# MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b

## CONTENTS

# 1 INTRODUCTION

## 1.1 OBJECTIVES

The objective of this document is to present the MODBUS messaging service over TCP/IP , in order to provide reference information that helps software developers to implement this service. The encoding of the MODBUS function codes is not described in this document, for this information please read the MODBUS Application Protocol Specification [1].

This document gives accurate and comprehensive description of a MODBUS messaging service implementation. Its purpose is to facilitate the interoperability between the devices using the MODBUS messaging service.

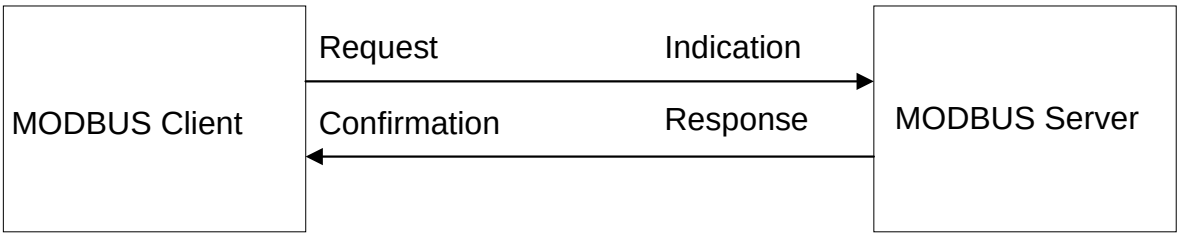
This document comprises mainly three parts:

- An overview of the MODBUS over TCP/IP protocol
- A functional description of a MODBUS client, server and gateway implementation.
- An implementation guideline that proposes the object model of an MODBUS implementation example.

## 1.2 CLIENT / SERVER MODEL

The MODBUS messaging service provides a Client/Server communication between devices connected on an Ethernet TCP/IP network.
This client / server model is based on four type of messages:

- **MODBUS Request,**
- **MODBUS Confirmation,**
- **MODBUS Indication,**
- **MODBUS Response**




**A MODBUS Request** is the message sent on the network by the Client to initiate a transaction,

**A MODBUS Indication** is the Request message received on the Server side,

**A MODBUS Response** is the Response message sent by the Server,

**A MODBUS Confirmation** is the Response Message received on the Client side

The MODBUS messaging services (Client / Server Model) are used for real time information exchange:

- between two device applications,
- between device application and other device,
- between HMI/SCADA applications and devices,
- between a PC and a device program providing on line services.

-

### 1.3 REFERENCE DOCUMENTS

This section gives a list of documents that are interesting to read before this one:

[1] MODBUS Application Protocol Specification V1.1a.
[2] RFC 1122 Requirements for Internet Hosts -- Communication Layers

## 2 ABBREVIATIONS

| | |
|---|---|
| **ADU** | Application Data Unit |
| **IETF** | Internet Engineering Task Force |
| **IP** | Internet Protocol |
| **MAC** | Medium Access Control |
| **MB** | MODBUS |
| **MBAP** | MODBUS Application Protocol |
| **PDU** | Protocol Data Unit |
| **PLC** | Programmable Logic Controller |
| **TCP** | Transport Control Protocol |
| **BSD** | Berkeley Software Distribution |
| **MSL** | Maximum Segment Lifetime |

## 3 CONTEXT

### 3.1 PROTOCOL DESCRIPTION

#### 3.1.1 General communication architecture

A communicating system over MODBUS TCP/IP may include different types of device:

- A MODBUS TCP/IP Client and Server devices connected to a TCP/IP network
- The Interconnection devices like bridge, router or gateway for interconnection between the TCP/IP network and a serial line sub-network which permit connections of MODBUS Serial line Client and Server end devices.

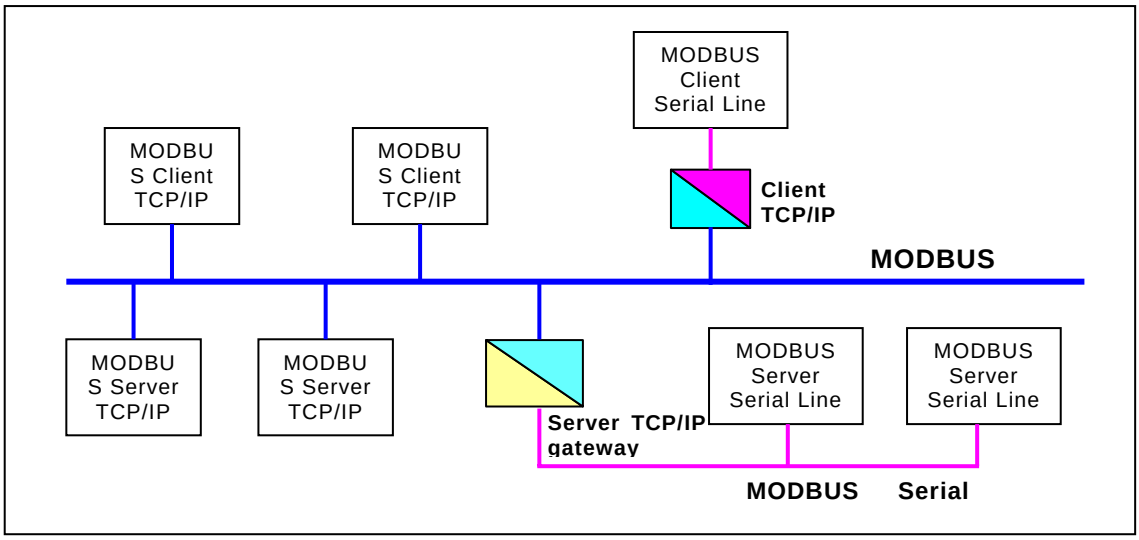



**Figure 1: MODBUS TCP/IP communication architecture**

The MODBUS protocol defines a **simple Protocol Data Unit (PDU)** independent of the underlying communication layers. The mapping of MODBUS protocol on specific buses or networks can introduce some additional fields on the **Application Data Unit (ADU)**.

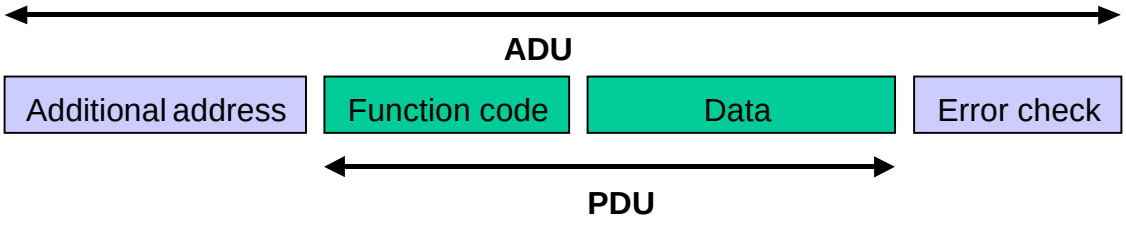



**Figure 2: General MODBUS frame**

The client that initiates a MODBUS transaction builds the MODBUS Application Data Unit. The function code indicates to the server which kind of action to perform.

### 3.1.2 MODBUS On TCP/IP Application Data Unit

This section describes the encapsulation of a MODBUS request or response when it is carried on a MODBUS TCP/IP network.

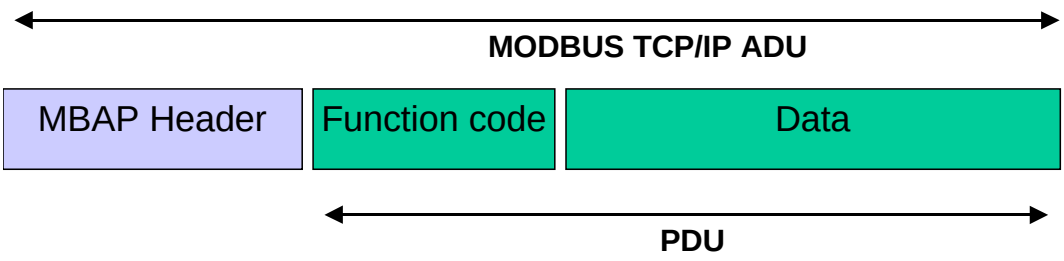



**Figure 3: MODBUS request/response over TCP/IP**

A dedicated header is used on TCP/IP to identify the MODBUS Application Data Unit. It is called the MBAP header (MODBUS Application Protocol header).

This header provides some differences compared to the MODBUS RTU application data unit used on serial line:

- The MODBUS 'slave address' field usually used on MODBUS Serial Line is replaced by a single byte 'Unit Identifier' within the MBAP Header. The 'Unit Identifier' is used to communicate via devices such as bridges, routers and gateways that use a single IP address to support multiple independent MODBUS end units.
- All MODBUS requests and responses are designed in such a way that the recipient can verify that a message is finished. For function codes where the MODBUS PDU has a fixed length, the function code alone is sufficient. For function codes carrying a variable amount of data in the request or response, the data field includes a byte count.
- When MODBUS is carried over TCP, additional length information is carried in the MBAP header to allow the recipient to recognize message boundaries even if the message has been split into multiple packets for transmission. The existence of explicit and implicit length rules, and use of a CRC-32 error check code (on Ethernet) results in an infinitesimal chance of undetected corruption to a request or response message.

### 3.1.3 MBAP Header description

The MBAP Header contains the following fields:

| Fields | Length | Description - | Client | Server |
|---|---|---|---|---|
| Transaction Identifier | 2 Bytes | Identification of a MODBUS Request / Response transaction. | Initialized by the client | Recopied by the server from the received request |
| Protocol Identifier | 2 Bytes | 0 = MODBUS protocol | Initialized by the client | Recopied by the server from the received request |
| Length | 2 Bytes | Number of following bytes | Initialized by the client ( request) | Initialized by the server ( Response) |
| Unit Identifier | 1 Byte | Identification of a remote slave connected on a serial line or on other buses. | Initialized by the client | Recopied by the server from the received request |

The header is 7 bytes long:

- **Transaction Identifier** - It is used for transaction pairing, the MODBUS server copies in the response the transaction identifier of the request.
- **Protocol Identifier** – It is used for intra-system multiplexing. The MODBUS protocol is identified by the value 0.
- **Length** - The length field is a byte count of the following fields, including the Unit Identifier and data fields.

- **Unit Identifier** – This field is used for intra-system routing purpose. It is typically used to communicate to a MODBUS+ or a MODBUS serial line slave through a gateway between an Ethernet TCP-IP network and a MODBUS serial line. This field is set by the MODBUS Client in the request and must be returned with the same value in the response by the server.

**All MODBUS/TCP ADU are sent via TCP to registered port 502.**

*Remark : the different fields are encoded in Big-endian.*

### 3.2 MODBUS FUNCTIONS CODES DESCRIPTION

Standard function codes used on MODBUS application layer protocol are described in details in the MODBUS Application Protocol Specification [1].

# 4 FUNCTIONAL DESCRIPTION

The MODBUS Component Architecture presented here is a general model including both MODBUS Client and Server Components and usable on any device.

Some devices may only provide the server or the client component.

In the first part of this section a brief overview of the MODBUS messaging service component architecture is given, followed by a description of each component presented in the architectural model.

## 4.1 MODBUS COMPONENT ARCHITECTURE MODEL

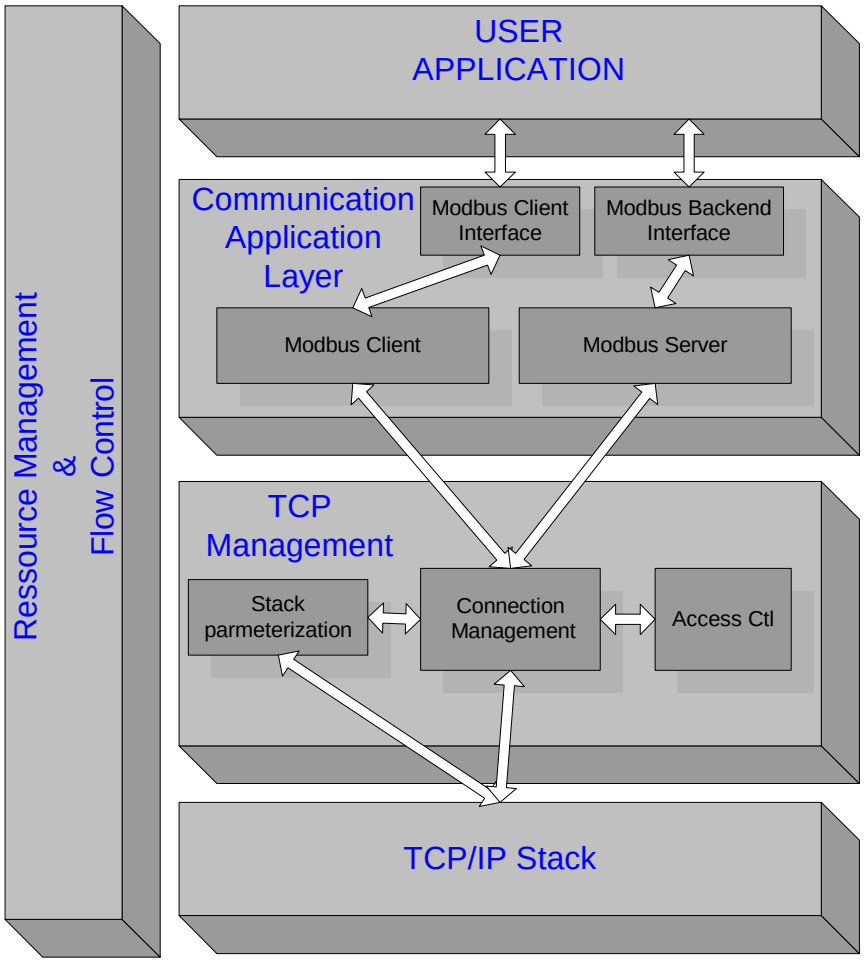



**Figure 4: MODBUS Messaging Service Conceptual Architecture**

- **Communication Application Layer**

A MODBUS device may provide a client and/or a server MODBUS interface.

A MODBUS backend interface can be provided allowing indirectly the access to user application objects.
Four areas can compose this interface: input discrete, output discrete (coils), input registers and output registers. A pre-mapping between this interface and the user application data has to be done (local issue).

| Primary tables | Object type | Type of | Comments |
|---|---|---|---|
| Discretes Input | Single bit | Read-Only | This type of data can be provided by an I/O system. |
| Coils | Single bit | Read-Write | This type of data can be alterable by an application program. |
| Input Registers | 16-bit word | Read-Only | This type of data can be provided by an I/O system |
| Holding Registers | 16-bit word | Read-Write | This type of data can be alterable by an application program. |

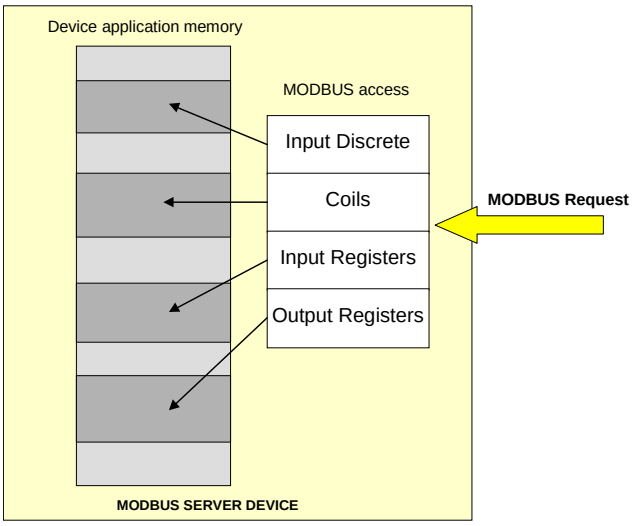



**Figure 5 MODBUS Data Model with separate blocks**

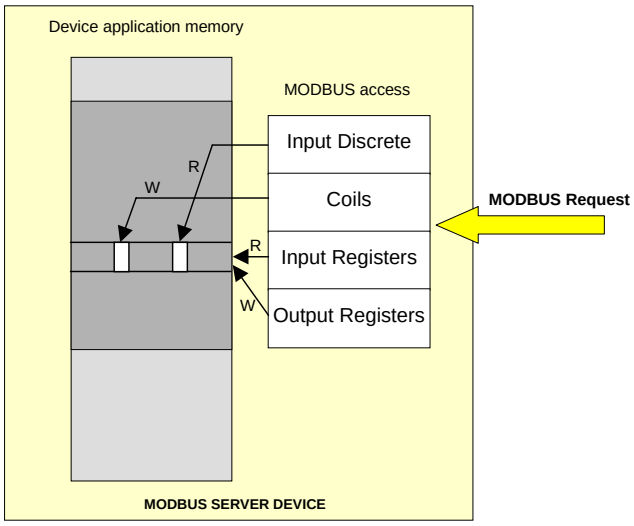



**Figure 6 MODBUS Data Model with only 1 block**

- MODBUS Client

The MODBUS Client allows the user application to explicitly control information exchange with a remote device. The MODBUS Client builds a MODBUS request from parameter contained in a demand sent by the user application to the MODBUS Client Interface.
The MODBUS Client uses a MODBUS transaction whose management includes waiting for and processing of a MODBUS confirmation.

- MODBUS Client Interface

The MODBUS Client Interface provides an interface enabling the user application to build the requests for various MODBUS services including access to MODBUS application objects. The MODBUS Client interface (API) is not part of this Specification, although an example is described in the implementation model.

- MODBUS Server

On reception of a MODBUS request this module activates a local action to read, to write or to achieve some other actions. The processing of these actions is done totally transparently for the application programmer. The main MODBUS server functions are to wait for a MODBUS request on 502 TCP port, to treat this request and then to build a MODBUS response depending on device context.

- MODBUS Backend Interface

The MODBUS Backend Interface is an interface from the MODBUS Server to the user application in which the application objects are defined.

Informative Note: The Backend Interface is not defined in this Specification

- **TCP Management layer**

Informative Note: The TCP/IP discussion in this Specification is based in part upon reference [2] RFC 1122 to assist the user in implementing the MODBUS Application Protocol Specification [1] over TCP/IP.

One of the main functions of the messaging service is to manage communication establishment and ending and to manage the data flow on established TCP connections.

- Connection Management

A communication between a client and server MODBUS Module requires the use of a TCP connection management module. It is in charge to manage globally messaging TCP connections.

Two possibilities are proposed for the connection management. Either the user application itself manages TCP connections or the connection management is totally done by this module and therefore it is transparent for the user application. The last solution implies less flexibility.

The **listening TCP port 502 is reserved for MODBUS** communications. It is mandatory to listen by default on that port. However, some markets or applications might require that another port is dedicated to MODBUS over TCP. For that reason, it is highly recommended that the clients and the servers give the possibility to the user to parameterize the MODBUS over TCP port number. **It is important to note that even if another TCP server port is configured for MODBUS service in certain applications, TCP server port 502 must still be available in addition to any application specific ports.**

- Access Control Module

In certain critical contexts, accessibility to internal data of devices must be forbidden for undesirable hosts. That's why a security mode is needed and security process may be implemented if required.

- **TCP/IP Stack layer**

The TCP/IP stack can be parameterized in order to adapt the data flow control, the address management and the connection management to different constraints specific to a product or to a system. Generally the BSD socket interface is used to manage the TCP connections.

- **Resource management and Data flow control**

In order to equilibrate inbound and outbound messaging data flow between the MODBUS client and the server, data flow control mechanism is provided in all layers of MODBUS messaging stack.
The resource management and flow control module is first based on TCP internal flow control added with some data flow control in the data link layer and also in the user application level.

## 4.2 TCP CONNECTION MANAGEMENT

### 4.2.1 Connections management Module

#### 4.2.1.1 General description

A MODBUS communication requires the establishment of a TCP connection between a Client and a Server.
The establishment of the connection can be activated either explicitly by the User Application module or automatically by the TCP connection management module.
In the first case an application-programming interface has to be provided in the user application module to manage completely the connection. This solution provides flexibility for the application programmer but it requires a good expertise on TCP/IP mechanism.
In the second case the TCP connection management is completely hidden to the user application that only sends and receives MODBUS messages. The TCP connection management module is in charge to establish a new TCP connection when it is required.
The definition of the number of TCP client and server connections is not on the scope of this document (value n in this document). Depending on the device capacities the number of TCP connections can be different.

**Implementation Rules :**

1) Without explicit user requirement, it is recommended to implement the automatic TCP connection management

2) It is recommended to keep the TCP connection opened with a remote device and not to open and close it for each MODBUS/TCP transaction,
*Remark: However the MODBUS client must be capable of accepting a close request from the server and closing the connection. The connection can be reopened when required.*

3) It is recommended for a MODBUS Client to open a minimum of TCP connections with a remote MODBUS server (with the same IP address). One connection per application could be a good choice.

4) Several MODBUS transactions can be activated simultaneously on the same TCP Connection.
*Remark: If this is done then the MODBUS transaction identifier must be used to uniquely identify the matching requests and responses.*

5) In case of a bi-directional communication between two remote MODBUS entities ( each of them is client and server), it is necessary to open separate connections for the client data flow and for the server data flow.

6) A TCP frame must transport only one MODBUS ADU. It is advised against sending multiple MODBUS requests or responses on the same TCP PDU

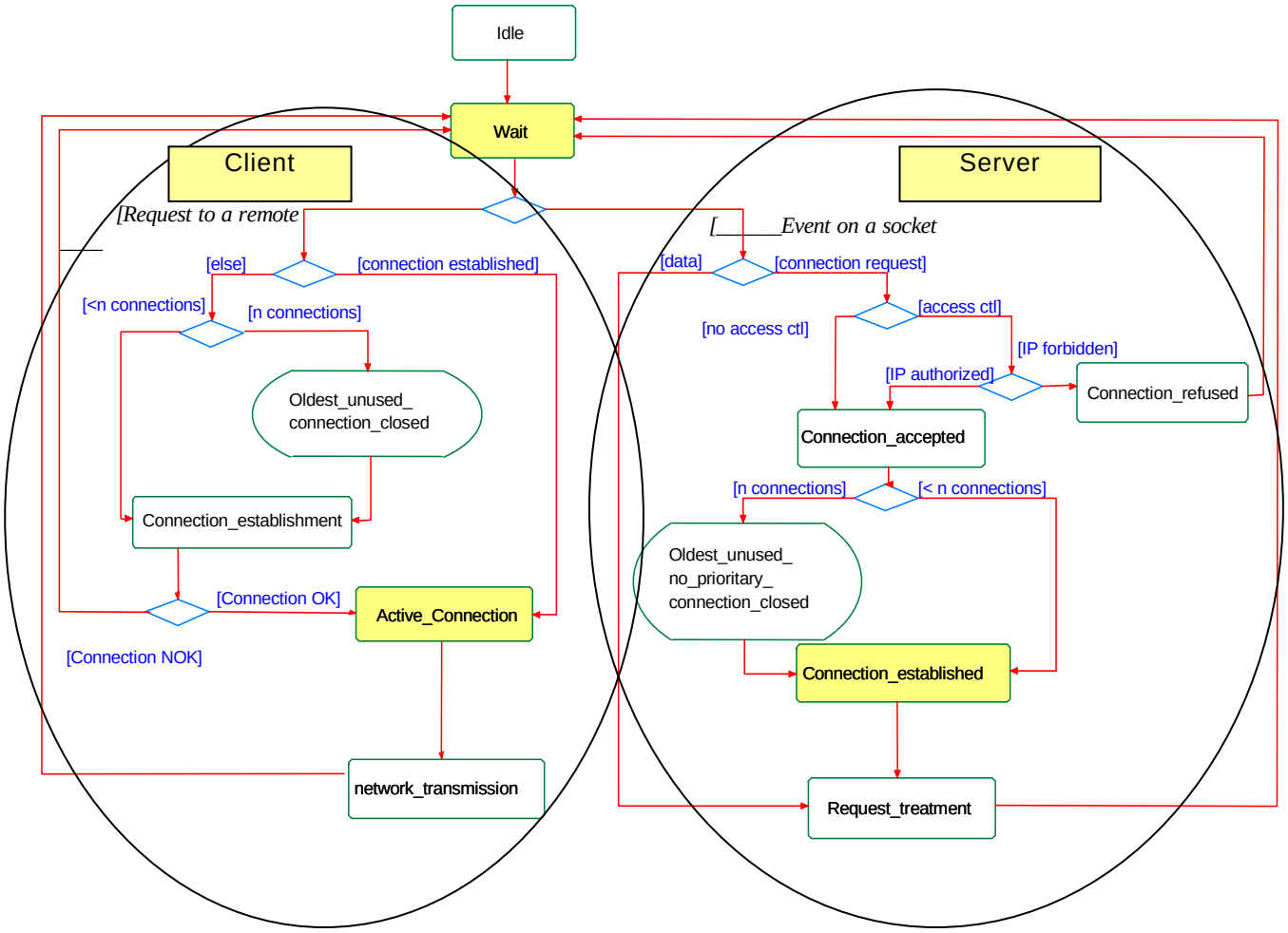



**Figure 7: TCP connection management activity diagram**

1. **Explicit TCP connection management**

   The user application module is in charge of managing all the TCP connections: active and passive establishment, connection ending, etc. This management is done for all MODBUS communication between a client and a server. The BSD Socket interface is used in the user application module to manage the TCP connection. This solution offers a total flexibility but it implies that the application programmer has sufficient TCP knowledge.
   A limit of number of client and server connections has to be configured taking into account the device capabilities and requirement.

2. **Automatic TCP connection management**

   The TCP connection management is totally transparent for the user application module. The connection management module may accept a sufficient number of client and server connections.
   Nevertheless a mechanism must be implemented in case of exceeding the number of authorized connection. In such a case we recommend to close the oldest unused connection.
   A connection with a remote partner is established at the first packet received from a remote client or from the local user application. This connection will be closed if a termination arrived from the network or decided locally on the device. On reception of a connection request, the access control option can be used to forbid device accessibility to unauthorized clients.

The TCP connection management module uses the Stack interface (usually BSD Socket interface) to communicate with the TCP/IP stack.

In order to maintain compatibility between system requirements and server resources, the TCP management will maintain 2 pools of connection.

- The first pool **( priority connection pool**) is made of connections that are never closed on a local initiative. A configuration must be provided to set this pool up. The principle to be implemented is to associate a specific IP address with each possible connection of this pool. The devices with such IP addresses are said to be "marked". Any new connection that is requested by a marked device must be accepted, and will be taken from the priority connection pool. It is also necessary to configure the maximum number of Connections allowed for each remote device to avoid that the same device uses all the connections of the priority pool.
- The second pool (**non-priority connection pool**) contains connections for non marked devices. The rule that takes over here is to close the oldest connection when a new connection request arrives from a non-marked device and when there is no more connection available in the pool.

A configuration might be optionally provided to assign the number of connections available in each pool. However (It is not mandatory) the designers can set the number of connections at design time if required.

#### 4.2.1.2 Connection management description

- **Connection establishment :**

The MODBUS messaging service must provide a listening socket on Port 502, which permits to accept new connection and to exchange data with other devices.
When the messaging service needs to exchange data with a remote server, it must open a new client connection with a remote Port 502 in order to exchange data with this distant. The local port must be higher than 1024 and different for each client connection.

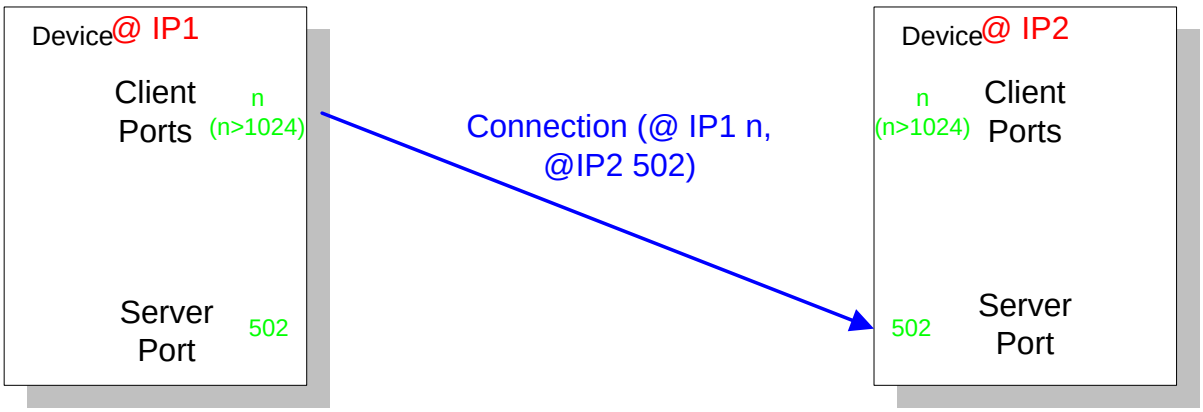



**Figure 8: MODBUS TCP connection establishment**

If the number of client and server connections is greater than the number of authorized connections the oldest unused connection is closed. The access control mechanism can be activated to check if the IP address of the remote client is authorized. If not the new connection is refused.

- **MODBUS data transfer**

A MODBUS request has to be sent on the right TCP connection already opened. The IP address of the remote is used to find the TCP connection. In case of multiple TCP connections opened with the same remote, one connection has to be chosen to send the MODBUS message, different choice criteria can be used like the oldest one, the first one. The connection has to maintain open during all the MODBUS communications. As described in the following sections a client can initiate several MODBUS transactions with a server without waiting the ending of the previous one.

- **Connection closing**

When the MODBUS communications are ended between a Client and a Server, the client has to initiate a connection closing of the connection used for these communications.

### 4.2.2 Impact of Operating Modes on the TCP Connection

Some Operating Modes (communication break between two operational End Points, Crash and Reboot on one of the End Point, ...) may have impacts on the TCP Connections. A connection can be seen closed or aborted on one side without the knowledge of the other side. The connection is said to be "**half-open"**.
This section describes the behavior for each main Operating Modes. It is assumed that the **Keep Alive** TCP mechanism is used on both end points (See section 4.3.2)

#### 4.2.2.1 Communication break between two operational end points:

The origin of the communication break can be the disconnection of the Ethernet cable on the Server side. The expected behavior is:

- If no packet is currently sent on the connection:
  The communication break will not be seen if it lasts less than the Keep Alive timer value. If the communication break lasts more than the Keep Alive timer value, an error is returned to the TCP Management layer that can reset the connection.
- If Some packets are sent before and after the disconnection:
  The TCP retransmission algorithms (Jacobson's, Karn's algorithms and exponential backoff. See section 4.3.2) are activated. This may lead to a stack TCP layer Reset of the Connection before the Keep Alive timer is over.

#### 4.2.2.2 Crash and Reboot of the Server end point

After the crash and Reboot of the Server, the connection is "**half-open"** on Client side. The expected behavior is:

- If no packet is sent on the half-open connection:
  The TCP half-open connection is seen opened from the Client side as long as the Keep Alive timer is not over. After that an error is returned to the TCP Management layer that can reset the connection.
- If some packets are sent on the half-open connection:
  The Server receives data on a connection that doesn't exist anymore. The stack TCP layer sends a Reset to close the half-open connection on the Client side

#### 4.2.2.3 Crash and Reboot of the Client

After the crash and Reboot of the Client, the connection is "**half-open"** on Server side. The expected behavior is:

- No packet is sent on the half-open connection:
  The TCP half-open connection is seen opened from the Server side as long as the Keep Alive timer is not over. After that an error is returned to the TCP Management layer that can reset the connection.
- If the Client opens a new connection before the Keep Alive timer is over :
  Two cases have to be studied:
  - The connection opening has the **same characteristics as the half-open connection** on the server side (same source and destination Ports, same source and destination IP Addresses), therefore the connection opening will fail at the TCP stack level after the Time-Out on Connection Establishment (75s on most of Berkeley implementations). To avoid this long Time-Out during which it is not possible to communicate, it is advised to ensure that different source port numbers than the previous one are used for a connection opening after a reboot on the client side.

  - The connection opening has **not the same characteristics as the half-open connection** on the server side (different source Ports, same destination Port, same source and destination IP Address ), therefore the connection is opened at the stack TCP level and signaled to the Server TCP Management layer.
    If the Server TCP Management layer only supports one connection from a remote Client IP Address, it can close the old half-opened connection and use the new one.
    If the Server TCP Management layer supports several connections from a remote Client IP Address, the new connection stays opened and the old one also stays half-opened until the expiration of the Keep Alive Timer that will return an error to the TCP Management layer. After that the TCP Management layer will be able to Reset the old connection.

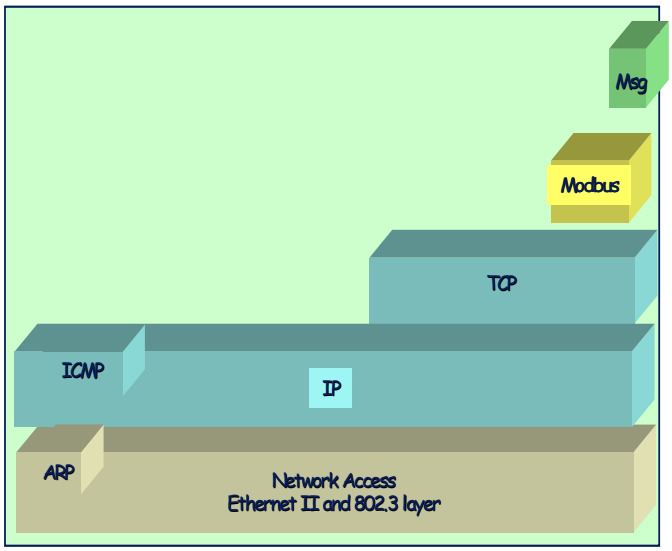
### 4.2.3 Access Control Module

The goal of this module is to check every new connection and using a list of authorized remote IP addresses the module can authorize or forbid a remote Client TCP connection.

In critical context the application programmer needs to choose the Access Control mode in order to secure its network access. In such a case he needs to Authorize/forbid access for each remote @IP. The user needs to provide a list of IP addresses and to specify for each IP address if it's authorized or not. By default, on security mode, the IP addresses not configured by the user are forbidden. Therefore with the access control mode a connection coming from an unknown IP address is closed.

## 4.3 USE of TCP/IP STACK

A TCP/IP stack provides an interface to manage connections, to send and receive data, and also to do some parameterizations in order to adapt the stack behavior to the device or system constraints.




The goal of this section is to give an overview of the Stack interface and also some information concerning the parameterization of the stack. This overview focuses on the features used by the MODBUS Messaging.

For more information, the advice is to read the RFC 1122 that provides guidance for vendors and designers of Internet communication software. It enumerates standard protocols that a host connected to the Internet must use as well as an explicit set of requirements and options.
The stack interface is generally based on the BSD (Berkeley Software Distribution) Interface that is described in this document.
.

### 4.3.1 Use of BSD Socket interface

*Remark : some TCP/IP stacks propose other types of interfaces for performance issues. A MODBUS client or server can use these specific interfaces, but this use will be not described in this specification.*

A socket is an endpoint of communication. It is the basic building block for communication. A MODBUS communication is executed by sending and receiving data through sockets. The TCPIP library provides only stream sockets using TCP and providing a connection-based communication service.
The Sockets are created via the **socket ()** function. A socket number is returned, which is then used by the creator to access the socket. Sockets are created without addresses (IP address and port number). Until a port is bound to a socket, it cannot be used to receive data.
The **bind ()** function is used to bind a port number to a socket. The **bind ()** creates an association between the socket and the port number specified.
In order to initiate a connection, the client must issue the **connect ()** function specifying the socket number, the remote IP address and the remote listening port number (active connection establishment).
In order to complete a connection, the server must issue the **accept ()** function specifying the socket number that was specified in the prior **listen ()** call (passive connection establishment). A new socket is created with the same properties as the initial one. This new socket is connected to the client's socket, and its number is returned to the server. The initial socket is thereby free for other clients that might want to connect with the server.

After the establishment of the TCP connection the data can be transferred. The **send()** and **recv()** functions are designed specifically to be used with sockets that are already connected.

The **setsockopt ()** function allows a socket's creator to associate options with a socket. These options modify the behavior of the socket. The description of these options is given in the section 4.3.2.

The **select ()** function allows the programmer to test events on all sockets.

The **shutdown ()** function allows a socket user to disable send () and/or receive () on the socket.

Once a socket is no longer needed, its socket descriptor can be discarded by using the **close ()** function.

Figure 39: MODBUS Exchanges describes a full MODBUS communication between a client and a s server. The Client establishes the connection and sends 3 MODBUS requests to the server without waiting the response of the first one. After receiving all the responses the Client closes the connection properly.

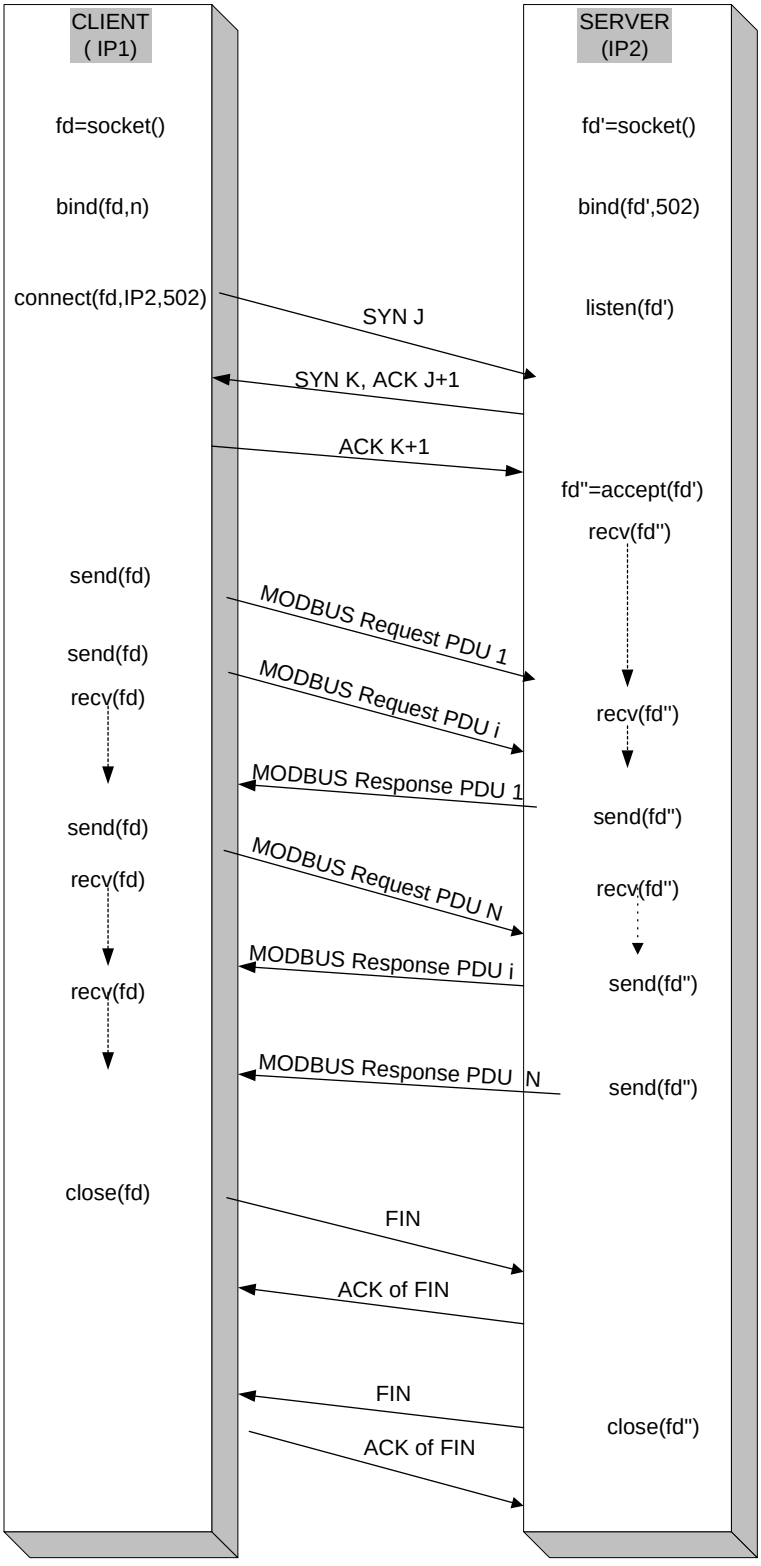



**Figure 9: MODBUS Exchanges**

### 4.3.2 TCP layer parameterization

Some parameters of the TCP/IP stack can be adjusted to adapt its behavior to the product or system constraints. The following parameters can be adjusted in the TCP layer:

- **Parameters for each connection:**

SO-RCVBUF, SO-SNDBUF:

These parameters allow setting the high water mark for the Send and the Receive Socket. They can be adjusted for flow control management. The size of the received buffer is the maximum size advertised window for that connection. Socket buffer sizes must be increased in order to increase performances. Nevertheless these values must be smaller than internal driver resources in order to close the TCP window before exhausting internal driver resources.

The received buffer size depends on the TCP Windows size, the TCP Maximum segment size and the time needed to absorb the incoming frames. With a Maximum Segment Size of 300 bytes (a MODBUS request needs a maximum of 256 bytes + the MBAP header size), if we need 3 frames buffering, the socket buffer size value can be adjusted to 900 bytes. For biggest needs and best-scheduled time, the size of the TCP window may be increased.

TCP-NODELAY:

Small packets (called tinygrams) are normally not a problem on LANs, since most LANs are not congested, but these tinygrams can lead to congestion on wide area networks. A simple solution, called the "NAGLE algorithm", is to collect small amounts of data and sends them in a single segment when TCP acknowledgments of previous packets arrive.
In order to have better real-time behavior it is recommended to send small amounts of data directly without trying to gather them in a single segment. That is why it is recommended to force the TCP-NODELAY option that disables the "NAGLE algorithm" on client and server connections.

SO-REUSEADDR:

When a MODBUS server closes a TCP connection initialized by a remote client, the local port number used for this connection cannot be reused for a new opening while that connection stays in the "Time-wait" state (during two MSL : Maximum Segment Lifetime).

It is recommended specifying the SO-REUSEADDR option for each client and server connection to bypass this restriction. This option allows the process to assign itself a port number that is part of a connection that is in the 2MSL wait for client and listening socket.

SO-KEEPALIVE:

By default on TCP/IP protocol no data are sent across an idle TCP connection. Therefore if no process at the ends of a TCP connection is sending data to the other, nothing is exchanged between the two TCP modules. This assumes that either the client application or the server application uses timers to detect inactivity in order to close a connection.
It is recommended to enable the KEEPALIVE option on both client and server connection in order to poll the other end to know if the distant has either crashed and is down or crashed and rebooted.
Nevertheless we must keep on mind that enabling KEEPALIVE can cause perfectly good connections to be dropped during transient failures, that it consumes unnecessary bandwidth on the network if the keep alive timer is too short.

- **Parameters for the whole TCP layer:**

  Time Out on establishing a TCP Connection:

  Most Berkeley-derived systems set a time limit of 75 seconds on the establishment of a new connection, this default value should be adapted to the real time constraint of the application.

  Keep Alive parameters:

  The default idle time for a connection is 2 hours. Idles times in excess of this value trigger a keep alive probe. After the first keep alive probe, a probe is sent every 75 seconds for a maximum number of times unless a probe response is received.
  The maximum number of keep Alive probes sent out on an idle connection is 8. If no probe response is received after sending out the maximum number of keep Alive probes,TCP signals an error to the application that can decide to close the connection

  Time-out and retransmission parameters:

  A TCP packet is retransmitted if its loss has been detected. One way to detect the loss is to manage a Retransmission Time-Out (RTO) that expires if no acknowledgement has been received from the remote side.
  TCP manages a dynamic estimation of the RTO. For that purpose a Round-Trip Time (RTT) is measured after the send of every packet that is not a retransmission. The Round-Trip Time (RTT) is the time taken for a packet to reach the remote device and to get back an acknowledgement to the sending device. The RTT of a connection is calculated dynamically, nevertheless if TCP cannot get an estimate within 3 seconds, the default value of the RTT is set to 3 seconds.
  If the RTO has been estimated, it applies to the next packet sending. If the acknowledgement of the next packet is not received before the estimated RTO expiration, the **Exponential BackOff** is activated. A maximum number of retransmissions of the same packet is allowed during a certain amount of time. After that if no acknowledgement has been received, the connection is aborted.
  The maximum number of retransmissions and the maximum amount of time before the abort of the connection (tcp_ip_abort_interval) can be set up on some stacks.

  Some retransmission algorithms are defined in TCP standards :

  - The **Jacobson's RTO estimation algorithm** is used to estimate the Retransmission Time-Out (RTO),
  - The **Karn's algorithm** says that the RTO estimation should not be done on a retransmitted segment,
  - The **Exponential BackOff** defines that the retransmission time-out is doubled for each retransmission with an upper limit of 64 seconds.
  - The **fast retransmission algorithm** allows retransmitting after the reception of three duplicate acknowledgments. This algorithm is advised because on a LAN it may lead to a quicker detection of the loss of a packet than waiting for the RTO expiration.

  The use of these algorithms is recommended for a MODBUS implementation.

### 4.3.3 IP layer parameterization

#### 4.3.3.1 IP Parameters

The following parameters must be configured in the IP layer of a MODBUS implementation :

- Local IP Address : the IP address can be part of a Class A, B or C.

- Subnet Mask, : Subnetting an IP Network can be done for a variety of reasons : use of different physical media (such as Ethernet, WAN, etc.), more efficient use of

network addresses, and the capability to control network traffic. The Subnet Mask has to be consistent with the IP address class of the local IP address.

- Default Gateway: The IP address of the default gateway has to be on the same subnet as the local IP address. The value 0.0.0.0 is forbidden. If no gateway is to be defined then this value is to be set to either 127.0.0.1 or the Local IP address.

*Remark : The MODBUS messaging service doesn't require the fragmentation function in the IP layer.*

The local IP End Point shall be configured with a **local IP Address** and with **a Subnet Mask** and a **Default Gateway** (different from 0.0.0.0) .

## 4.4 COMMUNICATION APPLICATION LAYER

### 4.4.1 MODBUS Client

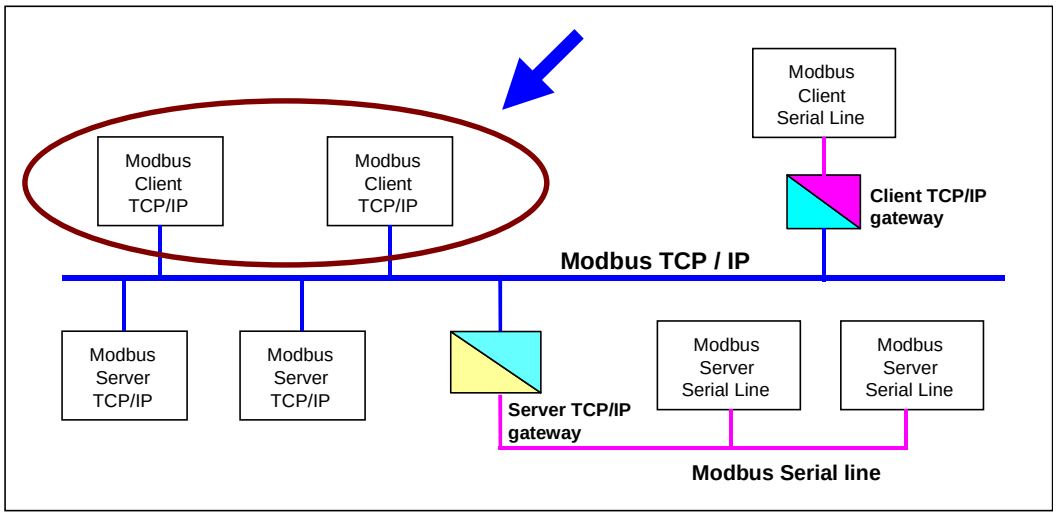



**Figure 10: MODBUS Client unit**

#### 4.4.1.1 MODBUS client design

The definition of the MODBUS/TCP protocol allows a simple design of a client. The following activity diagram describes the main treatments that are processed by a client to send a MODBUS request and to treat a MODBUS response.

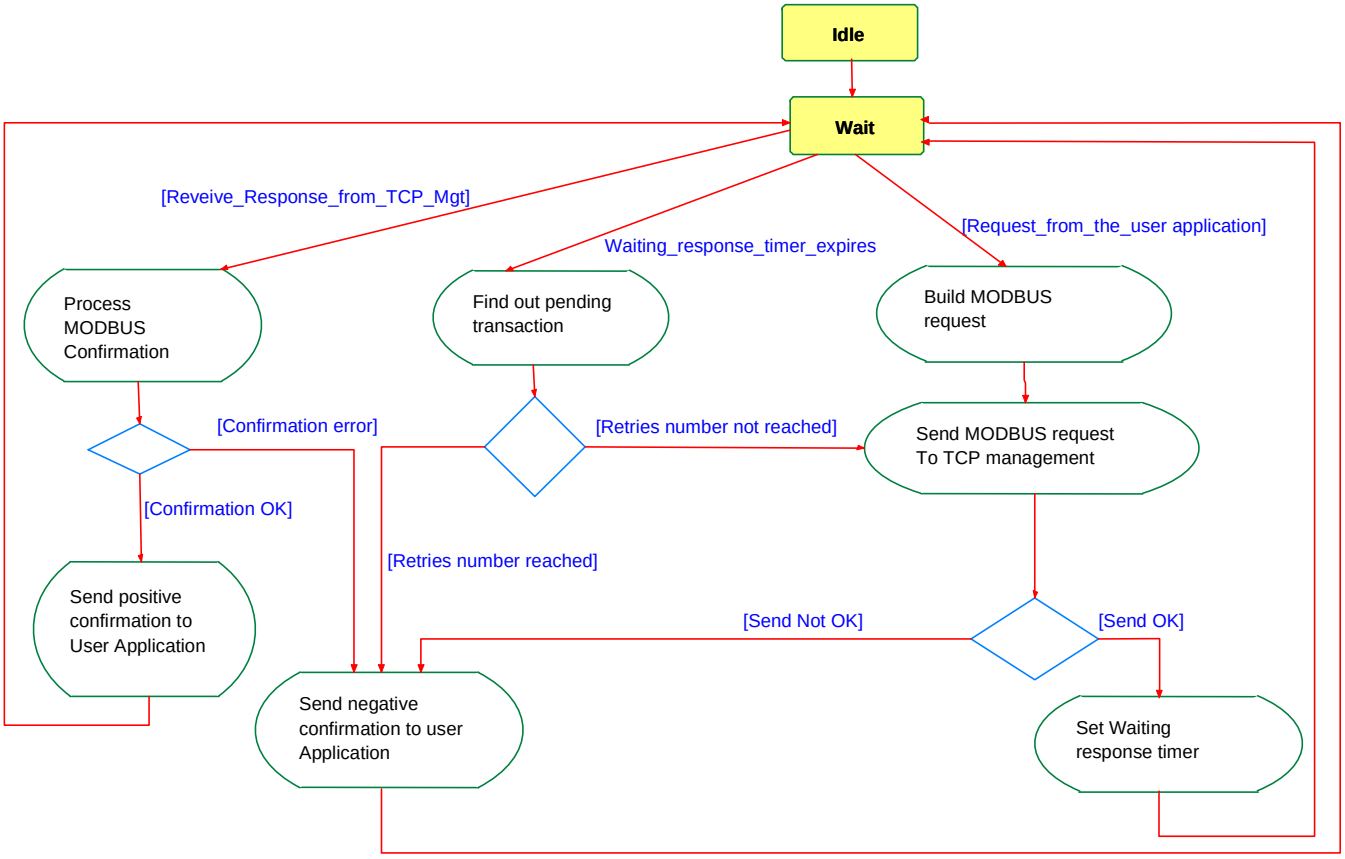



**Figure 11: MODBUS Client Activity Diagram**

A MODBUS client can receive three events:

- A new demand from the user application to send a request, in this case a MODBUS request has to be encoded and be sent on the network using the TCP management component service. The lower layer ( TCP management module) can give back an error due to a TCP connection error, or some other errors.
- A response from the TCP management, in this case the client has to analyze the content of the response and send a confirmation to the user application
- The expiration of a Time out due to a non-response. A new retry can be sent on the network or a negative confirmation can be sent to the User Application. *Remark : These retries are initiated by the MODBUS client, some other retries can also be done by the TCP layer in case of TCP acknowledge lack.*

#### 4.4.1.2 Build a MODBUS Request

Following the reception of a demand from the user application, the client has to build a MODBUS request and to send it to the TCP management.
Building the MODBUS request can be split in several sub-tasks:

- The instantiation of a MODBUS transaction that enables the Client to memorize all required information in order to bind later the response to the request and to send the confirmation to the user application.
- The encoding of the MODBUS request (PDU + MPAB header). The user application that initiates the demand has to provide all required information which enables the Client to encode the request. The MODBUS PDU is encoded according to the MODBUS Application Protocol Specification [1]. (MB function code, associated parameters and application data ). All fields of the MBAP header are filled. Then, the MODBUS request ADU is built prefixing the PDU with the MBAP header

- The sending of the MODBUS request ADU to the TCP management module which is in charge of finding the right TCP socket towards the remote Server. In addition to the MODBUS ADU the Destination IP address must also be passed.

The following activity diagram describes, more deeply than in Figure 11 MODBUS Client Activity Diagram, the request building phase.

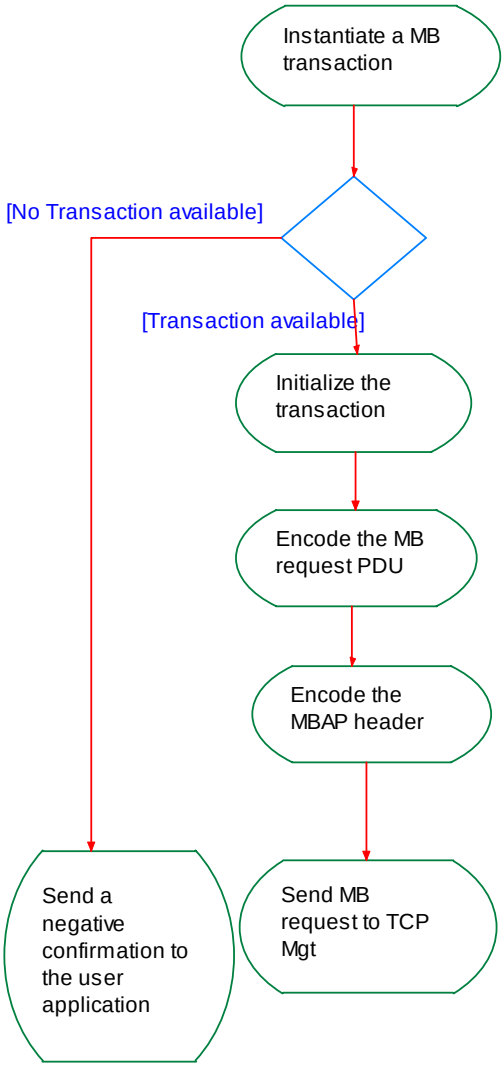



**Figure 12: Request building activity diagram**

The following example describes the MODBUS request ADU encoding for reading the register # 5 in a remote server :

- MODBUS Request ADU encoding :

| | Description | Size | Example |
|---|---|---|---|
| MBAP Header | Transaction Identifier Hi | 1 | 0x15 |
| | Transaction Identifier Lo | 1 | 0x01 |
| | Protocol Identifier | 2 | 0x0000 |
| | Length | 2 | 0x0006 |
| | Unit Identifier | 1 | 0xFF |

| | | | |
|---|---|---|---|
| *MODBUS request* | *Function Code (*)* | *1* | *0x03* |
| | *Starting Address* | *2* | *0x0004* |
| | *Quantity of Registers* | *2* | *0x0001* |

(*) please see the MODBUS Application Protocol Specification [1].

- **Transaction Identifier**

  The transaction identifier is used to associate the future response with the request. So, at a time, on a TCP connection, this identifier must be unique. There are several manners to use the transaction identifier:
  - For example, it can be used as a simple "TCP sequence number" with a counter which is incremented at each request.
  - It can also be judiciously used as a smart index or pointer to identify a transaction context in order to memorize the current remote server and the pending MODBUS request.

Normally, on MODBUS serial line a client must send one request at a time. This means that the client must wait for the answer to the first request before sending a second request. On TCP/MODBUS, several requests can be sent without waiting for a confirmation to the same server. The MODBUS/TCP to MODBUS serial line gateway is in charge of ensuring compatibility between these two behaviors.
The number of requests accepted by a server depends on its capacity in term of number of resources and size of the TCP windows. In the same way the number of transactions initialized simultaneously by a client depends also on its resource capacity. This implementation parameter is called "**NumberMaxOfClientTransaction**" and must be described as one of the MODBUS client features. Depending of the device type this parameter can take a value from 1 to 16.

- **Unit Identifier**

  This field is used for routing purpose when addressing a device on a MODBUS+ or MODBUS serial line sub-network. In that case, the “Unit Identifier” carries the MODBUS slave address of the remote device:

  If the MODBUS server is connected to a MODBUS+ or MODBUS Serial Line sub-network and addressed through a bridge or a gateway, the MODBUS Unit identifier is necessary to identify the slave device connected on the sub-network behind the bridge or the gateway. The destination IP address identifies the bridge itself and the bridge uses the MODBUS Unit identifier to forward the request to the right slave device.
  The MODBUS slave device addresses on serial line are assigned from 1 to 247 (decimal). Address 0 is used as broadcast address.

  On TCP/IP, the MODBUS server is addressed using its IP address; therefore, the MODBUS Unit Identifier is useless. The value 0xFF has to be used.

  When addressing a MODBUS server connected directly to a TCP/IP network, it’s recommended not using a significant MODBUS slave address in the “Unit Identifier” field. In the event of a re-allocation of the IP addresses within an automated system and if a IP address previously assigned to a MODBUS server is then assigned to a gateway, using a significant slave address may cause trouble because of a bad routing by the gateway. Using a non-significant slave address, the gateway will simply discard the MODBUS PDU with no trouble. 0xFF is recommended for the “Unit Identifier" as non-significant value.
  *Remark : The value 0 is also accepted to communicate directly to a MODBUS/TCP device*.

#### 4.4.1.3 Process MODBUS Confirmation

When a response frame is received on a TCP connection, the Transaction Identifier carried in the MBAP header is used to associate the response with the original request previously sent on that TCP connection:

- If the Transaction Identifier doesn't refer to any MODBUS pending transaction, the response must be discarded.
- If the Transaction Identifier refers to a MODBUS pending transaction, the response must be parsed in order to send a MODBUS Confirmation to the User Application (positive or negative confirmation)

Parsing the response consists in verifying the MBAP Header and the MODBUS PDU response:

- **MBAP Header**

  After the verification of the Protocol Identifier that must be 0x0000, the length gives the size of the MODBUS response.

  If the response comes from a MODBUS server device directly connected to the TCP/IP network, the TCP connection identification is sufficient to unambiguously identify the remote server. Therefore, the Unit Identifier carried in the MBAP header is not significant (value 0xFF) and must be discarded.

  If the remote server is connected on a Serial Line sub-network and the response comes from a bridge, a router or a gateway, then the Unit Identifier (value != 0xFF) identifies the remote MODBUS server which has originally sent the response.

- **MODBUS Response PDU**

  The function code must be verified and the MODBUS response format analyzed according to the MODBUS Application Protocol:

  - if the function code is the same as the one used in the request, and if the response format is correct, then the MODBUS response is given to the user application as a **Positive Confirmation**.
  - If the function code is a MODBUS exception code (Function code + 80H), the MODBUS exception response is given to the user application as a **Positive Confirmation**.
  - If the function code is different from the one used in the request (=non expected function code), or if the format of the response is incorrect, then an error is signaled to the user application using a **Negative Confirmation**.

*Remark: A positive confirmation is a confirmation that the command was received and responded to by the server. It does not imply that the server was able to successfully act on the command (failure to successfully act on the command is indicated by the MODBUS Exception response).*

The following activity diagram describes, more deeply than in Figure 11 MODBUS Client Activity Diagram, the confirmation processing phase.

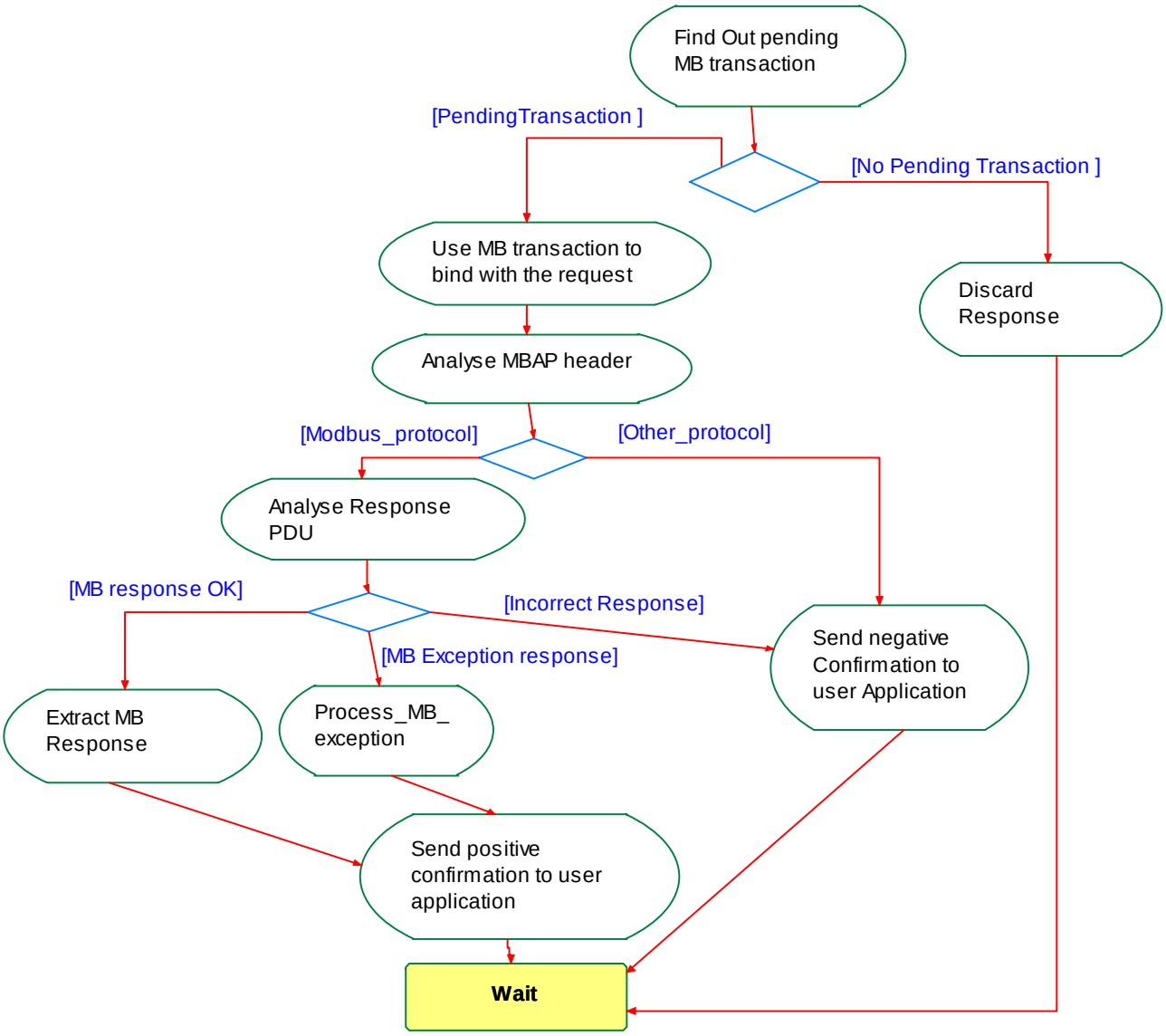



**Figure 13: Process MODBUS Confirmation activity diagram**

#### 4.4.1.4 Time-out management

There is deliberately NO specification of required response time for a transaction over MODBUS/TCP.
This is because MODBUS/TCP is expected to be used in the widest possible variety of communication situations, from I/O scanners expecting sub-millisecond timing to long distance radio links with delays of several seconds.
From a client perspective, the timeout must take into account the expected transport delays across the network, to determine a 'reasonable' response time. Such transport delays might be milliseconds for a switched Ethernet, or hundreds of milliseconds for a wide area network connection.
In turn, any 'timeout' time used at a client to initiate an application retry should be larger than the expected maximum 'reasonable' response time. If this is not followed, there is a potential for excessive congestion at the target device or on the network, which may in turn cause further errors. This is a characteristic, which should always be avoided.

So in practice, the client timeouts used in high performance applications are always likely to be somewhat dependent on network topology and expected client performance. Applications which are not time critical can often leave timeout values to the normal TCP defaults, which will report communication failure after several seconds on most platforms.

### 4.4.2 MODBUS Server

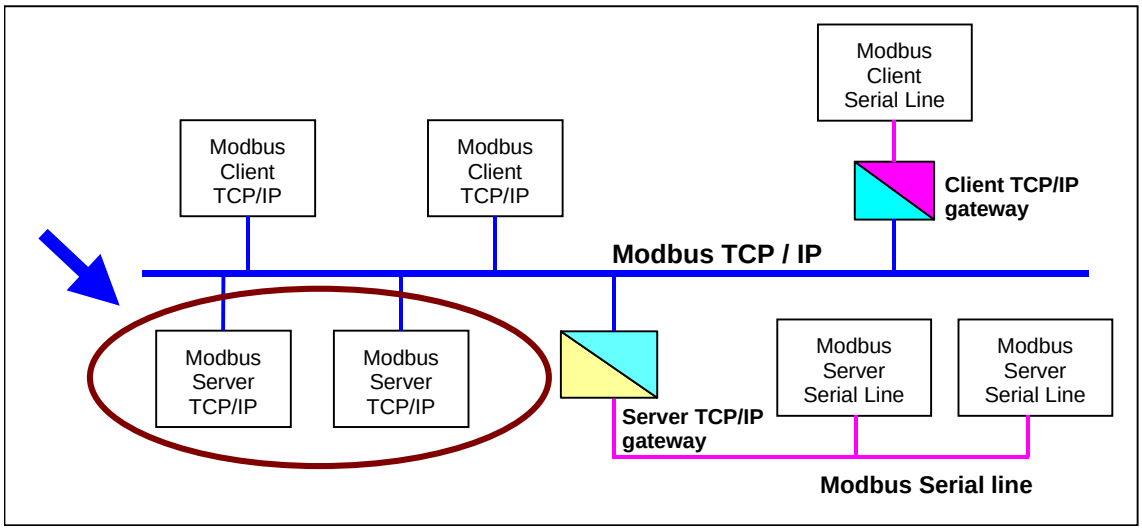



**Figure 14: MODBUS Server unit**

The role of a MODBUS server is to provide access to application objects and services to remote MODBUS clients.

Different kind of access may be provided depending on the user application :

- simple access like get and set application objects attributes
- advanced access in order to trigger specific application services

The MODBUS server has:

- To map application objects onto readable and writable MODBUS objects, in order to get or set application objects attributes.
- To provide a way to trigger services onto application objects.

In run time the MODBUS server has to analyze a received MODBUS request, to process the required action, and to send back a MODBUS response.

Informative Note: The application objects and services of the Backend Interface obtain the requested data based upon the function code, and the User is responsible.

#### 4.4.2.1 MODBUS Server Design

The MODBUS Server design depends on both :

- the kind of access to the application objects (simple access to attributes or advanced access to services)
- the kind of interaction between the MODBUS server and the user application (synchronous or asynchronous).

The following activity diagram describes the main treatments that are processed by the Server to obtain a MODBUS request from TCP Management, then to analyze the request, to process the required action, and to send back a MODBUS response.

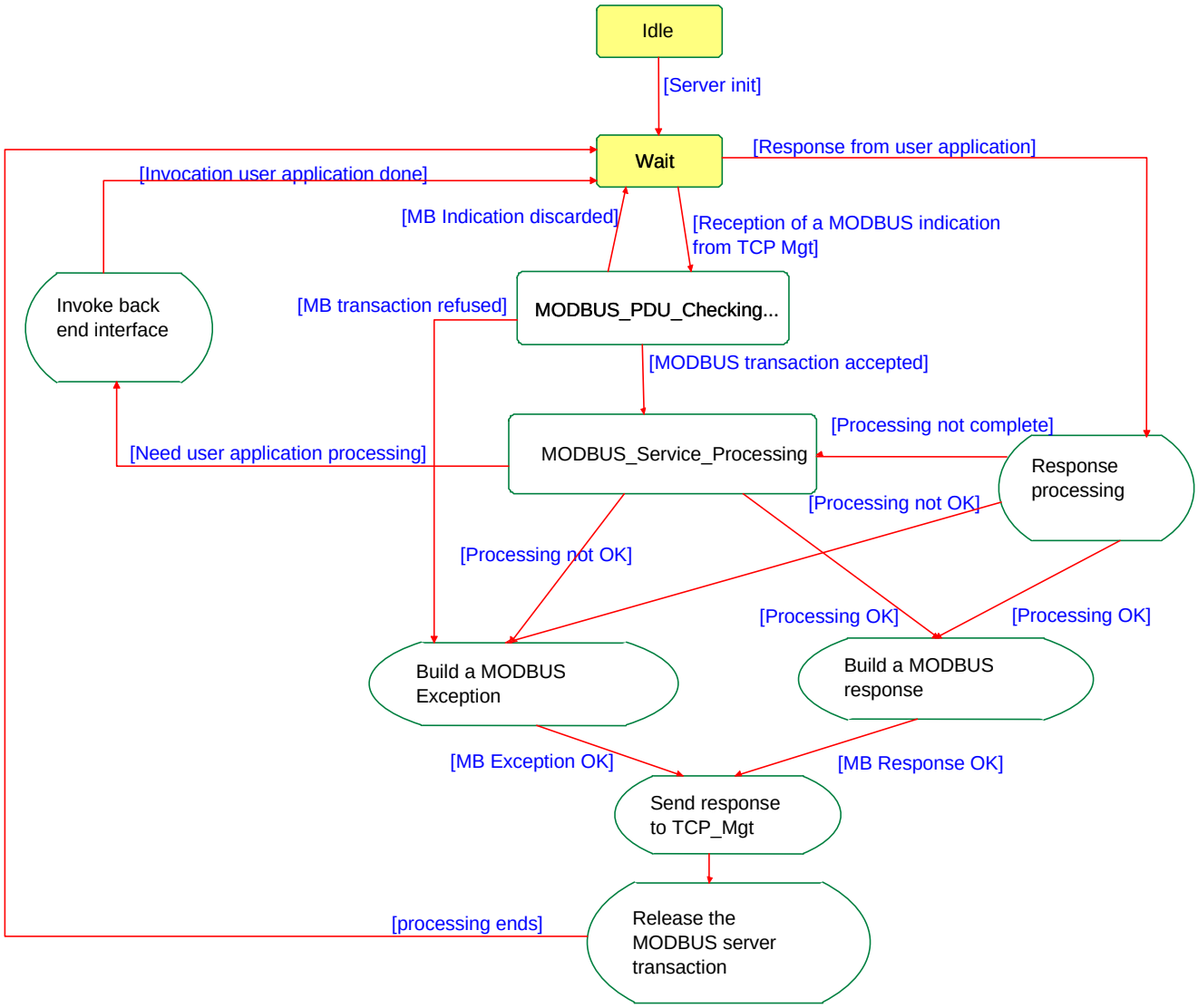



**Figure 15: Process MODBUS Indication activity diagram**

As shown in the previous activity diagram:

- Some services can be immediately processed by the MODBUS Server itself, with no direct interaction with the User Application ;
- Some services may require also interacting explicitly with the User Application to be processed ;
- Some other advanced services require invoking a specific interface called MODBUS Back End service. For example, a User Application service may be triggered using a sequence of several MODBUS request/response transactions according to a User Application level protocol. The Back End service is responsible for the correct processing of all individual MODBUS transactions in order to execute the global User Application service.

A more complete description is given in the following sections.

The MODBUS server can accept to serve simultaneously several MODBUS requests. The maximum number of simultaneous MODBUS requests the server can accept is one of the main characteristics of a MODBUS server. This number depends on the server design and its processing and memory capabilities. This implementation parameter is called "**NumberMaxOfSeverTransaction**" and must be described as one of the MODBUS server features. It may have a value from 1 to 16 depending on the device capabilities.

The behavior and the performance of the MODBUS server are significantly affected by the "NumberMaxOfTransaction" parameter. Particularly, it's important to note that the number of concurrent MODBUS transactions managed may affect the response time of a MODBUS request by the server.

#### 4.4.2.2 MODBUS PDU Checking

The following diagram describes the MODBUS PDU Checking activity.

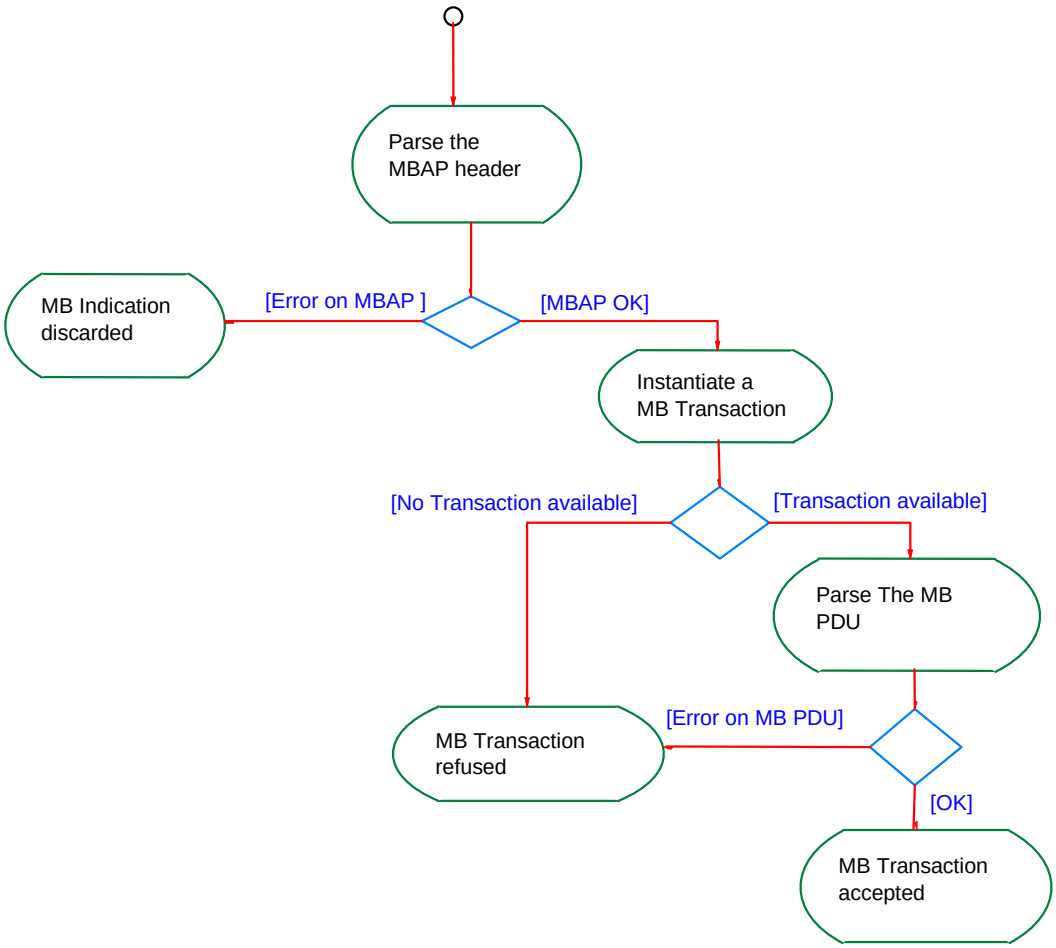



**Figure 16: MODBUS PDU Checking activity diagram**

The MODBUS PDU Checking function consists of first parsing the MBAP Header. The Protocol Identifier field has to be checked :

- If it is different from MODBUS protocol type, the indication is simply discarded.
- If it is correct (= MODBUS protocol type; value 0x00), a MODBUS transaction is instantiated.

The maximum number of MODBUS transactions the server can instantiate is defined by the "NumberMaxOfTransaction" parameter ( A system or a configuration parameter).

In case of no more transactions available, the server builds a MODBUS exception response (Exception code 6 : Server Busy).

If a MODBUS transaction is available, it's initialized in order to memorize the following information:

- The TCP connection identifier used to send the indication (given by the TCP Management)
- The MODBUS Transaction ID (given in MBAP Header)
- The Unit Identifier (given in MBAP Header)

Then the MODBUS PDU is parsed. The function code is first controlled :

- in case of invalidity a MODBUS exception response is built (Exception code 1 : Invalid function).
- If the function code is accepted, the server initiates the "MODBUS Service processing" activity.

#### 4.4.2.3 MODBUS service processing

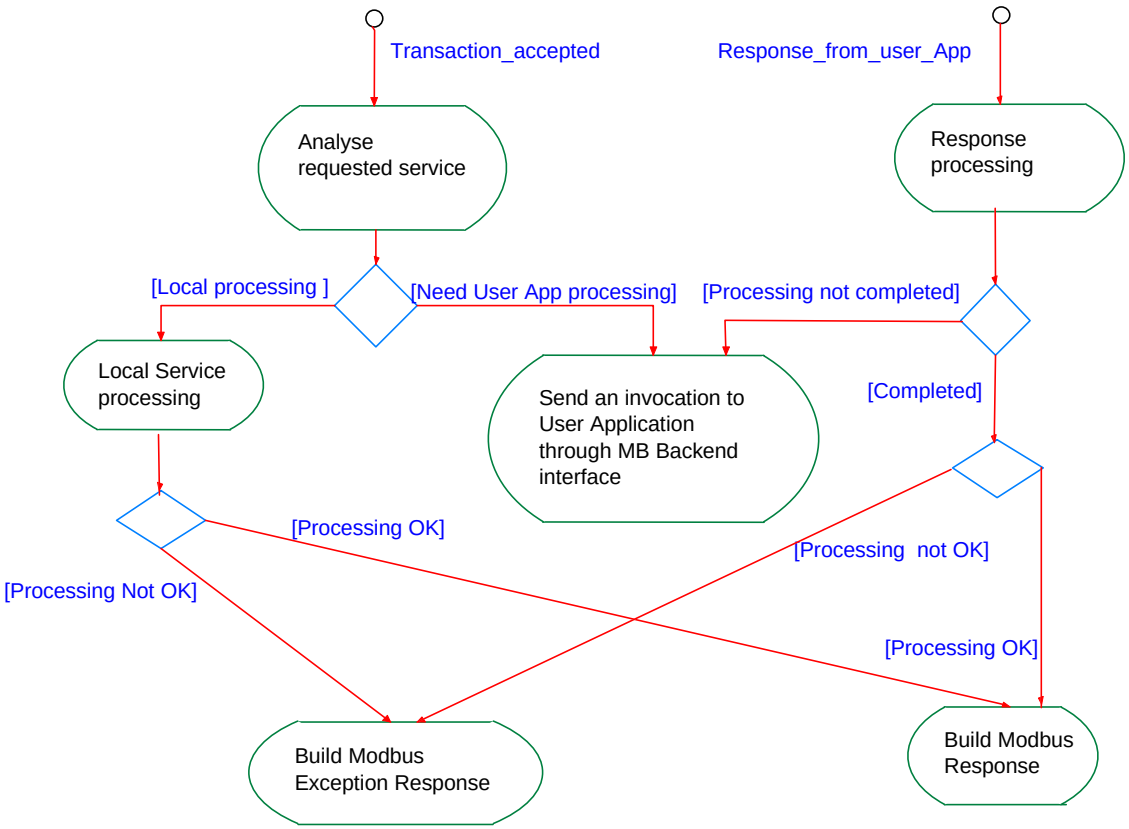



**Figure 17: MODBUS service processing activity diagram**

The processing of the required MODBUS service can be done in different ways depending on the device software and hardware architecture as described in the hereafter examples :

- Within a compact device or a mono-thread architecture where the MODBUS server can access directly to the user application data, the required service can be processed "locally" by the server itself without invoking the Back End service. The processing is done according to the MODBUS Application Protocol Specification [1]. In case of an error, a MODBUS exception response is built.
- Within a modular multi-processor device or a multi-thread architecture where the "communication layers" and the "user application layer" are 2 separate entities, some trivial services can be processed completely by the Communication entity while some others can require a cooperation with the User Application entity using the Back End service.

To interact with the User Application, the MODBUS Backend service must implement all appropriate mechanisms in order to handle User Application transactions and to manage correctly the User Application invocations and associated responses.

#### 4.4.2.4 User Application Interface (Backend Interface)

Several strategies can be implemented in the MODBUS Backend service to achieve its job although they are not equivalent in terms of user network throughput, interface bandwidth usage, response time, or even design workload.

The MODBUS Backend service will use the appropriate interface to the user application :

- Either a physical interface based on a serial link, or a dual-port RAM scheme, or a simple I/O line, or a logical interface based on messaging services provided by an operating system.
- The interface to the User Application may be synchronous or asynchronous.

The MODBUS Backend service will also use the appropriate design pattern to get/set objects attributes or to trigger services. In some cases, a simple "gateway pattern" will be adequate. In some other cases, the designer will have to implement a "proxy pattern" with a corresponding caching strategy, from a simple exchange table history to more sophisticated replication mechanisms.

The MODBUS Backend service has the responsibility to implement the protocol transcription in order to interact with the User Application. Therefore, it can have to implement mechanisms for packet fragmentation/reconstruction, data consistency guarantee, and synchronization whatever is required.

#### 4.4.2.5 MODBUS Response building

Once the request has been processed, the MODBUS server has to build the response using the adequate MODBUS server transaction and has to send it to the TCP management component.
Depending on the result of the processing two types of response can be built :

- A positive MODBUS response :
  - The response function code = The request function code

- A MODBUS Exception response :
  - The objective is to provide to the client relevant information concerning the error detected during the processing ;
  - The response function code = the request function code + 0x80 ;
  - The exception code is provided to indicate the reason of the error.

| Exception Code | MODBUS name | Comments |
|---|---|---|
| 01 | Illegal Function Code | The function code is unknown by the server |
| 02 | Illegal Data Address | Dependant on the request |
| 03 | Illegal Data Value | Dependant on the request |
| 04 | Server Failure | The server failed during the execution |
| 05 | Acknowledge | The server accepted the service invocation but the service requires a relatively long time to execute. The server therefore returns only an acknowledgement of the service invocation receipt. |
| 06 | Server Busy | The server was unable to accept the MB Request PDU. The client application has the responsibility of deciding if and when to re-send the request. |
| 0A | Gateway problem | Gateway paths not available. |
| 0B | Gateway problem | The targeted device failed to respond. The gateway generates this exception |

The MODBUS response PDU must be prefixed with the MBAP header which is built using data memorized in the transaction context.

- **Unit Identifier**
  The Unit Identifier is copied as it was given within the received MODBUS request and memorized in the transaction context.

- **Length**
  The server calculates the size of the MODBUS PDU plus the Unit Identifier byte. This value is set in the "Length" field.
- **Protocol Identifier**
  The Protocol Identifier field is set to 0x0000 (MODBUS protocol), as it was given within the received MODBUS request.
- **Transaction Identifier**
  This field is set to the "Transaction Identifier" value that was associated with the original request and memorized in the transaction context.

Then the MODBUS response must be returned to the right MODBUS Client using the TCP connection memorized in the transaction context. When the response is sent, the transaction context must be free.

# 5 IMPLEMENTATION GUIDELINE

The objective of this section is to propose an example of a messaging service implementation.
The model describes below can be used as a guideline during a client or a server implementation of a MODBUS messaging service.

Informative Note: The messaging service implementation is the responsibility of the User.

## 5.1 OBJECT MODEL DIAGRAM

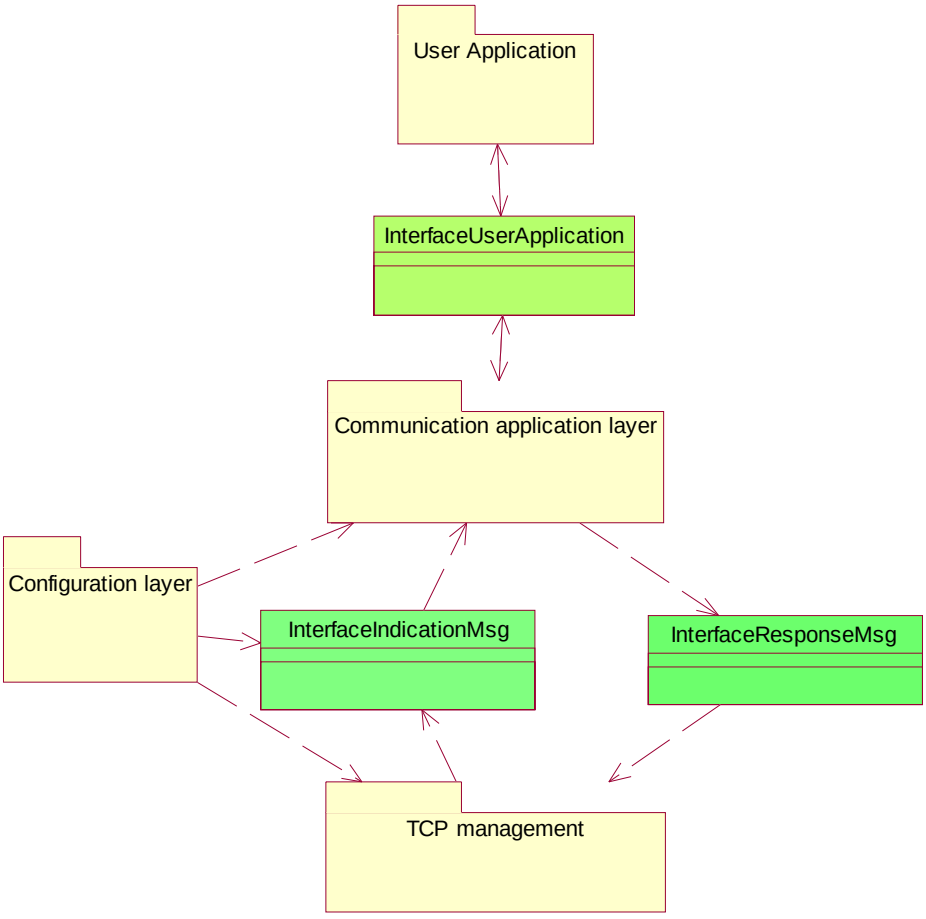



**Figure 18: MODBUS Messaging Service Object Model Diagram**

Four main packages compose the Object Model Diagram:

- **The Configuration layer** which configures and manages operating modes of components of other packages
- **The TCP Management** which interfaces the TCP/IP stack and the communication application layer managing TCP connection. It implies the management of socket interface.
- **The Communication application layer** which is composed by the MODBUS client on one side and the MODBUS server on the other side. This package is linked with the user application.

**The User application,** which corresponds to the device application, is completely dependent on the device and therefore it will be not part of this Specification.

This model is independent of implementation choices like the type of OS, the memory management, etc. In order to guarantee this independence generic Interface layers are used between the TCP management layer and the communication layer and between the communication layer and the user application layer.

Different implementations of this interface can be realized by the User: Pipe between two tasks, shared memory, serial link interface, procedural call, etc.

Some assumptions have to be taken to define the hereafter implementation model :

- Static memory management
- Synchronous treatment of the server
- One task to process the receptions on all sockets.

### 5.1.1 TCP management package

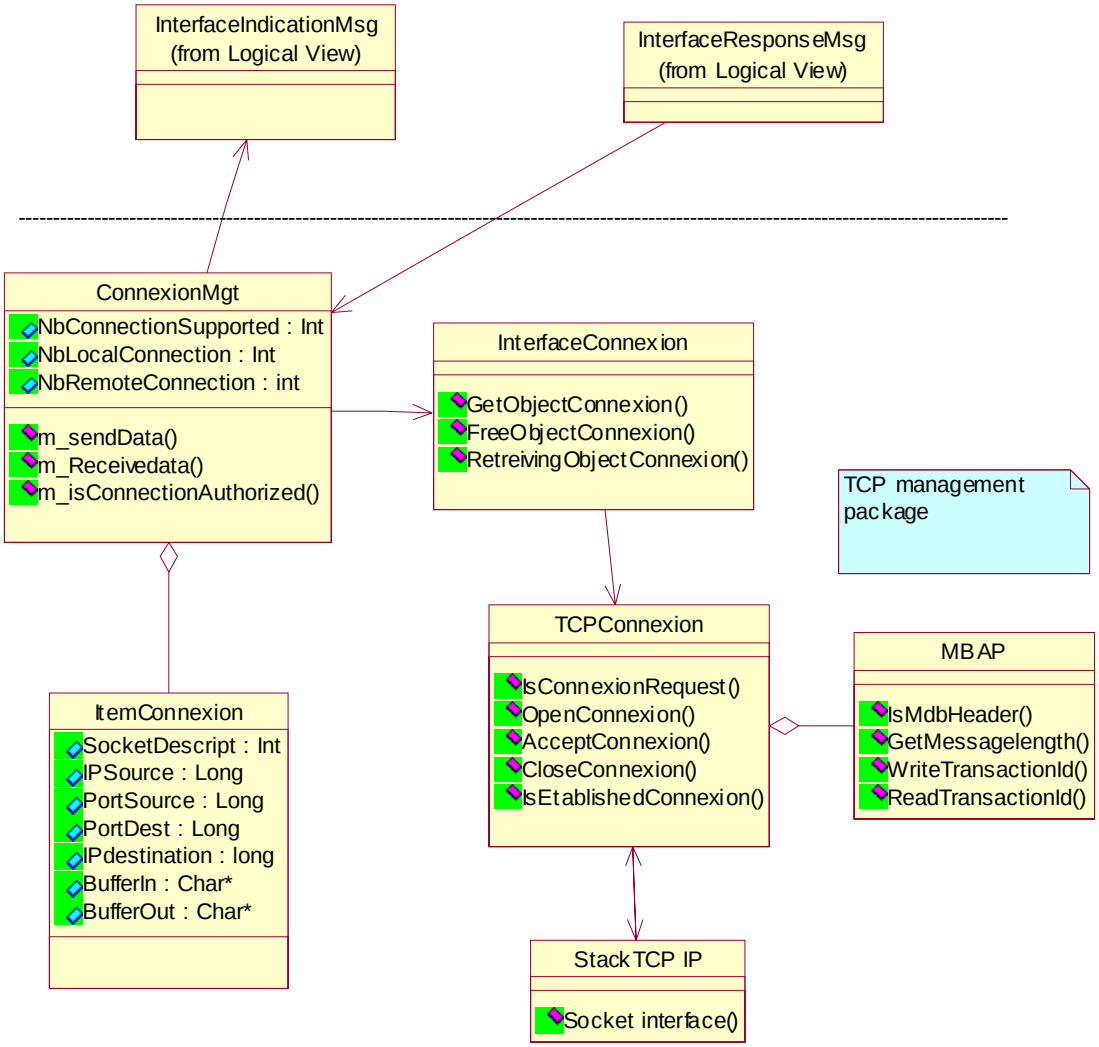



**Figure 19: MODBUS TCP management package**

The TCP management package comprises the following classes :

**CInterfaceConnexion**: The role of this class consists in managing memory pool for connections.

**CItemConnexion**: This class contains all information needed to describe a connection.

**CTCPConnexion**:, This class provides methods for managing automatically a TCP connection (Interface socket is provided by **CStackTCP_IP**).

**CConnexionMngt**: This class manages all connections and send query/response to MODBUS Server/MODBUS Client through **CinterfaceIndicationMsg** and **CInterfaceResponseMsg**. This class also treats the Access control for the connection opening.

**CMBAP**: This class provides methods for reading/writing/analyzing the MODBUS MBAP.

**CStackTCP_IP:** This class Implements socket services and provides parameterization of the stack.

### 5.1.2 Configuration layer package

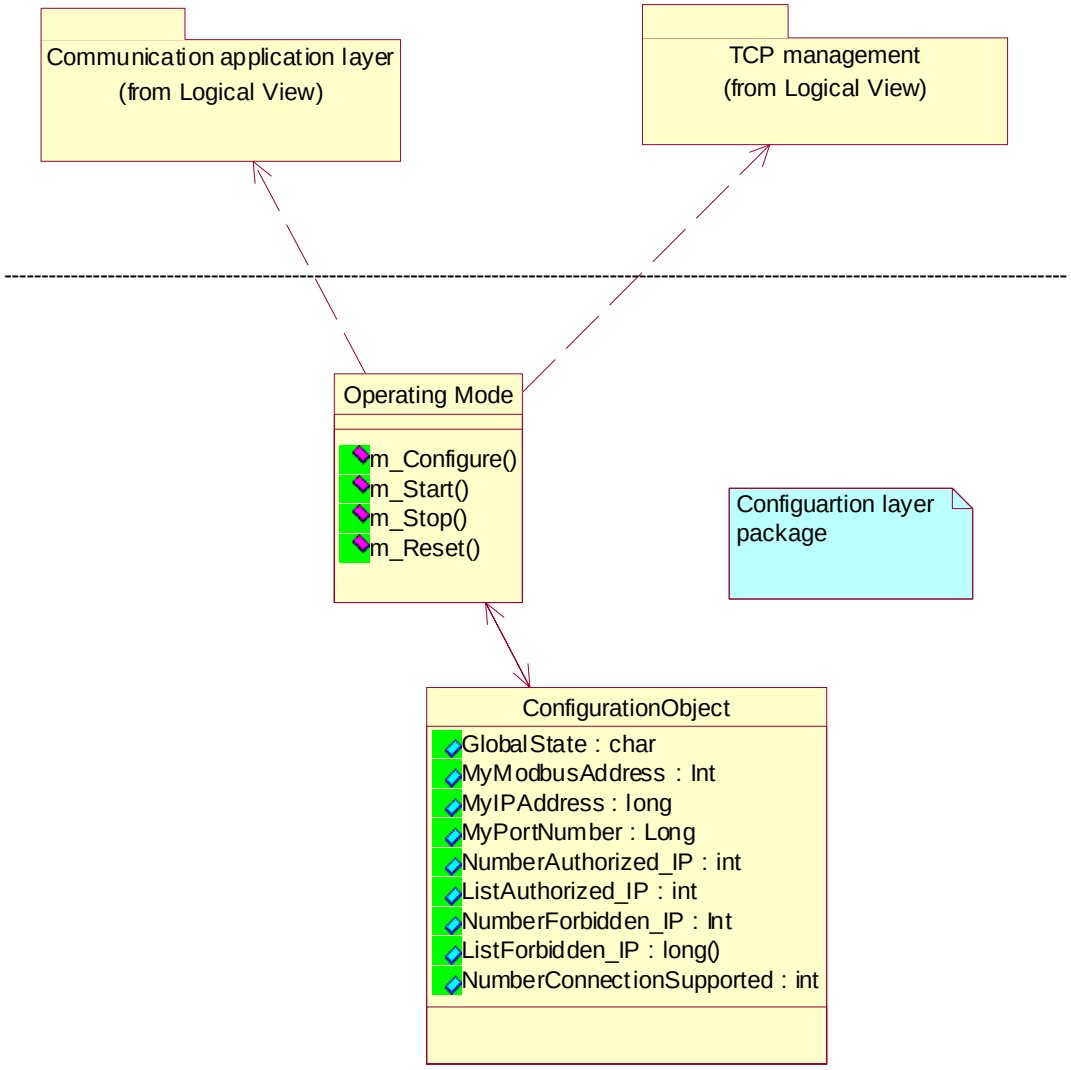



**Figure 20: MODBUS Configuration layer package**

The Configuration layer package comprises the following classes :

**TConfigureObject:** This class groups all data needed for configuring each other component. This structure is filled by the method m_Configure from the class **CoperatingMode**. Each class needing to be configured gets its own configuration data from this object. The configuration data is implementation dependent therefore the list of attributes of this class is provided as an example.

**COperatingMode:** The role of this class is to fill the **TConfigureObject** (according to the user configuration) and to manage the operating modes of the classes described below:

- CMODBUSServer
- CMODBUSClient

- CconnexionMngt

### 5.1.3 Communication layer package

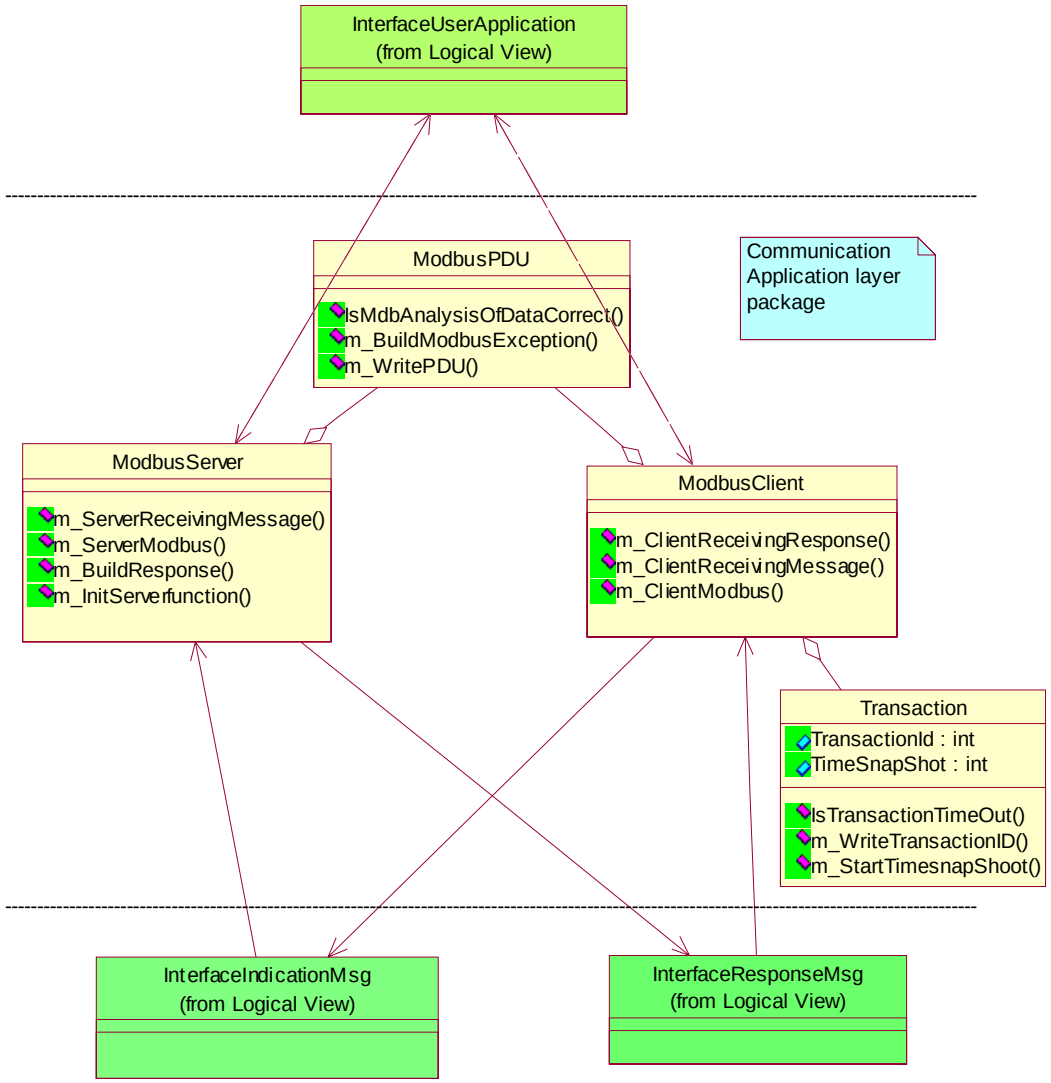



**Figure 21: MODBUS Communication Application layer package**

The Communication Application layer package comprises the following classes :

**CMODBUSServer**: MODBUS query is received from class **CInterfaceIndicationMsg** (by the method m_ServerReceivingMessage). The role of this class is to build the MODBUS response or the MODBUS Exception according the query (incoming from network). This class implements the Graph State of MODBUS server. Response can be built only if class **COperatingMode** has sent both user configuration and right operating modes.

**CMODBUSClient**: MODBUS query is read from class **CInterfaceUserApplication**, The client task receives query by the method m_ClientReceivingMessage. This class

implements the State Graph of MODBUS client and manages transactions for linking query with response (from network). Query can be sent over network only if class **CoperatingMode** has sent both user configuration and right operating modes.

**CTransaction**: This class implements methods and structures for managing transactions.

### 5.1.4 Interface classes

**CInterfaceUserApplication:** This class represents the interface with the user application, it provides two methods to access to the user data. In a real implementation this method can be implemented in different way depending of the hardware and software device capabilities (equivalent to an end-driver, example access to PCMCIA, shared memory, etc).

**CInterfaceIndicationMsg:** This Interface class is proposed for sending query from Network to the MODBUS Server, and for sending response from Network for the Client. This class interfaces TCPManagement and 'Communication Application Layer' packages (From Network). The implementation of this class is device dependent.

**CInterfaceResponseMsg**: This Interface class is used for receiving response from the Server and for sending query from the client to the Network. This class interfaces packages 'Communication Application Layer' and package 'TCPManagement' (To Network). The implementation of this class is device dependent.

## 5.2 IMPLEMENTATION CLASS DIAGRAM

The following Class Diagram describes the complete diagram of a proposal implementation.

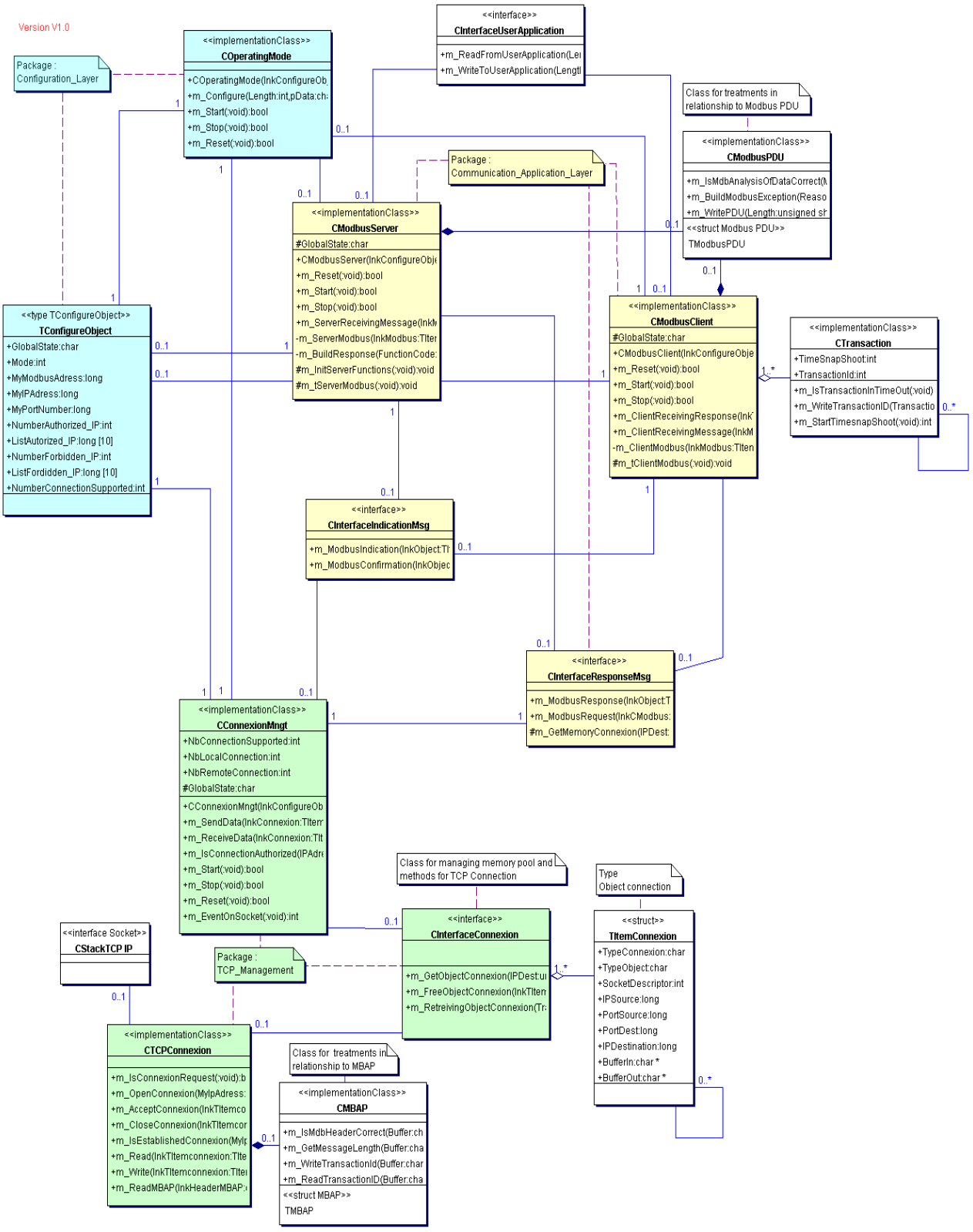



**Figure 22: Class Diagram**

## 5.3 SEQUENCE DIAGRAMS

Two Sequence diagrams are described hereafter are an example in order to illustrate a Client MODBUS transaction and a Server MODBUS transaction.

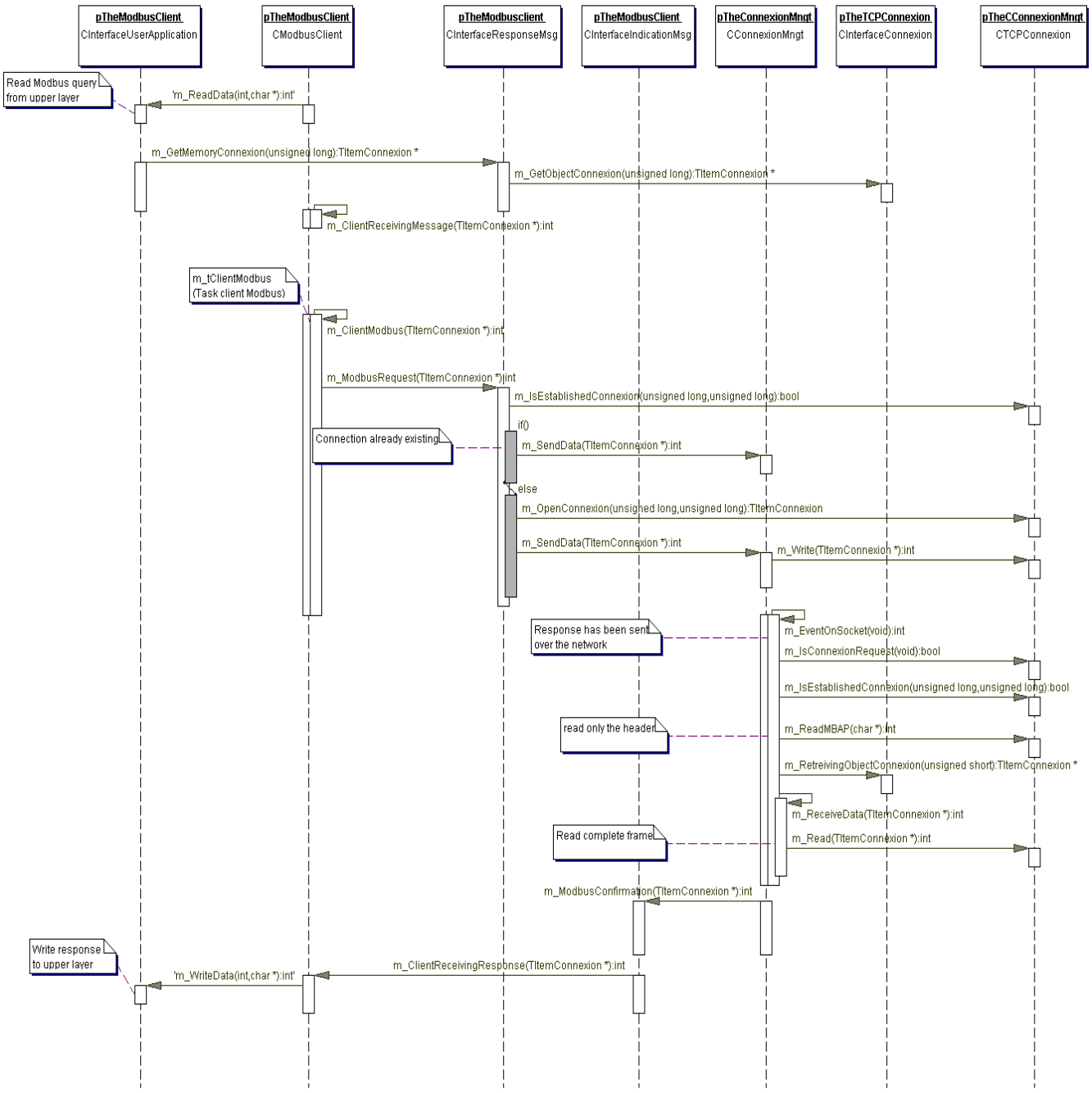



**Figure 23: MODBUS client sequence diagram**

General comments for a better understanding of the **Client** sequence diagram:

**First step**: A Reading query comes from User Application (method m_Read).

**Second Step**: The 'Client' task receives the MODBUS query (method **m_ClientReceivingMessage**). This is the entry point of the Client. To associate the query with the corresponding response when it will arrive, the Client uses a Transaction resource (Class **CTransaction**). The MODBUS query is sent to the TCP_Management by calling the class interface **CInterfaceResponseMsg** (method m_MODBUSRequest)

**Third Step:** If the connection is already established there is nothing to do on connection, the message can be send over the network. Otherwise, a connection must be opened before the message can be sent over the network.
At this time the client is waiting for a response (from a remote server)

**Fourth step:** Once a response has been received from the network, the TCP/IP stack receives data (method **m_EventOnSocket** is implicitly called).
If the connection is already established, then the MBAP is read for retrieving the connection object (connection object gives memory resource and other information).
Data coming from network is read and confirmation is sent to the client task via the class Interface **CInterfaceIndicationMsg** (method **m_MODBUSConfirmation**). Client task receives the MODBUS Confirmation (method **m_ClientReceivingResponse**).
Finally the response is–written to the user application (method m_WriteData), and transaction resource is freed.

Hereafter is an example of a MODBUS Server exchange.

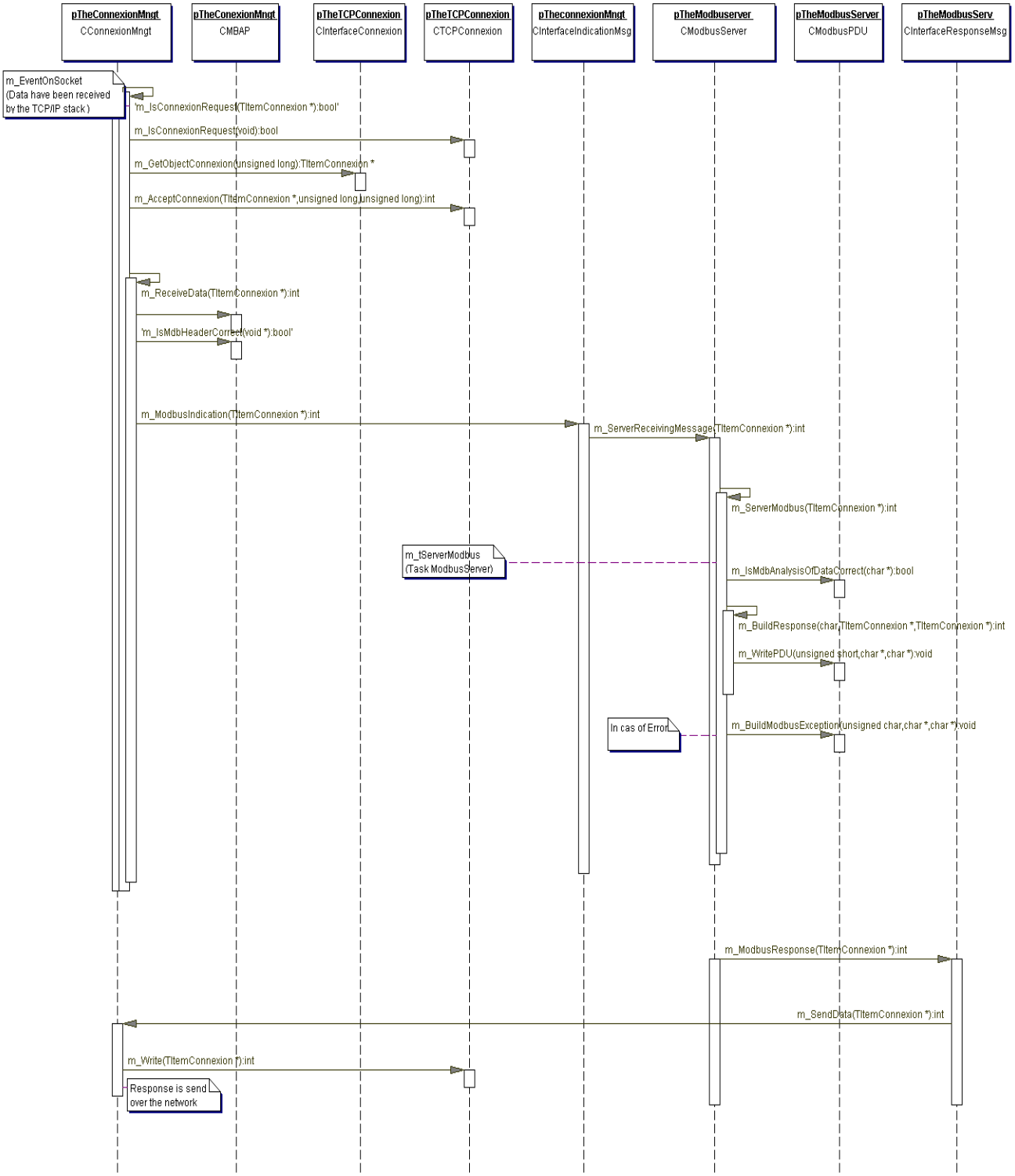



Figure 24: MODBUS server Diagram

General comments for a better understanding of the **Server** sequence diagram:

**First step**: a client has sent a query (MODBUS query) over the network.
The TCP/IP stack receives data (method **m_EventOnSocket** is implicitly called).

**Second step**: The query may be a connection request or not (method **m_IsConnexionRequest**).
If the query is a connection request, the connection object and buffers for receiving and sending the MODBUS frame are allocated (method **m_GetObjectConnexion**). Just after, the connection access control must be checked and accepted (method **m_AcceptConnexion**)

**Third step:** If the query is a MODBUS request, the complete MODBUS Query can be read (method **m_ReceiveData**). At this time the MBAP must be analyzed (method **m_IsMdbHeaderCorrect**). The complete frame is sent to the Server task via the **CinterfaceIndicationMessaging** Class (method **m_MODBUSIndication**). Server task receives the MODBUS Query (method **m_ServerReceivingMessage**) and analyses it.
If an error occurs (function code not supported, etc), a MODBUSException frame is built (**m_BuildMODBUSException**), otherwise the response is built.

**Fourth Step:** The response is sent over the network via the **CinterfaceResponseMessaging** (method **m_MODBUSResponse**). Treatment on the connection object is done by the method **m_SendData** (retrieve the connection descriptor, etc) and data is sent over the network.

### 5.4 CLASSES AND METHODS DESCRIPTION

#### 5.4.1 MODBUS Server Class

## Class CMODBUSServer

---

class **CMODBUSServer**

**Stereotype** implementationClass

Provides methods for managing MODBUS Messaging in Server Mode

---

| Field Summary | |
|---|---|
| protected char | **GlobalState**<br>state of the MODBUS Server |

| Constructor Summary | |
|---|---|
| **CMODBUSServer**(TConfigureObject * lnkConfigureObject)<br>Constructor : Create internal object | |

| Method Summary |
|---|

| | |
|---|---|
| protected void | **m_InitServerFunctions**(void )<br>Function called by the constructor for filling array of functions 'm_ServerFunction' |
| bool | **m_Reset**(void )<br>Method for Resetting Server, return true if reset |
| int | **m_ServerReceivingMessage**(TItemConnexion * lnkMODBUS)<br>Interface with CindicationMsg::m_MODBUSIndication for receiving Query from NetWork return negative value if problem |
| bool | **m_Start**(void )<br>Method for Starting Server, return true if Started |
| bool | **m_Stop**(void )<br>Method for Stopping Server, return true if Stopped |
| protected void | **m_tServerMODBUS**(void )<br>Server MODBUS task ... |

### 5.4.2 MODBUS Client Class

## Class CMODBUSClient

---

class **CMODBUSClient**

Provides methods for managing MODBUS Messaging in Client Mode

**Stereotype** implementationClass

---

| Field Summary | |
|---|---|
| protected char | **GlobalState**<br>State of the MODBUS Client |

| Constructor Summary | |
|---|---|
| **CMODBUSClient**(TConfigureObject * lnkConfigureObject)<br>Constructor : Create internal object , initialize to 0 variables. | |

| Method Summary | |
|---|---|
| int | **m_ClientReceivingMessage**(TItemConnexion * lnkMODBUS)<br>Interface provided for receiving message from application Layer Typically : Call CinterfaceUserApplication::m_Read for reading data call CInterfaceConnexion::m_GetObjectConnexion for getting memory for a transaction. Return negative value if problem |
| int | **m_ClientReceivingResponse**(TitemConnexion * lnkTItemConnexion)<br>Interface with CindicationMsg::m_Confirmation for receiving response from network return negative value if problem |
| bool | **m_Reset**(void )<br>Method for Resetting component, return true if reset |
| bool | **m_Start**(void )<br>Method for Starting component, return true if started |
| bool | **m_Stop**(void ) |

| | |
|---|---|
| | Method for Stopping component, return true if stopped |
| protected void | **m_tClientMODBUS**(void )<br>Client MODBUS task.... |

#### 5.4.3 Interface Classes

##### 5.4.3.1 Interface Indication class

### Class CInterfaceIndicationMsg

**Direct Known Subclasses:**

CConnexionMngt

---

class **CInterfaceIndicationMsg**

Class for sending message from TCP_Management to MODBUS Server or Client

**Stereotype** interface

---

| Method Summary | |
|---|---|
| int | **m_MODBUSConfirmation**(TItemConnexion * lnkObject)<br>Method for Receiving incoming Response, calling the Client : could be by reference, by Message Queue, Remote procedure Call, ... |
| int | **m_MODBUSIndication**(TItemConnexion * lnkObject)<br>Method for reading incoming MODBUS Query and calling the Server : could be by reference, by Message Queue, Remote procedure Call, ... |

##### 5.4.3.2 Interface Response Class

### Class CInterfaceResponseMsg

**Direct Known Subclasses:**

CMODBUSClient, CMODBUSServer

---

class **CInterfaceResponseMsg**

Class for sending response or sending query to TCP_Management from Client or Server

**Stereotype** interface

---

| Method Summary | |
|---|---|
| TitemConnexion * | **m_GetMemoryConnexion**(unsigned long IPDest)<br>Get an object ITemConnexion from memory pool. Return -1 if not enough memory |
| int | **m_MODBUSRequest**(TItemConnexion * lnkCMODBUS)<br>Method for Writing incoming MODBUS Query Client to ConnexionMngt : |

| | |
|---|---|
| | could be by reference, by Message Queue, Remote procedure Call, ... |
| int | **m_MODBUSResponse**(TItemConnexion * lnkObject)<br>Method for writing Response from MODBUS Server to ConnexionMngt could be by reference, by Message Queue, Remote procedure Call, ... |

#### 5.4.4 Connexion Management class

### Class CConnexionMngt

---

class **CConnexionMngt**

Class that manages all TCP Connections

**Stereotype** implementationClass

---

| Field Summary | |
|---|---|
| protected char | **GlobalState**<br>Global State of the Component ConnexionMngt |
| Int | **NbConnectionSupported**<br>Global number of connections |
| Int | **NbLocalConnection**<br>Number of connections opened by the local Client to a remote Server |
| Int | **NbRemoteConnection**<br>Number of connections opened by a remote Client to the local Server |

| Constructor Summary | |
|---|---|
| **CconnexionMngt**(TConfigureObject * lnkConfigureObject)<br>Constructor : Create internal object , initialize to 0 variables. | |

| Method Summary | |
|---|---|
| int | **m_EventOnSocket**(void )<br>wake-up |
| bool | **m_IsConnectionAuthorized**(unsigned long IPAddress)<br>Return true if new connection is authorized |
| int | **m_ReceiveData**(TItemConnexion * lnkConnexion)<br>Interface with CTCPConnexion::write method for reading data from network return negative value if problem |
| bool | **m_Reset**(void )<br>Method for Resetting ConnectionMngt component return true if Reset |
| int | **m_SendData**(TItemConnexion * lnkConnexion)<br>Interface with CTCPConnexion::read method for sending data to the network Return negative value if problem |
| bool | **m_Start**(void )<br>Method for Starting ConnectionMngt component return true if Started |
| bool | **m_Stop**(void )<br>Method for Stopping component return true if Stopped |