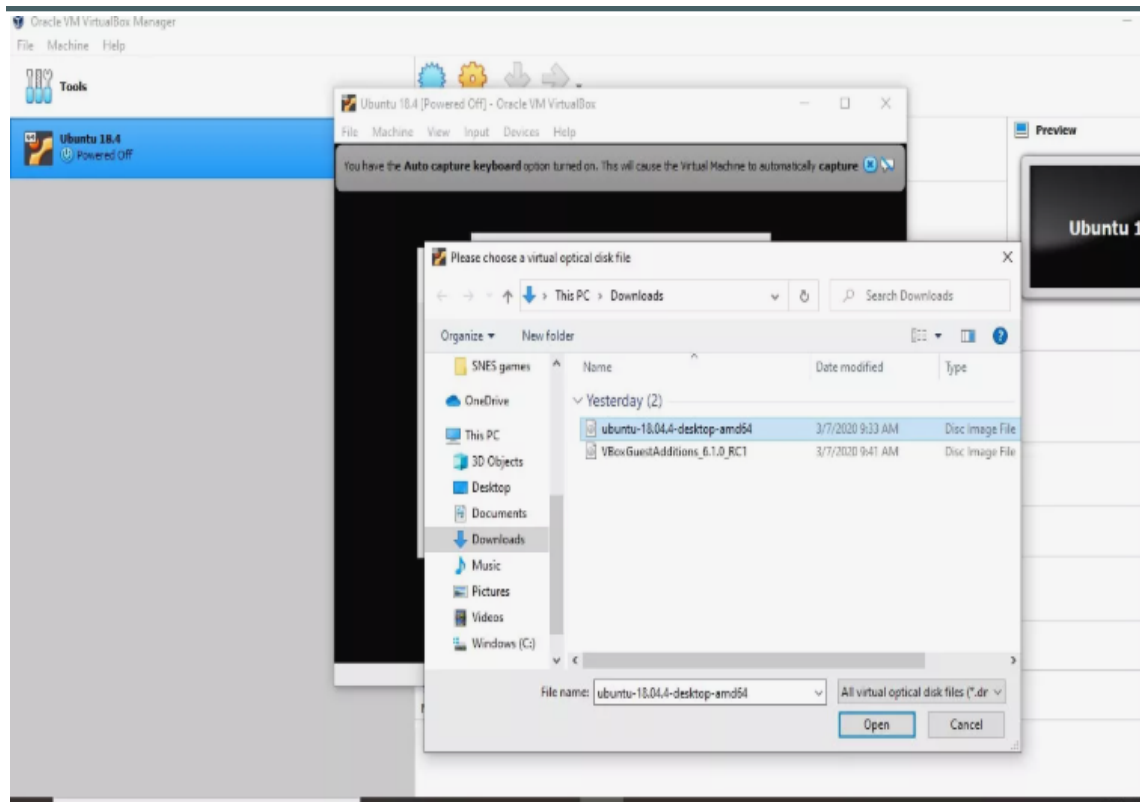
A virtual machine that runs on ubuntu 20.0 was created with a base memory of 2 GB.

Another similar machine was cloned with similar specifications for establishing communication between 2 VM's

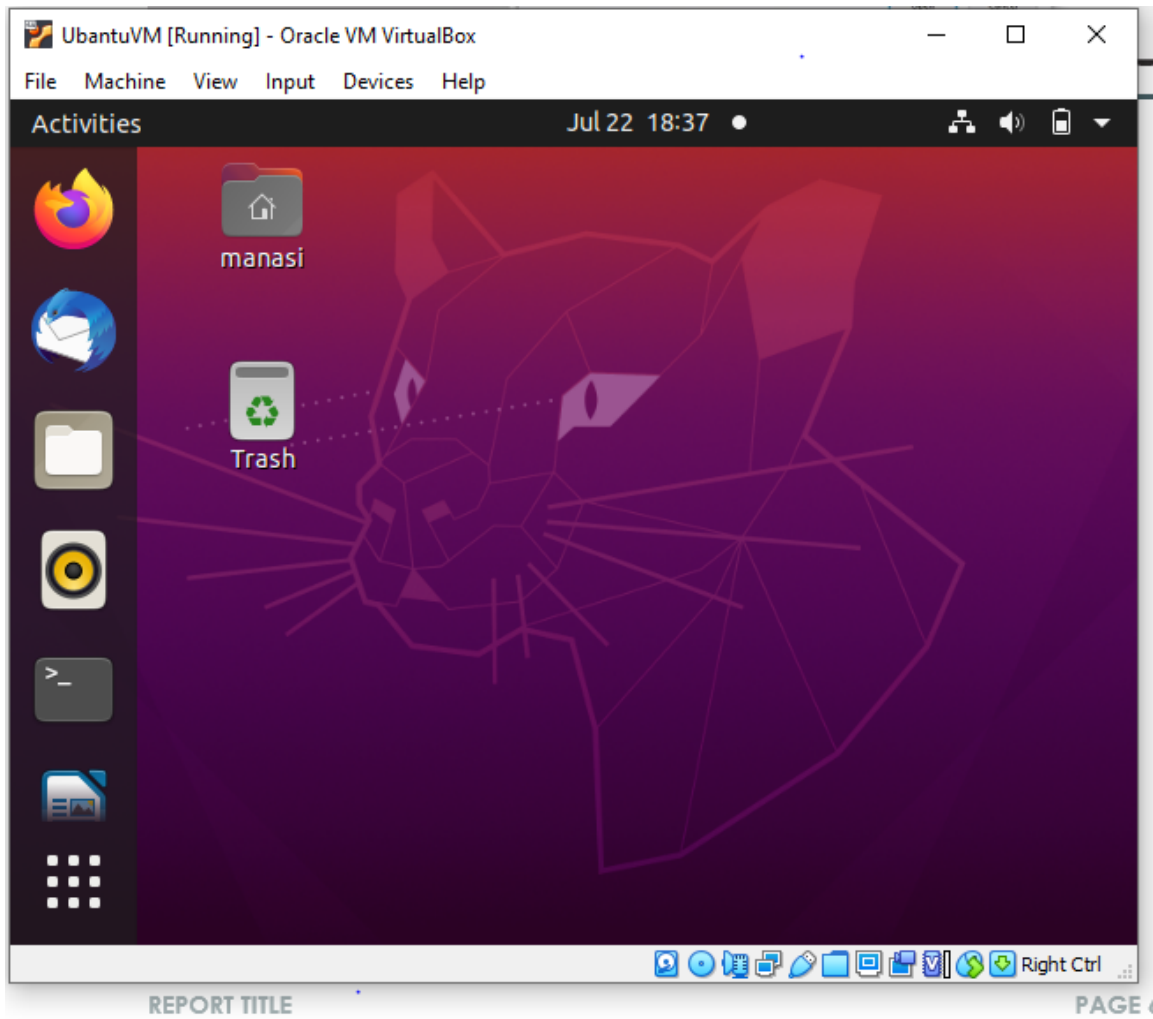
Here are few snippets



While installing ubuntu



After installing ubuntu :



About Modbus Protocol

MODBUS is an application-layer messaging protocol. It is used in industrial automation for communication between 2 PLC's ,transmitters ,digital meters and many such devices across varied networks.

☐ Why Modbus:

Modbus is an old yet popular protocol still relevant due to the flexibility it provides for different machines with different specifications to be on the same network and communicate with each other.

☐ Modbus Basic Structure:

- **Network type physical layer:**
 - ❖ **RS-485-** This is a full (5-wire) or half-duplex (3-wire) network. The network is multi-drop,. This communications method is common on industrial devices.

- ❖ **RS-232:** This method is still used when a single Modbus device needs to connect to an older piece of hardware that does not have a USB connection.
- ❖ **Ethernet:** This network type is standard Ethernet protocol; The advantage of this protocol is that the devices can be accessed from anywhere on the network, which is often installed throughout a facility

- **Modbus Registers and Tables:**

Defined for all Modbus enabled devices, Modbus tables are charts used to define the Modbus registers used in a device. Each piece of information accessible by Modbus in a device has a Modbus register

- **Master and Slave:**

The protocol defines the Master as the device that initiates the communication to a specific slave by sending it a request/ query

A Slave is that device on the network that responds the request sent by the Master that's specific to itself (by the Slave id)

- **Network Topology:**

Daisy chain type of topology is preferred due to its efficiency in controlling network traffic

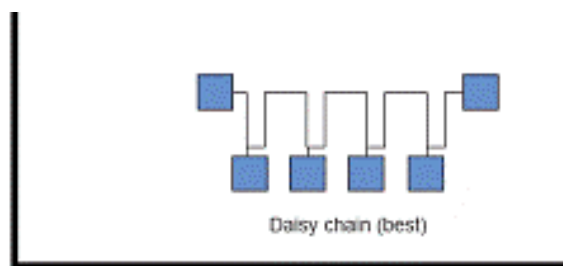


Figure 5—Many common network topologies exist for RS-485 networks

- **Data Types:**

Object Type	Access	Size	Address
Coils	Read-Write	1-bit	00001-09999
Discrete Input	Read-Only	1-bit	10001-19999
Input Register	Read-Only	16-bits	30001-39999
Holding Register	Read-Write	16-bits	40001-49999

An offset for each data address is used, since actual addresses are not used. Each table has a different offset value-1, 10001, 30001 and 40001.

□ Modbus is classified to:

1. Modbus RTU that uses serial communication
2. Modbus ASCII
3. Modbus TCP/IP (uses ethernet)

- Modbus RTU

- ❖ This type uses serial communication to communicate between devices. It is the process of sending data one bit at a time, sequentially, over a communication channel or computer bus.
- ❖ A remote terminal unit (RTU) is a microprocessor-controlled electronic device that interfaces objects in the physical world to a distributed control system or SCADA (supervisory control and data acquisition) system
- ❖ An RTU monitors the field digital and analog parameters and transmits data to a SCADA Master Station
- ❖ Each Modbus RTU message is terminated with two error checking bytes called a CRC or Cyclic Redundancy Check
- ❖ In Modbus RTU each byte is sent as a string of 8 binary characters framed with a start bit, and a stop bit, making each byte 10 bits.
- ❖ The range of data bytes in Modbus RTU can be any characters from 00 to FF.
- ❖ Example requesting data from registers 40108 to 40110 from slave address 17.

11 03 00 6B 00 03

Start	1	2	3	4	5	6	7	8	Par	Stop
-------	---	---	---	---	---	---	---	---	-----	------

- Modbus ASCII

- ❖ Modbus ASCII uses the ASCII character set to represent hexadecimal characters that each contain 4 bits of data.
- ❖ In Modbus ASCII, the number of data bits is reduced from 8 to 7
- ❖ In Modbus ASCII, each data byte is split into the two bytes representing the two ASCII characters in the Hexadecimal value.
- ❖ Modbus ASCII is terminated with an error checking byte called an LRC or Longitudinal Redundancy Check
- ❖ Example requesting data from registers 40108 to 40110 from slave address 17.

: 1 1 0 3 0 0 6 B 0 0 0 3 7 E CR LF

- Modbus TCP/IP

- ❖ TCP is Transmission Control Protocol and IP is Internet Protocol. These protocols are used together and are the transport protocol for the internet.
- ❖ When modbus information is sent using these protocols, the data is passed to TCP where additional information is attached and given to IP. IP then places the data in a packet (or datagram) and transmits it.
- ❖ A new 7-byte header called the MBAP header (Modbus Application Header) is added to the start of the message.

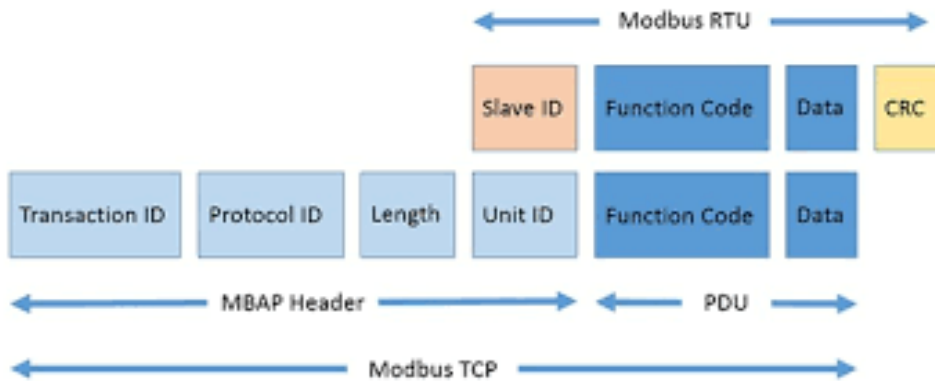
- o Protocol Identifier: 2 bytes set by the Client, always = 00 00
- o Length: 2 bytes identifying the number of bytes in the message to follow.
- o Unit Identifier: 1 byte set by the Client and echoed by the Server for identification of a remote slave connected on a serial line or on other buses.

- ❖ Example requesting data from registers : 0001 0000 0006 11 03 006B 0003

□ **Modbus-Serial Communication vs TCP/IP Communication:**

To compare, the difference between the 2 ideally lies in speed and accuracy. While most devices communicating through the Modbus protocol are devices like level and temperature sensors that do not require the data to be transmitted

fast(using Modbus TCP) but rather accurately which is done by Modbus RTU . Hence both have their own pros and cons that can be used as desired for the device using Modbus protocol.



Testing of Implementation

OBJECTIVE: To successfully read numerical data from coil, discrete input, holding registers, input registers from the Modbus server on request from the Modbus client

SPECIFIC LIBRARIES USED:

- **‘libmodbus’** - a free software library to send/receive data according to the Modbus protocol. Library cloned via its official GitHub repository This library is written in C and supports RTU (serial) and TCP (Ethernet) communications.
- **‘libxml’**- a software library for parsing XML documents.

PREREQUISITES:

- a client xml configuration file , consisting of necessary configuring parameters required for the client
Contents of the file:
 - Master IP address and Port
 - Count of slaves and its addresses
 - Addresses , count and scan period for all data types used(coil,discrete input, holding registers, input registers)
- Header file for server that contains pre-simulated data that would be requested by the client
Contents of header file:
 - Data contained in the four data types in form of ‘const arrays’
 - Count of number of bytes to be read
 - Address of first data type to be read

PROCEDURE:

CLIENT SIDE:

- Initialize and create arrays to hold the read values of type uint8_t for discrete input and coils and uint16_t for input registers and holding registers
- Parse the xml configuration document using xmlParseFile(). This would get the root element of the xml document.
- Set the tcp context of the client side, parse the ipaddress from the xml file using the user created retrieve_addr() function. It works similar to any function retrieving a node’s data from a tree’s structure.
- Parse the scanning period for discrete input and put the value in sleep()
- Retrieve the start address of discrete input from xml file and give it to the read_input_bits() from modbus library. It returns the number of bits read and we print the data
- Similarly for all other data types, we retrieve the scan period giving it to the sleep function and retrieve the starting address and give that to respective read function call of the data type . the data

is printed according to the number of bytes read in a for loop from the start address to the required address.

- The data read here are pre-defined numbers from the server's header file "unittest.h"
- Free the memory previously created for holding the discrete and analog values using free()
- The client then ends the connection with server using modbus_close()

SERVER SIDE:

- Firstly initialize pointers to modbus_t and modbus_mapping_t structures .
 - o The modbus_t structure is an opaque structure containing all necessary information to establish a connection with other Modbus devices according to the selected variant.
 - o The modbus_mapping_t structure has in store the pointer to the array that stores the address of input bits, coils, holding registers and input registers
- Initialize the query array where the modbus request would be stored
- Along with it initialize the fd_set (structure) that places sockets into sets according to whether they are available for reading. this is used in the select function when there are multiple connections associated with the server.
- Create a tcp connection with the client using modbus_new_tcp() by giving the ip address and port number
- Next step is to create modbus map of all the slave devices. In this program the function modbus_mapping_new_address() return a pointer to structure modbus_mapping_t. the function takes in the start address of each data type along with the total no of bytes for each that is mentioned in unittest.h
- To simulate sample data in the server the following steps are performed:
 - o For the discrete data types(coil,discrete input bits) modbus_set_bits_from_bytes() is used. It set bits in the mapping array for discrete input type by reading an array of bytes, in this scenario from the array of bytes present the unittest.h file.
 - o For analog data types(registers): a for loop sequentially transfers the data to the respective mapping array of register from the array of data for registers in the header file unittest.h
- Server then listens for any incoming connection from client via modbus_tcp_listen() and returns a new socket called server_socket .

- Declare a signal for closing of server socket if the server socket is not validly created while the server is listening for new connections
- In the next steps clear the reference set of sockets `rdset(structure)` and add the server socket to `fd_set`. Along with it the program also keeps a track of the max file descriptor in variable `fdmax`.
- In an infinite for loop the `select()` is called giving the requires parameters of `fdmax` and `rdset`. It will now be used to multiplex the incoming requests via a single process.
- Another for loop starting from '0' till `fdmax` is created to run through existing connections for data to be read. If the particular `fd` is not present in the `fd_set` then the loop 'continues'.
- Once the required iterated socket matches the server socket value then that would signify a new connection. The client address structure is declared, length of the same is calculated.
- The connection is then accepted using `accept()` and a 'newfd' file descriptor is returned .we add this in our `fd_set` and update our `maxfd` if its lesser than `newfd`.
- Else if the required iterated socket doesn't match the server socket then the `modbus_set_socket()` set the socket or file descriptor in the libmodbus context. This is useful for managing multiple client connections to the same server.
- The `modbus_recieve()` then receives an indication request from the socket of the context *ctx*. The slave/server receives and analyzes the indication request sent by the masters/clients. The function store the indication request in query array and returns the request length
- If the request length is greater than zero then the `modbus_reply()` shall send a response to received request using the information of the modbus context *ctx*. If the request indicates to read or write a value the operation will be done in the modbus mapping *mb_mapping* according to the type of the manipulated data. It returns the length of the response sent if successful to the client.

- If the above mentioned procedure goes successfully then we remove that particular socket from the reference socket fd_set and return 0.

Reading values from discrete data types(coil and discrete input bits)

The image displays two screenshots of a Linux terminal window, likely Ubuntu, showing the execution of a Modbus client and server program.

Left Screenshot: The terminal shows the user `manasi@manasi-VirtualBox` in the directory `~/Documents/ModbusProjectsbyBaba`. The user runs `cd /home/manasi/Documents/ModbusProjectsbyBaba` and then `cc -g client2.c -o client -L/usr/include -lmodbus -lm2`. The program `client2.c` is compiled into `client`. The user then runs `./client`, which connects to `192.168.2.2:1502`. The terminal displays a series of hexadecimal data packets and a confirmation message. The output shows input bits for a Modbus read request, with values ranging from `0x0` to `0xA`.

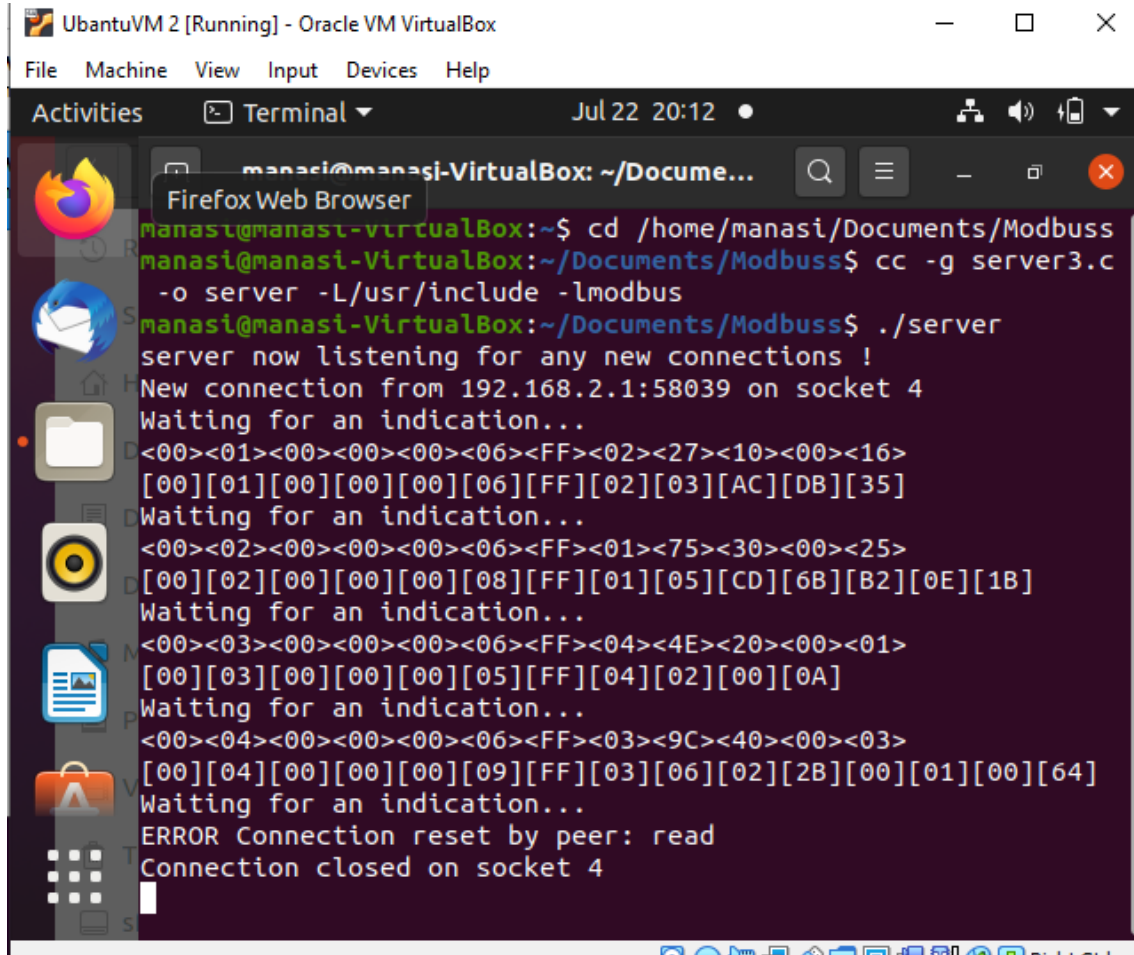
Right Screenshot: The terminal shows the user `manasi@manasi-VirtualBox` in the directory `~/Documents/Modbus`. The user runs `cd /home/manasi/Documents/Modbus` and then `cc -g server3.c -o server -L/usr/include -lmodbus`. The program `server3.c` is compiled into `server`. The user then runs `./server`, which starts listening for new connections. A new connection is established from `192.168.2.1:58039` on socket 4. The terminal displays a series of hexadecimal data packets and a confirmation message. The output shows discrete output bits for a Modbus write request, with values ranging from `0x0` to `0xA`.

Smaller window: SERVER bigger window: CLIENT

Reading values from analog data types(holding registers and analog input registers)

```
[00][03][00][00][00][06][FF][04][4E][20][00][01]
Waiting for a confirmation...
<00><03><00><00><00><05><FF><04><02><00><0A>
1/1 modbus_read_input_registers: values are -----
reg[0]=10 (0xA)
OK
[00][04][00][00][00][06][FF][03][9C][40][00][03]
Waiting for a confirmation...
<00><04><00><00><00><09><FF><03><06><02><2B><00><01><00><64>
2/5 modbus_read_registers: values are -----
reg[0]=555 (0x22B)
reg[1]=1 (0x1)
reg[2]=100 (0x64)
OK
ALL TESTS PASS WITH SUCCESS.
manasi@manasi-VirtualBox:~/Documents/ModbusProjectsbyBaba$
```

CLIENT



UbuntuVM 2 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Activities Terminal Jul 22 20:12

Firefox Web Browser

```
manasi@manasi-VirtualBox:~/Documents/Modbus$ cd /home/manasi/Documents/Modbus
manasi@manasi-VirtualBox:~/Documents/Modbus$ cc -g server3.c
-o server -L/usr/include -lmodbus
manasi@manasi-VirtualBox:~/Documents/Modbus$ ./server
server now listening for any new connections !
New connection from 192.168.2.1:58039 on socket 4
Waiting for an indication...
<00><01><00><00><00><06><FF><02><27><10><00><16>
[00][01][00][00][00][06][FF][02][03][AC][DB][35]
Waiting for an indication...
<00><02><00><00><00><06><FF><01><75><30><00><25>
[00][02][00][00][00][08][FF][01][05][CD][6B][B2][0E][1B]
Waiting for an indication...
<00><03><00><00><00><06><FF><04><4E><20><00><01>
[00][03][00][00][00][05][FF][04][02][00][0A]
Waiting for an indication...
<00><04><00><00><00><06><FF><03><9C><40><00><03>
[00][04][00][00][00][09][FF][03][06][02][2B][00][01][00][64]
Waiting for an indication...
ERROR Connection reset by peer: read
Connection closed on socket 4
```

SERVER