ISTITUTO NAZIONALE DI RICERCA METROLOGICA
Repository Istituzionale

Flexible Arduino Board - Mid-Tier and Application Software

Original
Flexible Arduino Board - Mid-Tier and Application Software / Francese, Claudio. - (2017).

Availability:
This version is available at: 11696/75184 since: 2023-01-12T14:44:53Z

Publisher:

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

07 May 2024

Claudio Francese

Flexible Arduino Board - Mid-Tier and Application Software

T.R. 26/2017 13/12/2017

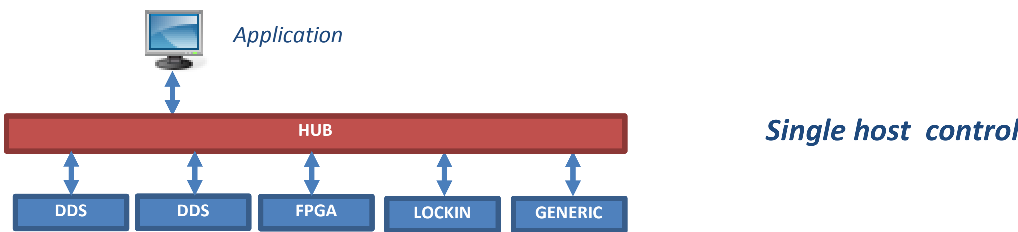
I.N.RI.M. TECHNICAL REPORT

Contents

Abstract 3
Software requirements 4
Software organization in layers 4
Application Layer 5
Communication protocol 5
Register organization of the Software 6
Mapping the connected devices 7
Mid-Tier Layer 9
Abstract data-channel 10
Implementation of the Mid-Tier Software 12
Modular structure of the Mid-Tier Layer Applications 13
The base Client and Server objects 14
Support classes 15
The Server class 16
The Client class 19
Concurrent resources access 20
The Mid-Tier Application class 21
Board simulator 21
Routing layer 23
Implemented Modules 27
Serial port 27
TCP socket 28
HTTP 29
Appendix 33
Internetworking 33
Possible Instruments Abstraction in Python 34
Software-defined instruments 37
Bibliography 38

Abstract

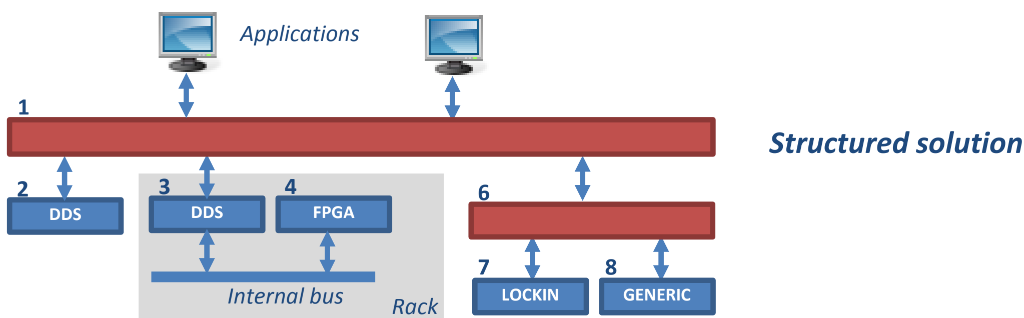
An instrument board based on Arduino™ has been designed and the firmware is described in Technical Reports (1) and (2). Although a direct connection from the controlling computer to one or more boards can be used, this solution does not allow to implement complex experimental configurations and only one computer per time can control the boards. Another issue of this approach is the low scalability of the controlling software, usually tied to a given setup.



Single host control

This report describes the study, design and implementation of a "Structured solution" for the Application software with the following goals in mind

- Connecting the boards to form a network of distributed resources
- Allowing concurrent access to the resources from many applications
- Granting access from dedicated hardware or from computers with different operating systems
- Improve flexibility and future integration with other instruments



Structured solution

The above requirements took to divide the software in functional layers and defining a communication protocol among them. The Python 2.7 language has been widely used as it provides a good prototyping environment and offers a great number of ready-to-use support libraries as well.

The results of this work provide a working environment for the interconnected boards and a starting point for further developments and investigations on additional features and real-time solutions also in other programming languages.

Software requirements

The aim of the design is to achieve a separation of the software functions into layers, with an increasing abstraction degree which allows the end-user of the boards to implement his/her software or to operate the boards without a deep knowledge of the internal details of the hardware and firmware.

The main requirements for the software are

- Easy access of the boards from different operating systems and programming languages.
- Support for distributed operation.
- Support for concurrent operations.
- Support for future features.

Software organization in layers

The requirements led to organize the overall software in three levels with a decreasing abstraction degree

1. Application Layer software and user interface.
2. Mid-Tier Layer software which acts as an interface between Layer 1 and Layer 3.
3. Board Layer software – this software, also called *firmware*, has already been discussed in Technical Reports (1) and (2). It will not be discussed further in this report.

The Application Layer allows the end-user to interact with the boards, both with a Graphical User Interface (GUI) or with a custom software. As many type of boards and connection types could be present in a real system, the application software does not connect directly to them and an intermediate software layer (Mid-Tier Layer) is present. This layer acts as a gateway between the boards and the applications and provides a unified communication protocol, which simplifies the applications development as well.

Figure 1 shows the overall architecture of the software.

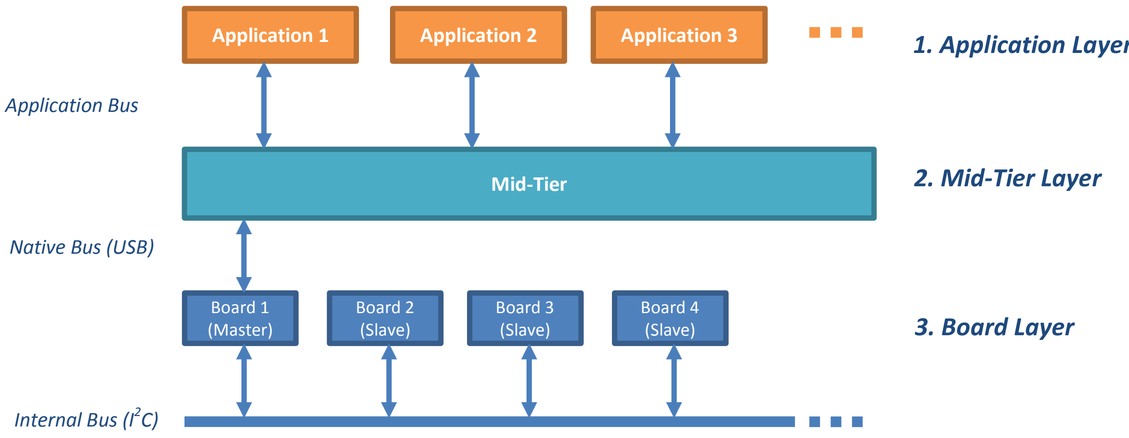


FIGURE 1 -GENERAL ARCHITECTURE OF THE SOFTWARE

Application Layer

The application layer provides the user interface and allows to operate both the boards and the Mid-Tier Software. This report describes the communication protocol and the register organization of the data providing some application examples. As the internal details of the graphical programming techniques are of no interest with respect to the overall software design, they will not be covered by this report.

Communication protocol

The chosen communication protocol for the application software is the same board communication protocol already defined in (1). The Mid-Tier Layer will use the same protocol as well. This choices satisfy the requirement for the application software to operate a board with or without the presence of an intermediate (Mid-Tier Layer) software.

The set of commands used by the application is shown in Table 1.

Command	Description	Implementation
P	Request communication protocol	Execute locally on the connected node
?	Who	Execute locally on the connected node
??	List	Execute locally on the connected node
F	~~Forward~~	~~Execute locally on the connected node~~ **Superseded by "/" command**
*	System	Execute locally on the connected node
I	Identify	Execute locally on the connected node
A	Acknowledge	Execute locally on the connected node
R	Read register	Execute locally on the connected node
-	Reply	Execute locally on the connected node
#	Remarks	Execute locally on the connected node
W	Write register	Execute locally on the connected node
/	Address to another node Syntax: /ID1/ID2/… `cmd` Execute `cmd` on node ID2 along path ID1 - ID2 - …	Forward the command for execution on the specified node

TABLE 1 - APPLICATION COMMANDS SET

By convention, the applications expect the first connected device (both a Mid-Tier Layer Node or a directly connected board) to respond to the requests with the above protocol.

When the connection with the node is established, the application requests the connected device to identify itself without ambiguity and to provide information about the type of device it is (i.e. an ordinary board or a Mid-Tier Node).

This operation is performed by reading the register #2 (REG_FW_DRIVER) of the attached board or node. Again, this register is one of the defined standard registers which must be implemented both in an ordinary board or a "software device" (for example a Mid-Tier Layer Node) as shown in Table 2.

Figure 2 shows an example of two applications, each requesting the connected device to identify itself and to provide the board type information. In the figure, the green numbers above each device are the IDs of the nodes.

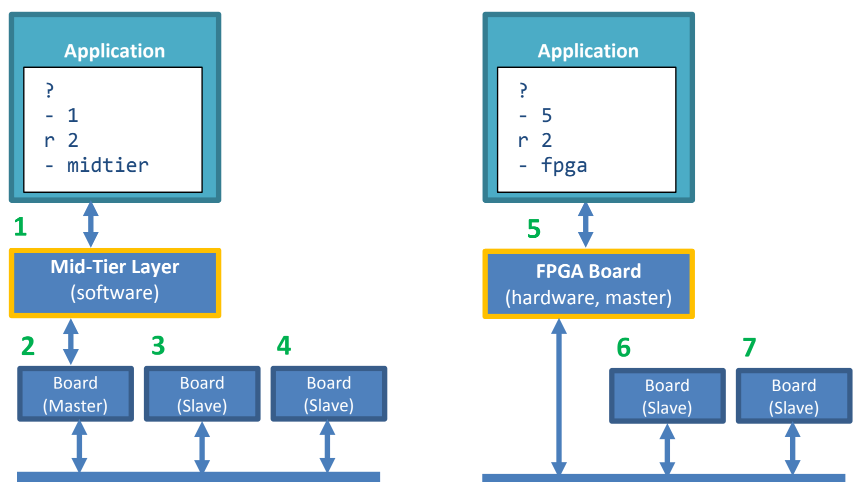


FIGURE 2 - APPLICATION COMMUNICATION EXAMPLE

Register organization of the Software

The base register set which the Application Layer expect to access is shown in Table 2.

Register ID	Register Name	Type	Description
1	NODE_ID	Text	ID of the Node
2	SW_DRIVER	Text	Name of driver class
3	SW_NAME	Text	Software name
4	SW_VER	Text	Software version
5	SW_BUILD	Text	Software build date
14	DBG_LASTBOOT	Int 32	Last node restart in milliseconds
18	PARAM_STATE	Int 32	Parameters' state, used to detect changes
20	NODE_NAME	Text	Node name

TABLE 2 - REGISTER SET OF THE MID-TIER LAYER

The above registers are a subset of those defined for a generic Arduino-based board in (1) and allow the software to start the base communication with any type of device.

An example of access to an unknown-type board is given in the following section.

Mapping the connected devices

When more than one device are interconnected to form a network, a mapping algorithm must be implemented to provide the topology of the network. Thus, at start-up, the application software maps the connected devices starting from the first connected node (both a physical board or a software node).

A recursive algorithm can be used to enter the node, list its children and for every found child apply the algorithm again. An implementation example in Python language follows.

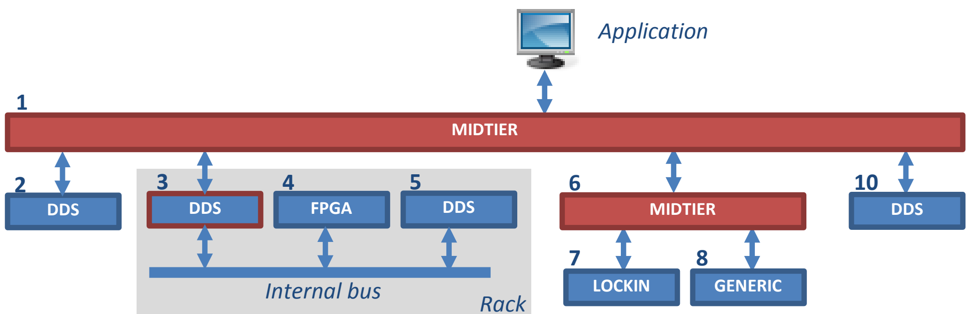
```
def Command(command, address=""):                       # SEND A COMMAND TO AN ADDRESSED NODE, RETURN THE REPLY
    if address != "":
        command = address + " " + command
    return TxRx(command)[1:].lstrip()

def Scan(address):
    tmp = [address]                                     # INCLUDE THE SCANNED NODE IN THE RESULTS
    for ID in Command("??", address).split():           # SCAN EVERY NODE ATTACHED TO THE CURRENT NODE
        tmp = tmp + Scan(address+"/"+str(ID))
    return tmp
```

The function `TxRx(command)` sends a command to the first attached node and returns the reply. The function call `Command("??", address).split()`is used to retrieve the IDs of the children attached to the addressed node as a list.

After the scan has been performed, the information on the nodes can be obtained by querying every node using its address, for example a read access to its register 2 to request the node type as in the example below

```
nodes = Scan("")     # START FROM THE FIRST NODE ATTACHED TO THE APPLICATION
print "Connected with ID:", Command("?"),"Type:", Command("r 2")
for address in sorted(nodes):
    if address != "":
        print address, "\t", Command("r 2", address)
```

An example is provided for this simple network



When the scan starts from node 1, the following messages are exchanged. The found children of the node are returned as a list of their IDs and if a node has no children its reply is empty ("- " reply string).

```
??
- 10    2       3       6
```

In the first step shown above, nodes 4, 5,7 and 8 are not listed because they are not directly connected to

node 1. Instead, access is provided through the rack interface at node 3 and the Mid-Tier node 6. When the scan process continues, the remaining nodes of the network are recursively explored.

```
/10 ??
-
/2 ??
-
/3 ??
- 4     5
/3/4 ??
-
/3/5 ??
-
/6 ??
- 7     8
/6/7 ??
-
/6/8 ??
-
```

At this point the list of nodes is used to request the type of every node

```
Connected with ID: 1 Type: midtier
/10     dds
/2      dds
/3      dds
/3/4    fpga
/3/5    dds
/6      midtier
/6/7    lockin
/6/8    genericboard
```

A textual representation of data shows the original topology of the network

```
1       midtier
        2       dds
        3       dds
                4       fpga
                5       dds
        6       midtier
                7       lockin
                8       genericboard
        10      dds
```

The messages exchanged with the application during the board-type identification are shown below for reference

```
?
- 1
r 2
- midtier
/10 r 2
- dds
/2 r 2
- dds
/3 r 2
- dds
/3/4 r 2
- fpga
/3/5 r 2
- dds
/6 r 2
- midtier
/6/7 r 2
- lockin
/6/8 r 2
- genericboard
```

An alternative scan method is provided by the Mid-Tier Layer and is described in section “Board simulator” at page 21.

Mid-Tier Layer

The core of the software is the Mid-Tier Layer because it abstracts the behaviour of the boards and allows to add functions to the system of boards which would be difficult to achieve. The Mid-Tier Layer is seen by the connected Application Software as an ordinary board, so it must implement the same communication protocol described at pages 5 and 6. In this phase of the design, this software layer is intended to be installed and run on medium power computers, like ordinary PCs or single board computers, e.g. RaspberryPi, BeagleBoard and similar devices as shown in Figure 3-a.

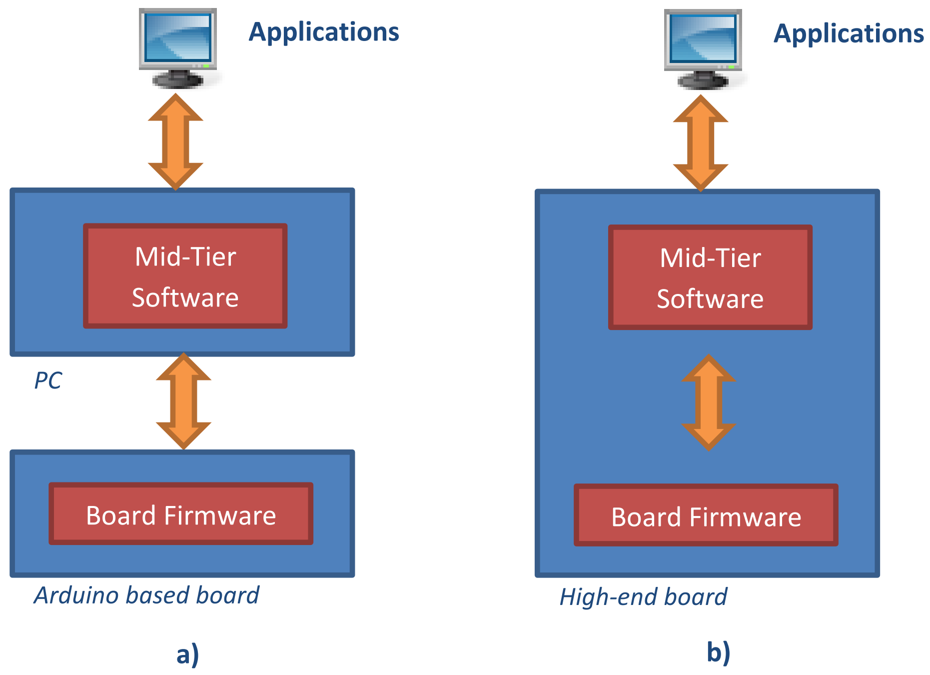


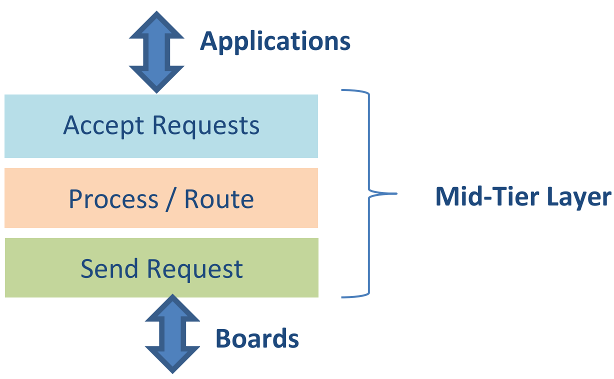
FIGURE 3 – ALTERNATIVE SOLUTIONS FOR BOARD FIRMWARE AND MID-TIER LAYER

When the board provides more computing resources, e.g. a FPGA+CPU board or single-board PC connecting to a dedicated hardware, the scheme in Figure 3-b could be used. In this case, the Mid-Tier Layer runs directly inside the physical instrument board and is part of its firmware.

Independently from the choice between any of the two options, the architecture of the Mid-Tier Layer is the same and must provide three layers of functions:

- Accept requests → act as a Slave / Server
- Process or Route the request → act as a Router
- Send the Requests → act as a Master / Client

The Server component of the Mid-Tier accepts requests from other nodes whatever type they are (for example applications or other Mod-Tier Nodes). After the request has been received, some processing is performed on it (typically to decide if the requested operation can be executed locally by the node itself or if it must be forwarded to another node). At the end of the request-processing chain, a reply is sent back to the node which originated the request. The Client component of the software is complementary to the Server component and allows to generate requests to another node and wait for replies from it.



The current implementation of the Client side provides two connection types

- Serial communication, typically used to control the physical Boards
- Network communication to forward messages to other Mid-Tier software Nodes or advanced boards which will be developed in future

If needed, future expansion of the software will implement other communication protocols not yet specified at the time of writing this report.

The Server component currently provides two connection types

- Serial communication to allow legacy and/or simple applications to send messages to the software
- Network communication to handle messages coming from the Application Layer Software or from another Mid-Tier Layer Node

The current network protocols are

- HTTP: the Mid-Tier is presented as a web server and data can be exchanged through an HTTP GET request and a following reply. This protocol on one hand adds a lot of overhead to the exchanged data thus slowing the data transfer but on the other one eases external access in complex network infrastructures (i.e. firewalls or filtered TCP ports). HTTP is also a good cross-platform protocol and is native in many different programming languages or can easily implemented, thus allows portability of the applications.
- Raw socket, used to exchange data with a higher throughput when the network infrastructure allows it (i.e. local subnets or open TCP ports to the Internet)

Abstract data-channel

Both the serial and the network protocol used by the Mid-Tier Layer provide a transparent mechanism which hides the actual physical implementation of the channel to the surrounding software. The aim of this abstraction is to simplify the development of the software only focusing on its overall logic.

The message which is exchanged, is actually embedded "as is" inside the desired transport protocol shown in Figure 4. Data are sent onto the resulting "abstract data-channel" from the source to the destination and the application does not deal with the physical channel.

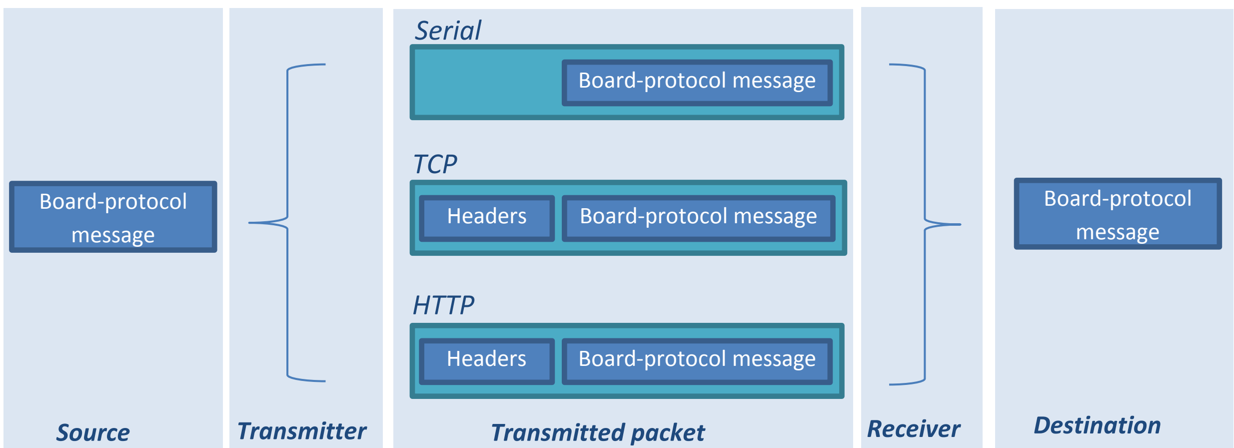


FIGURE 4 - MESSAGE EMBEDDED IN THE TRANSPORT PROTOCOL

The abstract data-channel also allows to implement complex and distributed topologies as shown in Figure 5 where instruments installed in two geographically separated locations can be operated through TCP connections of multiple levels of Mid-Tier Nodes. The example also shows a legacy application (for example running a low-end microcontroller board) in Lab4 driving some boards through a serial connection.

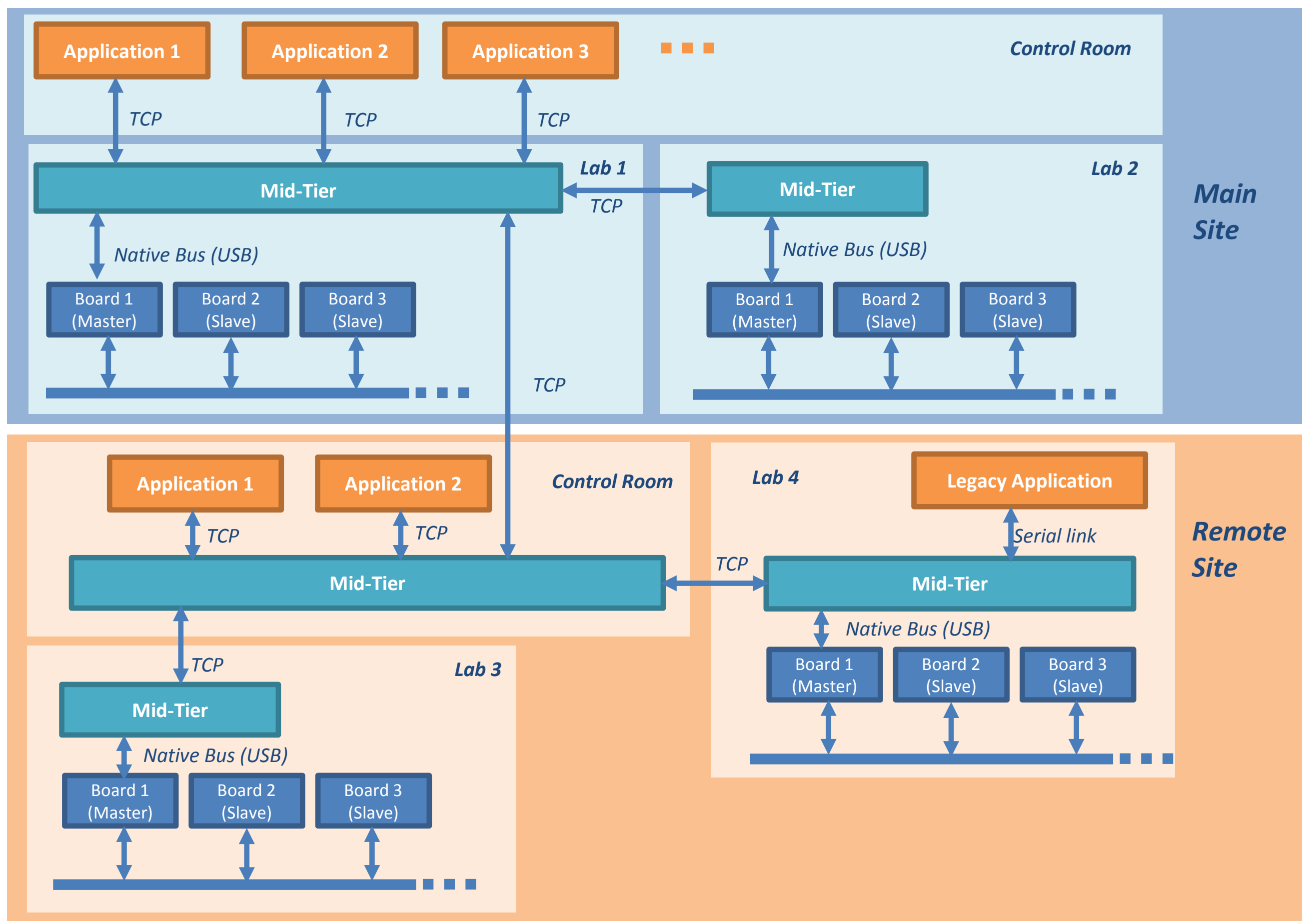


FIGURE 5 - MID-TIER SOFTWARE IN A DISTRIBUTED SETUP

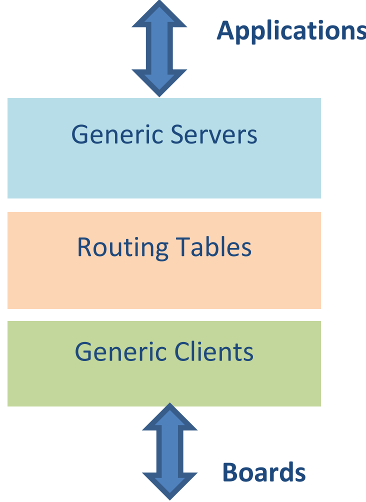
As shown in Figure 5, the described network is actually a tree. Although more instances of the Mid-Tier software could run on a same network node to create more complex architectures (for example with redundant connecting paths or nodes acting simultaneously as masters and slaves), attention must be paid to avoid critical situations like loops which could take to infinite data retransmissions along them.

The current implementation of the Mid-Tier Layer software does not provide any mechanism to avoid loop recognition and consequences of circular data paths.

Implementation of the Mid-Tier Software

The software has been written in Python 2.7.3-32bit language and tested in Microsoft Windows 7 64-bit.

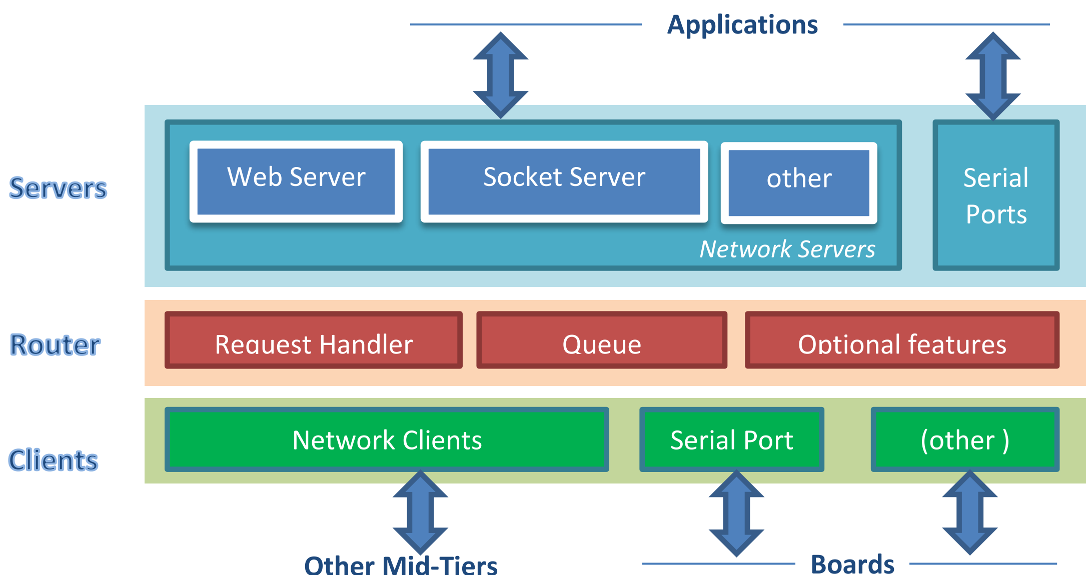
The advantages offered by the language in terms of cross-platform portability, object oriented paradigm (OOP) nature and ready-to-use libraries compensate the performance which could be a little bit worse compared to a native application compiled for a fixed architecture and operating system.



The three components which the software is divided into are actually implemented as Python Objects as described in the following section.

Modular structure of the Mid-Tier Layer Applications

A base structure has been defined, implementing a pool of generic servers, clients and some interconnecting routing tables. Each Server, Routing Table, and the Mid-Tier Application itself are instances of proper classes. OOP allows to describe the base behaviour and specialize it later by class derivation. The interconnection capabilities of the software can be easily extended further in future.



The resulting high-level applications which use the described approach are quite simple. For example, the following Python code describes a node with ID equal to 20 which accepts requests from a serial port, a tcp socket and http protocol. The node automatically forwards the requests to another http server (`srv01`) listening on TCP port 8020.

```
app = appclass.AppClass(
    ID = 20,
    Servers = [
                modules.serialmodule.Server("\\\\.\\com1", 115200),
                modules.tcpmodule.Server('0.0.0.0', 5000),
                modules.httpmodule.Server("127.0.0.1", 8080,"./htdocs/")
                ],
    Clients = [
               modules.httpmodule.Client("http://srv01:8020/") # CONNECT TO ANOTHER MID-TIER NODE
               ]
    )
```

The same access to a serial connected device, can be implemented by changing only one line of code in the "`Clients=`" section as in the following example.

```
app = appclass.AppClass(
    ID = 20,
    Servers = [
                modules.serialmodule.Server("\\\\.\\com1", 115200),
                modules.tcpmodule.Server('0.0.0.0', 5000),
                modules.httpmodule.Server("127.0.0.1", 8080,"./htdocs/")
                ],
    Clients = [
               modules.serialmodule.Client("com2", 115200) # CONNECT TO A DEVICE THROUGH THE SERIAL PORT
               ]
    )
```

The details of the Client and Server objects are discussed in section "Implemented Modules" at page 27 and following.

The base Client and Server objects

The objects are based on the two main base classes `ServerModule` , `ClientModule` and some support classes.

The minimal implementation of the Mid-Tier Software is graphically shown in Figure 6 and requires that two methods are declared respectively in the Server and Client objects

- The `Server.OnRequest(data)` method which is called at every request from the client.
  The parameter `data` passed to the function is the content of the request (for example “r 20” to read the register #20).
  The function **must** return a value as a result of the request which is sent back to the client.
- The `Client.TxRx(data)` method which is called to issue a command to the attached device. The function returns the reply from the device.

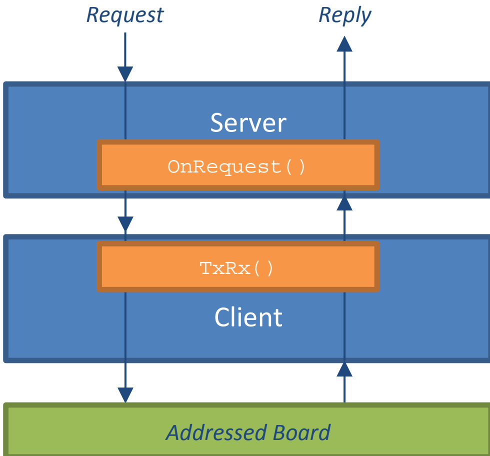


FIGURE 6 - STRUCTURE OF A SIMPLE MID-TIER APPLICATION

As an example, the implementation of the Mid-Tier software which accepts requests from a TCP socket on TPC port 5000 and is connected to a board through a serial connection on port COM37 is

```
server = tcpmodule.Server('0.0.0.0', 5000)
client = serialmodule.Client("\\\\.\\com37", 115200)
server.OnRequest = client.TxRx
```

The last line of code in the above example binds the action following a request to the actual communication with the board.

A more complete implementation of the application will slightly differ from the example because there will be more clients instances, one for each connected device, and the callback function associated to the request will implement a decision algorithm to choose the proper client object. Finally, once the object has been found, the corresponding `TxRx()` method will be called.

```
def route(data):
   # some search criteria in an array “clients” of client objects -> “index” is the index of the desired client
   return clients[index].TxRx(data)

server = tcpmodule.Server('0.0.0.0', 5000)
client = serialmodule.Client("\\\\.\\com37", 115200)
server.OnRequest = route
```

After having discussed the top-down structure of the software, the following sections will describe the classes involved in its implementation.

Support classes

The implemented communication is ASCII, line oriented. Two support classes have been designed to simplify the development of the software.

The class `SimpleReceiver` implements an object able to queue the received data (independently from the physical channel) and execute an action through a callback function when a line is complete. The support class is not intended to be used alone and a *parent* object must be specified in its constructor. The parameter links the class to the part of the code which actually has access to the physical communication channel. A requirement for the `SimpleReceiver` class to work is the implementation in the specified parent object of the callback function `OnLineReceived()` associated to a complete line reception. This function is discussed in the two sections relative to the Server and Client classes.

```
class SimpleReceiver(object):
    '''
    This class calls the OnLineReceived(data) function when a \n or \r terminated line is received
    Data are appended at each call of the HandleIncomingData(rxdata) method.
    '''
    def __init__(self, parent, Timeout=1.0):
        self.Timeout = Timeout
        self.NextTimeout = time.clock()
        self.parent = parent
        self.run = True
        self.startofline = True
        self.rxbuffer = ''

    def HandleIncomingData(self, rxdata):
        if len(rxdata)>0:
            rxdata = rxdata.replace('\r','\n')  # RECOGNIZE BOTH \n AND \r AS LINE TERMINATOR
            for rxchar in rxdata:
                if self.startofline:
                    if rxchar != '\n':
                        self.startofline = False
                        self.rxbuffer = rxchar
                else:
                    if rxchar != '\n':
                        self.rxbuffer = self.rxbuffer + rxchar
                    else:
                        self.parent.OnLineReceived(self.rxbuffer)
                        self.startofline = True
            self.NextTimeout = time.clock() + self.Timeout
```

The described class works in situations where data are immediately available, for example in a stream of generated requests inside a loop where the software can call the `HandleIncomingData` method at each iteration.

```
parentobject = ... # SOME OBJECT WICH EXECUTES REQUESTS
receiver = SimpleReceiver(parentobject)

while True:
        d =  ... # GENERATED REQUEST DATA
        receiver.HandleIncomingData(d)
```

Unfortunately, in many real situations data are not immediately available and no event or messaging subsystem exists to notify the software for new data available. In these cases, the function used to receive data from the physical link would block the main execution thread for long times.
For this reason, the second support class `ThreadedReceived` is provided. This class uses the multithreading capabilities of the Python language to implement an object able to call an external `RxFunction` (even blocking) in a separate thread and parse the resulting data through the `SimpleReceiver` object. Running the

receiving function in a separate thread does not block the main one and allows to call the function `HandleIncomingData()` as soon as data are available.

```
class ThreadedReceiver(SimpleReceiver):
    '''
    This class is the threaded extension of SimpleReceiver class. The thread is not automatically started.
    '''
    def __init__(self, RxFunction, OnLineReceived, Timeout=1.0):
        '''
        Constructor
        '''
        SimpleReceiver.__init__(self, OnLineReceived, Timeout)
        self.RxFunction = RxFunction
        self.rxthread = threading.Thread(target=self.rxthreadfunction)
        self.rxthread.daemon = True

    def rxthreadfunction(self):
        while self.run:
            rxdata = self.RxFunction()
            if len(rxdata)>0:
                self.HandleIncomingData(rxdata)
            else:
                time.sleep(0.01)    # IDLE
```

The Server class

A Server module waits for requests from a given physical channel. When a command is received, an action is triggered and finally a reply is sent back to the connected client through the same channel.

Abstracting the server's behaviour requires to implement a base class which will be extended later according to the requirements (e.g. given by the communication protocol or physical channel).

Thus the base class `ServerModule` for the server is very simple and provides the declaration of the required callback function `OnRequest()` and an auxiliary method `OnLineReceived()`.

```
class ServerModule(object):
    '''
    Base Communication Module (Server Side)
    '''
    def __init__(self, receiver=None):
        self.receiver = receiver
        self.OnRequest  = lambda x:x # CALLED AT EVERY REQUEST TO THE SERVER, SHOULD BE OVERRIDDEN
        self.OnLineReceived = lambda data:self.TxFunction(self.OnRequest(data))
```

The `OnLineReceived()` method has been introduced to solve a deep asymmetry issue present in the data exchange sequences in the Server and in the Client objects. Receiving a new line of data must take to different behaviours in the two classes so the actions are kept separate in their respective codes, while the receiving mechanism is left the same. Figure 7 and Figure 8 graphically show the different sequences in the two classes.

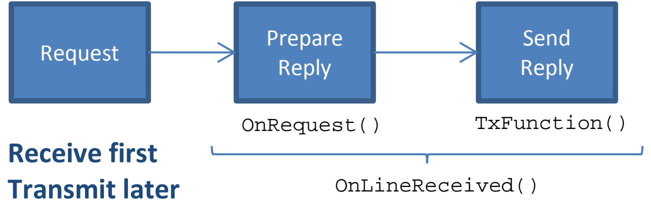


FIGURE 7 - ONLINERECEIVED FOR A SERVER

As the transmission of the reply message depends on the physical channel in use, the actual transmission is performed by a separate callback function `TxFunction()` which will be declared in the derived Server class.

It should also be noted that the added complexity of the object approach of this software design saves time when the high level application is developed. As a matter of fact every physical communication channel and every communication protocol has is its own behaviour and operation model, the aim of the design is to provide a common interface between the physical communication channel and the upper layer of the software: the `OnRequest()` method.

As an example, although both a Serial connection and a TCP socket are point to point connections, the respective support libraries provide quite different functions. The following table briefly shows a comparison of data receivers through the two cited links.

<table>
<tr><th>Reading data from a serial connection</th><th>Reading data from a TCP socket</th></tr>
<tr><td>Character / byte oriented
Polling needed</td><td>Inherently packet / line oriented
Threaded servers exist</td></tr>
<tr><th>Sample code</th><th>Sample code</th></tr>
<tr><td><pre>import serial
import time
ser = serial.Serial('COM3', 9600, timeout=0)

while True:
 try:
  print ser.readline()
  time.sleep(1)
 except ser.SerialTimeoutException:
  print('Data could not be read')</pre></td><td><pre>import SocketServer
class MyTCPHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        self.data = self.request.recv(1024).strip()
        print self.data
        self.request.sendall(self.data.upper())

s = SocketServer.TCPServer(("0.0.0.0", 5000), MyTCPHandler)
s.serve_forever()</pre></td></tr>
</table>

A simple example of the implementation of a server module is the Serial connection below.

```
class Server(base.ServerModule):
    def __init__(self, serialport, baudrate, wait=4.0):
        self.link = SerialLink(serialport, baudrate, wait)
        super(Server, self).__init__(base.ThreadedReceiver(self, Timeout=1.0))
        self.TxFunction = self.link.Tx
        self.RxFunction = self.link.Rx
        self.receiver.rxthread.start()
```

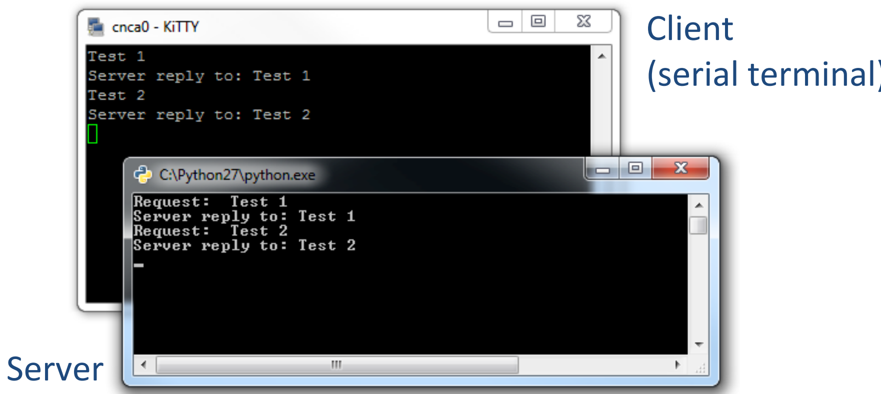
It should be noted that the `Server` class is derived from its base class and it also uses a dedicated `SerialLink` object which actually handles the serial link data transmission through two functions.

```
class SerialLink(object):
    def __init__(self, serialport, baudrate, wait):
        self.serialport = serialport
        self.baudrate = baudrate

        try:
            self.link = serial.Serial(port = self.serialport, baudrate=self.baudrate, timeout=0.05)
            time.sleep(wait)
        except:
            self.link = None

    def Rx(self):
        if self.link != None:
            return self.link.read()
        else:
            return ''

    def Tx(self, data):
        if self.link != None:
            self.link.write(data+"\n")
```

The Serial Server can be tested through the simple following script

```
from modules.serialmodule import Server

def handler(reqdata):
    print "Request: ", reqdata
    replydata = "Server reply to: " + reqdata
    return replydata

server = Server("\\\\.\\cncb0", 115200, wait = 0.1)
server.OnRequest = handler
```

and the operation result is shown below.



The Client class

Generally a client module for a communication channel is quite simple with respect to its Server counterpart because the client task is expected to be triggered by a software action which causes it to send a command through a physical channel to an external device, wait for a reply and finally pass the reply back to the software.

Again abstraction is needed in order to support different physical media or communication protocols and a base class is defined.

The requirement for the Client class is the implementation of the `TxRx()` method which will be called by the upper level software. Some auxiliary methods are defined to better integrate the class in the developed framework.

```
class ClientModule(object):
    '''
    Base Communication Module (Client Side)
    '''
    def __init__(self, receiver):
        self.OnRequest  = lambda x:x # CALLED AT EVERY REQUEST
        self.receiver = receiver
        self.atomicrequest = threading.Lock()
        super(ClientModule, self).__init__()

    def TxRx(self, command):
        self.reply = None
        with self.atomicrequest: # PROTECT CONCURRENT REQUESTS
            self.TxRxTransaction(command)

            if self.reply == None:
                return "- fail"
            else:
                return self.reply

    def TxRxTransaction(self, command):
        self.TxFunction(command)
        self.receiver.NextTimeout = time.clock() + self.receiver.Timeout

        while self.reply == None and time.clock() < self.receiver.NextTimeout:
            time.sleep(0.01)


    def OnLineReceived(self, data):
        self.reply = data
```

As for the Server class, also the `OnLineReceived()` must be implemented and it is used to stop the wait phase after a command transmission to the external device. This is shown in Figure 8 and it can be observed that the behaviour is deeply different with respect to the Server behaviour previously shown in Figure 7.

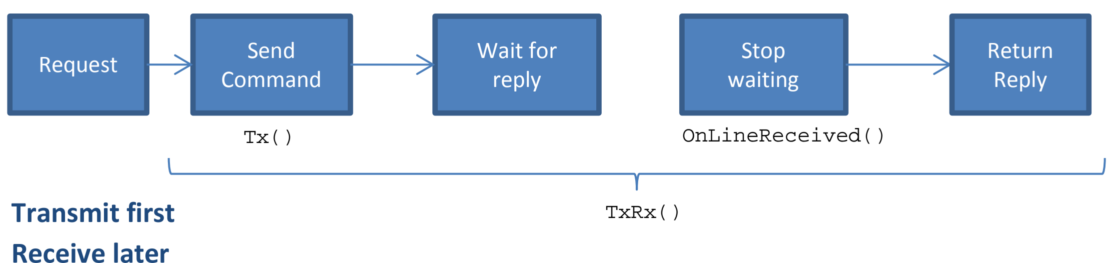


FIGURE 8 - ONLINERECEIVED FOR A CLIENT

Concurrent resources access

As in a complex application many servers could receive requests for the same client, some request-reply sequences could overlap thus giving erroneous results. For example Figure 9-a shows the threat of two potentially overlapping requests, the second coming before the transmission of the reply to the first request. Without any protection, the `TxRx()` method could erroneously return Reply2 to Request1 and Reply1 to Request2.

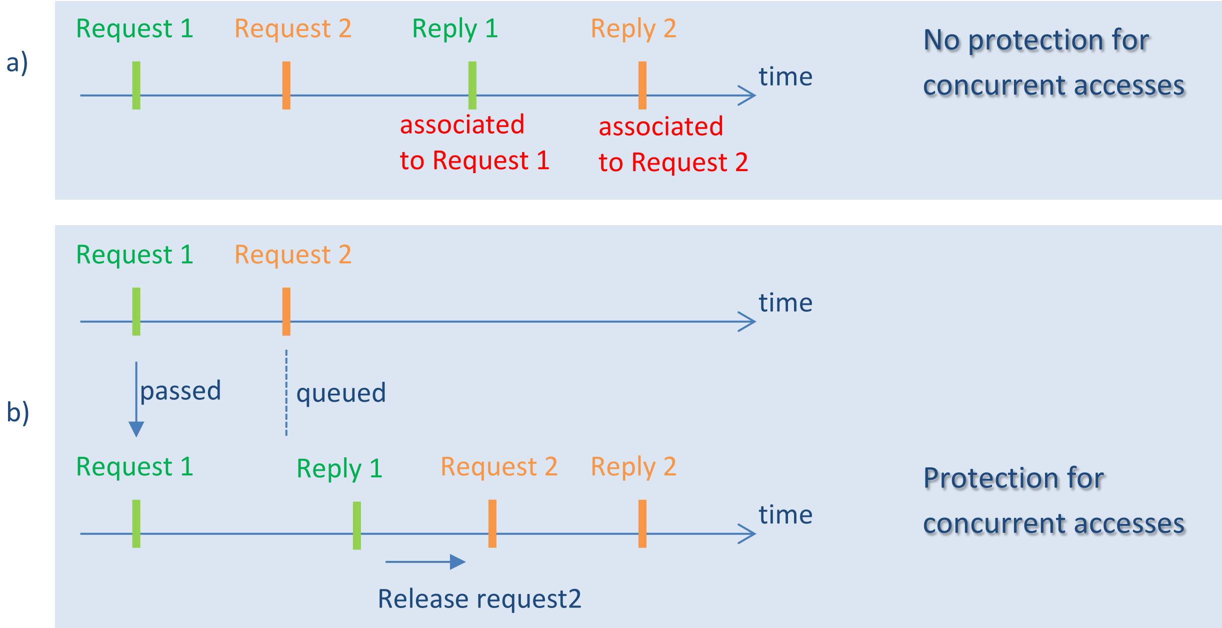


FIGURE 9 - CONCURRENT OPERATIONS PROTECTION

The introduced protection is implemented with a `Lock` object `atomicrequest` which prevents the execution of the same code during one call (server request 2) before the completion of the same part of code previously called by the server request 1.

The last auxiliary implemented method is `TxRxTransaction()` which is intended to be used in situations where a native function already exist to send data, wait and return a reply.

An example is the implemented http library to retrieve an URL from a web server. In this case the complete implementation of the Client for the HTTP protocol is

```
class Client(base.ClientModule):
    def __init__(self, baseurl):
        super(Client, self).__init__(None)
        self.baseurl = baseurl

    def TxRxTransaction(self, command):
        try:
            url = self.baseurl + "?" + urllib.quote(command)
            response = urllib2.urlopen(url)
            self.reply = response.read()
        except:
            self.reply = None
```

The Mid-Tier Application class

The Application class actually implements the Mid-Tier Software which becomes a node of the network of interconnected devices.

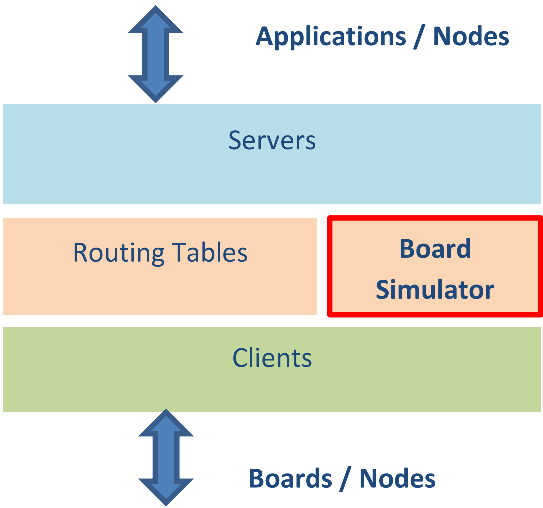
The Application is instantiated by specifying the ID of the Node, a list of Server objects (Serial, HTTP, TCP socket) and a list of Client object which can connect to hardware boards or other Nodes.

The Server and Clients lists define how the node is connected to the network.

```
class AppClass(object):
    '''
    Base Communication Application
    '''

    def __init__(self, ID, Servers=[], Clients={}):
        self.BoardMidTier = boardmidtier.BoardMidTier(self, ID)
        self.Servers = Servers
        self.Clients = Clients
```

Board simulator

As the application is presented as a register-organized board, a board simulator needs to be embedded into the software.



The simulated board is requested to

- Respond to an identification request: “?” command
- List all the attached devices (i.e. list the IDs of the connected boards and Nodes: “??” command
- Provide information about the firmware (version, type, driver, etc) as specified in protocol (1)
- Use the same register allocation convention of the Arduino base board as in (1)

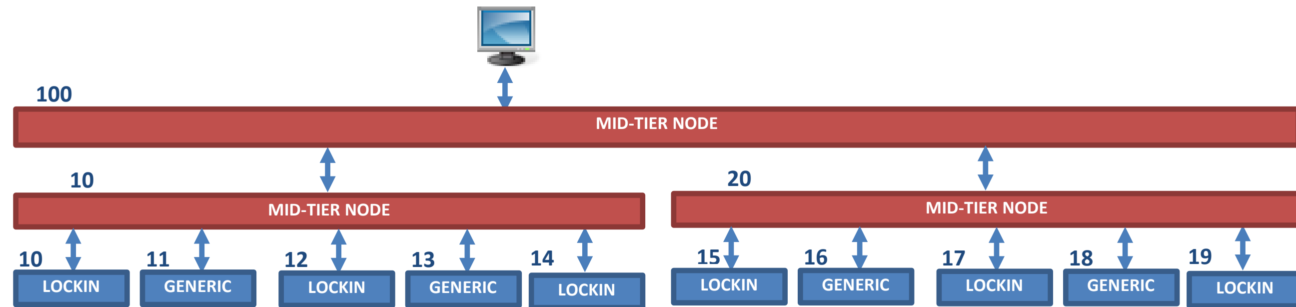
The above requirements permit to connect, address and properly operate the board in a network of many interconnected boards and nodes.

The implementation of the simulator is shown in Figure 10 and provides the minimal communication protocol, base register set and the functions to access the simulated resources.

```
import time, types

PROTO_WHO       = '?'
PROTO_LIST      = '??'
PROTO_GET       = 'r'
PROTO_SET       = 'w'
PROTO_SYS       = '*'
PROTO_REPLY     = '-'
PROTO_REMARKS   = '#'

STO_BOARD_ID            = 1   # ID of Board
REG_FW_DRIVER           = 2   # Name of driver class
REG_FW_NAME             = 3   # Firmware name
REG_FW_VER              = 4   # Firmware version
REG_FW_BUILD            = 5   # Firmware build date
DBG_LASTBOOT            = 14  # Program uptime
PARAM_STATE             = 18  # Parameter alteration counters
STO_BOARD_NAME          = 20  # Board name

class BoardMidTier(object): # GENERIC BOARD
    def __init__(self, parent, myid):
        self.regs = {}
        self.Tx = lambda x:None
        self.starttime = time.time()
        self.Clients  = {}
        self.regs[STO_BOARD_ID]      = str(myid)
        self.regs[REG_FW_DRIVER]     = 'midtier'
        self.regs[REG_FW_NAME]       = 'midtier.py'
        self.regs[REG_FW_VER]        = '1.0'
        self.regs[REG_FW_BUILD]      = time.strftime("%Y-%m-%d %H:%M:%S")
        self.regs[DBG_LASTBOOT]      = lambda:int((time.time()-self.starttime)*1000)
        self.regs[STO_BOARD_NAME]   = 'MidTier '+str(myid)

    def GET(self, reg):
        f = self.regs.get(int(reg),None)
        if f == None:
            return '- fail'
        elif type (f) is types.LambdaType:
            return "- " + str(f())
        else:
            return "- " + str(self.regs.get(int(reg)))

    def SET(self, reg, val):
        f = self.regs.get(int(reg),None)
        if f == None or type (f) is types.LambdaType:
            return '- fail'
        else:
            self.regs[int(reg)] = val
            return '- ok'

    def Execute(self, command):
        try:
            if command[0] == PROTO_GET:
                cmd, reg = command.split(" ")
                return self.GET(reg)
            elif command[0] == PROTO_SET:
                cmd, reg, val = command.split(" ",2)
                return self.SET(reg, val)
            elif command == PROTO_WHO:
                return self.GET(STO_BOARD_ID)
            elif command == PROTO_LIST:
                ret = '\t'.join(['-']+
                    [path[1:] for (path, handler) in self.Clients.items()]
                    )
                return ret
             else:
                return '- fail'
        except:
            return '- fail'
```

FIGURE 10 - BOARD SIMULATOR

The simulator has been tested using the simple three-level network graphically defined below



When the application is connected to node 100, its board simulator allows the execution of commands as an ordinary node.

```
?
- 100
??
-       20      10
/10 ??
-       14      13      12      11      10
/20 ??
-       17      16      15      19      18
```

Scan method

A simple implementation of a native network mapping function has been defined inside the board simulator class in a test release of the software

```
    def Scan(self, handler, address=""):
        tmp = [address]                            # INCLUDE THE SCANNED NODE IN THE RESULTS
        if address == "":
            command = "??"
        else:
            command = address + " ??"
        children = handler.TxRx(command)[1:].lstrip().split()
        for ID in children:    # SCAN EVERY NODE ATTACHED TO THE CURRENT NODE
            tmp = tmp + self.Scan(handler, address + "/"+ID)
        return tmp
```

The results, associated to the contents of a special register are shown.

```
- /20   /20/17  /20/16  /20/15  /20/19  /20/18  /10     /10/14  /10/13  /10/12  /10/11  /10/10
```

Although the function is very practical, it has been later removed from the current implementation because it took to timeout errors in big networks. In those situations, the time needed to scan all the network by distributing the operation across the network-nodes was too long with respect to the time the client was set out to wait for. A workaround consisting in asking in two separate phases the network scan operation and the results reading is under development.

Routing layer

The routing layer is quite simple as it relies of the `OnRequest()` method of the servers and a Python dictionary of Client objects, each element associated to the ID of the Node the client is connected to.

The dictionary is populated when the array of clients is passed to the constructor of the Application class. For each passed client, an attempt is done to connect to a device trough it. At this point, if the device responds with its ID, that ID is used as a key for that client instance and is stored in the dictionary. Otherwise the client is discarded. The Client member of the class is implemented as a property. Thus

assigning a value to it in the constructor or in any point of the software, actually triggers the actions described above.

```
    @property
    def Clients(self):
        return self.BoardMidTier.Clients

    @Clients.setter
    def Clients(self, ClientsList):
        self.BoardMidTier.Clients = {}
        for client in ClientsList:
            ID = client.TxRx("?")[2:]
            if ID != "fail":
                route = "/{ID}".format(ID=ID)
                self.BoardMidTier.Clients[route] = client
```

The keys of the clients dictionary are strings in the format "/ID" (i.e. "/1", "/2", …) and this convention is the most important part of the routing method in the network as described later.

On the other end, when the server list is passed, it is scanned. For each server instance, its callback function `OnRequest()` is associated to the common method `Route()` of the Application class. Again, the Servers member is implemented as a property.

```
    @property
    def Servers(self):
        return self._Servers

    @Servers.setter
    def Servers(self, value):
        self._Servers = value
        for server in self._Servers:
            server.OnRequest = self.Route
```

Finally, the actual routing is performed by the `Route(command)` method with the following algorithm

- When a request (command) starts with "/", routing must be performed, otherwise the command must be executed locally by the Board Simulator.
- If routing is requested, the message is split into two components:
    - The Address, starting from the first "/" character up to the second (excluded)
    - The Message, which is the remaining part of the command
- The Address is the key for the clients dictionary used to select the proper client
- The Message is transmitted using the selected client

```
    def Route(self, command):
        if command.startswith("/"):                      # ROUTING REQUESTED
            path, command  = command.split(" ",1)        # NOW command IS THE MESSAGE COMPONENT
            fulldestination = path.split("/",2)
            destination = "/"+fulldestination[1]         # ADDRESS COMPONENT IN THE FORMAT "/ID"
            if len(fulldestination) > 2:
                command = "/"+fulldestination[2]+" "+command
            if destination in self.BoardMidTier.Clients.keys(): # A CLIENT WHICH CAN SEND TO THE ADDRESS EXISTS
                return  self.BoardMidTier.Clients[destination].TxRx(command) # TRANSMIT MESSAGE, RETURN REPLY
        else:
            return self.BoardMidTier.Execute(command)    # LOCALLY EXECUTE THROUGH THE SIMULATOR, RETURN REPLY
        return "- fail"                                  # IF SOMETHING WENT WRONG RETURN AN ERROR
```

The side effect of this algorithm is that the Address component of the command is dropped at every routing step. This inherently provides the mechanism to perform the routing in a multi-level structure as

shown in Figure 11 where the first Node with ID 1 receives a message for the Board with ID 3. The Node 1 has a client associated to the key “/2” which can send data to Node 2. When data is routed from Node 1 to Node 2, Address component is dropped and Node 2 receives “/3”. Again the Address component “/3” is dropped in Node 2 and Board 3 receives the command in a form which is ready for execution. The reply from Board 3 is back-propagated to the Client which originated the request though a chain of returned replies at the end of the `TxRx()` function which is executed in each node.

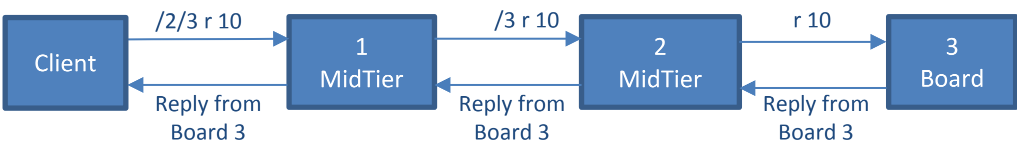


FIGURE 11 - MULTILEVEL ROUTING

The Application objects actually are event triggered through the callback functions defined in the Server objects which usually run in separate threads. So, the application inherently allows to define many levels of nodes which run in parallel. This is useful to test the software and to implement simulators of other nodes for special purposes.

The sample script below simulates a small network of two interconnected Mid-Tier Nodes and a real Arduino-based DDS board. The structure of the small network and the output are shown in Figure 12.

```
import modules

Node10 = modules.appclass.AppClass(
    ID = 10,
    Servers = [modules.httpmodule.Server("127.0.0.1", 8010,"./htdocs/")],
    Clients = [modules.serialmodule.Client("\\\\.\\com37", 115200)]
    )

client = modules.httpmodule.Client("http://127.0.0.1:8010/node/")
print client.TxRx("?")


Node20 = modules.appclass.AppClass(
    ID = 20,
    Servers = [
                modules.serialmodule.Server("\\\\.\\cncb0", 115200),
                modules.tcpmodule.Server('0.0.0.0', 5000),
                modules.httpmodule.Server("127.0.0.1", 8020,"./htdocs/")
                ],
    Clients = [modules.httpmodule.Client("http://127.0.0.1:8010/node/")]
    )


print ":: Node 10 ::"
print Node10.Clients
print ":: Node 20 ::"
print Node20.Clients

while True:
        pass
```

In the same figure, the results of accesses through TCP, Serial connection and HTTP are shown as well.

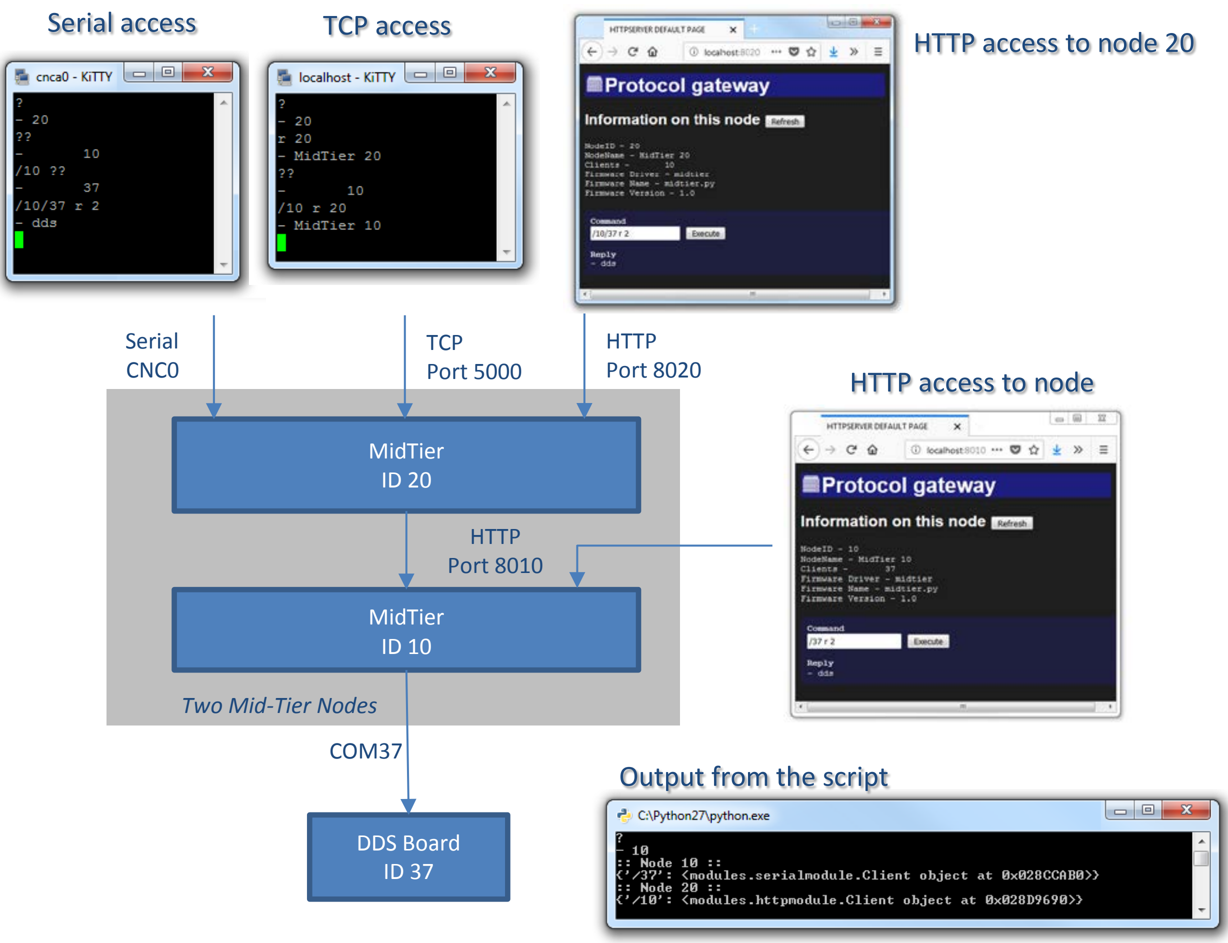


FIGURE 12 - SMALL NETWORK EXAMPLE

Implemented Modules

Currently, three modules have been implemented:

- Serial port connection
- TCP socket
- HTTP request/reply (web server and client)

Serial port

The serial port module is very simple and is described as an implementation example in "The Server class" section at page 16.

Below, the full code of both the Client and Server sides of the module

```
'''
@author: Claudio Francese
'''
import time, threading, base, serial

class SerialLink(object):
    def __init__(self, serialport, baudrate, wait):
        self.serialport = serialport
        self.baudrate = baudrate

        try:
            self.link = serial.Serial(port = self.serialport, baudrate=self.baudrate, timeout=0.05)
            time.sleep(wait)
        except:
            self.link = None

    def Rx(self):
        if self.link != None:
            return self.link.read()
        else:
            return ''

    def Tx(self, data):
        if self.link != None:
            print data
            self.link.write(data+"\n")


class Client(base.ClientModule):
    def __init__(self, serialport, baudrate, wait=4.0):
        self.link = SerialLink(serialport, baudrate, wait)
        super(Client, self).__init__(base.ThreadedReceiver(self, Timeout=1.0))
        self.TxFunction = self.link.Tx
        self.RxFunction = self.link.Rx
        self.receiver.rxthread.start()

class Server(base.ServerModule):
    def __init__(self, serialport, baudrate, wait=4.0):
        self.link = SerialLink(serialport, baudrate, wait)
        super(Server, self).__init__(base.ThreadedReceiver(self, Timeout=1.0))
        self.TxFunction = self.link.Tx
        self.RxFunction = self.link.Rx
        self.receiver.rxthread.start()
```

TCP socket

The Server side of the TCP socket module is based on a previously implemented multithreaded socket server (`TCPServer`). The actual Server Module uses an instance of the `TCPServer` as the communication link. The implementation of the Client side of the module is simple and consists in a socket which is created when a command is sent to the server and closed upon reception of the reply. The receiving function is run in a thread. An alternative implementation on a reliable network could keep the socket open even after receiving the reply, thus speeding-up the overall performance at the price of a permanent allocated resource and the risk of possible faults of the software in case of link drops.

```
'''
@author: claudio
'''
import time, threading, base, socket, SocketServer

class TCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):
        def OnLineReceived(self, data):
            self.request.send(self.server.parent.OnRequest(data)+"\n")

        def handle(self):
            cur_thread = threading.current_thread()            # CLIENT RUNS IN A SEPARATE THREAD
            self.server.clients[cur_thread.name] = self.request # UPDATE DICTIONARY OF CONNECTED CLIENTS
            receiver = base.SimpleReceiver(self)

            while True:
                try:
                    receiver.HandleIncomingData(self.request.recv(1024))    # RX AND PROCESS DATA FROM CLIENT
                except socket.error:                             # CLIENT DISCONNECTION
                    break
            self.server.clients.pop(cur_thread.name)            # REMOVE DISCONNECTED CLIENT FROM DICTIONARY

    def __init__(self, HOST, TCPPort=5000, Data=None):
        SocketServer.TCPServer.__init__(self, (HOST, int(TCPPort)), TCPServer.ThreadedTCPRequestHandler )
        self.clients = {}
        server_thread = threading.Thread(target=self.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        self.run = True

class Server(base.ServerModule):
    def __init__(self, ipaddress, tcpport):
        self.link = TCPServer(HOST=ipaddress, TCPPort=tcpport)
        self.link.parent =self
        super(Server, self).__init__(self.link)

class Client(base.ClientModule):
    def __init__(self, ipaddress, tcpport, timeout=1.0):
        self.ipadddress = ipaddress
        self.tcpport  = tcpport
        self.timeout  = timeout
        self.receiver = base.SimpleReceiver(self, timeout)
        super(Client, self).__init__(self.receiver)

    def TxFunction(self, data):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.settimeout(self.timeout)
        self.socket.connect((self.ipadddress, self.tcpport))
        self.socket.sendall(data+"\n")
        self.reply = None
        self.rxthread = threading.Thread(target = self.RxThreadFunction)
        self.rxthread.start()

    def RxThreadFunction(self):
        try:
            self.reply = self.socket.recv(1024).strip()
        except socket.timeout:
            self.reply = None
        finally:
            self.socket.close()
```

HTTP

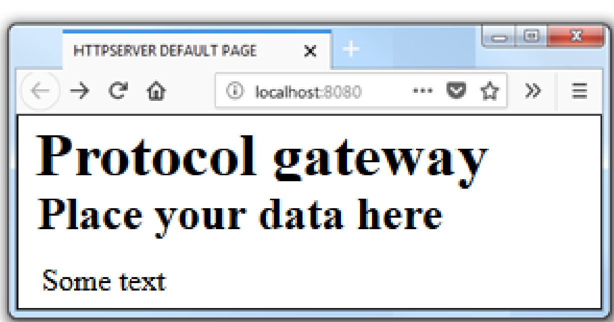
The HTTP Server module is the most complex part of the mid-tier software. The aim of the design is to allow the integration of the module in the common Server Module framework and to provide the capability to respond to some preformatted URLs.

As the Server module replies to GET queries as a web server, a basic file serving mechanism has been implemented for URLs of type ***http://server:port/htdocs/****.../.../filename.extension*, where *port* is the TCP port the server is listening to.

The server should also serve a special file (the default file) when no path and file are specified. Thus, requesting the URL ***http://server:port***, requests the default file *index.html* to be returned.

This behaviour allows to query the server module through a common web-browser. The returned data are specified in HTML format. For example, the following index.html could be used to show some information to the user.

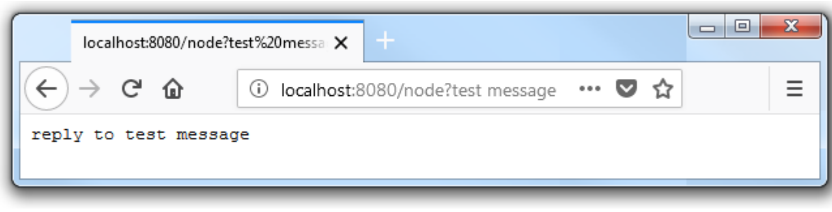
```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>HTTPSERVER DEFAULT PAGE</title>
</head>
<body>
<h1>Protocol gateway</h1>
<h2>Place your data here</h2>
<div>
Some text
</div>
</body>
</html>
```



The last, and more important URL which **must** be served has the format **http://server:port/node/?command** and is used to specify the command which the Mid-Tier software should process. Processing the URL requests the server to execute its `OnRequest()` method and the return value of that function is sent back to the requesting client reply.

The following example shows a simple data handler and how the server behaves when the URL http://localhost:8080/node?test%20message is accessed through a web-browser.

```
import modules.httpmodule as http

def callback(data):
    return "reply to " + data

server = http.Server("127.0.0.1",
        8080, "./htdocs/")
server.OnRequest = callback

while True:
        pass
```



Before describing in details the implementation of the HTTP Server Module, a last example of a default HTML page is given for reference.

The server allows to serve any type of file, so Stylesheet and Javascript files can be served as well. This allows, for example, to provide an HTML simple yet effective user interface to the software.

The following example shows how visiting the default page automatically requests the web-browser to fetch data from the server with some AJAX calls. The page contains a form with a text field which can be used to send requests to the server, thus providing a user interface. The resulting page is formatted with a stylesheet as well.

```
<!DOCTYPE html>
<html><head>
<meta charset="ISO-8859-1">
<title>HTTPSERVER DEFAULT PAGE</title>
<link href="/htdocs/styles.css" rel="stylesheet" />
<script src="/htdocs/jquery.min.js"></script>
<script src="/htdocs/he.js"></script>
</head>
<body>
<h1><img src="/htdocs/server.png">Protocol gateway</h1>
<div>
<h2>Information on this node <button onclick="RefreshAllInfo();return false;">Refresh</button></h2>
<div id="info"></div>
</div>
<script>
function NodeInfo(label, command) {
         $.ajax({
             async: false,
             type: 'GET',
             url: '/node?'+command,
             success: function(result){
                  $("#info").append(label + " "+ he.encode(result)+"<br>");
                  }
             }
         ); }

function Command(command) {
         $.ajax({
             async: false, type: 'GET',
             url: '/node?'+document.getElementById("frm1").elements[0].value,
             success: function(result){
                  $("#reply").text(he.encode(result));
                  }
             }
         ); }

function RefreshAllInfo() {
         $("#info").text("");
         NodeInfo("NodeID", "?");
         NodeInfo("NodeName", "r 20");
         NodeInfo("Clients", "??");
         NodeInfo("Firmware Driver", "r 2");
         NodeInfo("Firmware Name", "r 3");
         NodeInfo("Firmware Version", "r 4");
         }

RefreshAllInfo();
</script>
<br>
<form id="frm1" action="/node/">
         <b>Command</b> <br>
         <input type="text" name="fname" value="?">
         <button onclick="Command(); return false;">Execute</button><br><br>
         <b>Reply</b><div id="reply"></div>
</form>
</body></html>
```

As the code shows, the node data are fetched trough the JavaScript function `NodeInfo()` which sends the register-read request to the server. The command execution is requested through a call to the JavaScript `Command()` function which sends the request to the server and writes the reply in the proper place (the *<DIV id=”reply”>* tag).

The resulting behaviour of the described html file has already been shown in a previous section for other purposes and is reported in the figure aside (taken from Figure 12 - Small Network Example).

The HTTP Server module is based on a previously developed multithreaded HTTP server. For simplicity, the implementation of the server module is split into two sections, the Server Module object and the multithreaded server.

The Server Module instantiates the multithreaded http server object and sets the properties which define the actions which must be performed when a given URL is requested as shown in the table below

Server method	*Associated URL*	Action
handler_default	*http://server:port*	Sends the file index.html
handler_htdocs	*http://server:port/**htdocs**/.../file*	Sends the specified file
handler_node	*http://server:port/**node**?command*	Sends the data generated by OnRequest() method

The handler_node function, calls the support function `NodeCallback()` which simply assembles the requested command data form the URL components. The resulting code is shown below.

```
class Server(base.ServerModule):
    def __init__(self, ipaddress, port, htdocs):
        def HandlerWrapper(*args):
            GetHandler(self, *args)

        super(Server, self).__init__()
        self.ip = ipaddress
        self.port = port
        self.htdocs = htdocs + ('' if htdocs[-1] == '/' else '/')
        self.server = HTTPServer((self.ip, self.port), HandlerWrapper)
        self.thread = threading.Thread(target = self.server.serve_forever)
        self.thread.daemon = True

        self.handler_default    = lambda handler : handler.SendFile('index.html')
        self.handler_htdocs     = lambda handler : handler.SendFile("/".join(handler.pathlist[1:]))
        self.handler_node       = lambda handler : handler.SendData( self.OnRequest(self.NodeCallback(handler)) )
        self.thread.start()

    def NodeCallback(self, handler):
        cmd = urllib.unquote(handler.query)
        if len(handler.pathlist)>1:
            cmd = "/"+"/".join(handler.pathlist[1:])+" "+cmd
        return cmd
```

The other part of the module is the multithreaded http server which relies on the library `BaseHTTPServer.HTTPServer` and `BaseHTTPServer.BaseHTTPRequestHandler` classes.

The description of the internals of the library objects is beyond the scope of this report and further information can be found in (3).

For our purposes, developing the multithreaded server needs to define the handler object for the requests which the server receives.

Thus, the `GetHandler` class defines the method `do_GET()` which actually splits the requested URL into its components and calls the proper callback function associated with that URL or sends and HTTP error message if the URL is not recognised.

Some support methods are defined to allow the code to server files as well.

```
import base, threading
import mimetypes, urlparse, urllib, urllib2

from BaseHTTPServer import HTTPServer
from BaseHTTPServer import BaseHTTPRequestHandler

class GetHandler(BaseHTTPRequestHandler):
    def __init__(self, server, *args):
        self.httpserver = server
        BaseHTTPRequestHandler.__init__(self, *args)

    def Redirect(self,url=None): # Default redirects to home
        if url == None:
            url = '/'
        self.send_response(303)
        self.send_header('Location', url)
        self.end_headers()

    def SendData(self, data, mime='text/plain'):
        self.send_response(200)
        self.send_header('Content-type',mime)
        self.end_headers()
        self.wfile.write(data)

    def SendFile(self, filename):
        try:
            filename = self.httpserver.htdocs + filename
            with open(filename,'rb') as f:
                self.SendData(f.read(), mimetypes.guess_type(filename)[0])
        except:
            self.send_error(404)
        return None

    def log_message(self, format, *args):
        return

    def send_error(self, code, message=None):
        self.SendData("- fail " + str(code))

    def do_GET(self):   # EXTRACT THE CALLBACK FUNCTION
        parsed_path      = urlparse.urlparse(self.path)
        self.fullpath    = parsed_path.path
        self.pathlist    = parsed_path.path.strip("/").split('/')
        self.querydict   = urlparse.parse_qs(parsed_path.query)
        self.querylist   = urlparse.parse_qsl(parsed_path.query)
        self.query       = parsed_path.query

        if '' == self.pathlist[0]:  # HANDLER FOR THE DEFAULT URL
            self.httpserver.handler_default(self)
        elif 'htdocs' == self.pathlist[0]:
            self.httpserver.handler_htdocs(self)
        elif 'node' == self.pathlist[0]:
            self.httpserver.handler_node(self)
        else:
            self.send_error(400) # Bad request (incomplete path)
        return
```

The Client Side of the HTTP Module is much simpler and is shown below.

```
class Client(base.ClientModule):
    def __init__(self, baseurl):
        super(Client, self).__init__(None)
        self.baseurl = baseurl

    def TxRxTransaction(self, command):
        try:
            url = self.baseurl + "?" + urllib.quote(command)
            response = urllib2.urlopen(url)
            self.reply = response.read()
        except:
            self.reply = None
```

Appendix

Internetworking

In a distributed setup some actions must be taken to allow access to the resources. As Figure 13 shows, when a user application in site A tries to access a Board resource in site B, Mid-Tier(A) needs to establish a TCP connection to the listening TCP port of Mid-Tier(B). In usual conditions this does not happen because of the security network policies implemented on the Gateway GW(B) in site B. Additionally, the Mid-Tier software at both ends could be behind a Firewall (FW(A) and FW(B)) in order to filter-out the undesired or harmful network traffic.

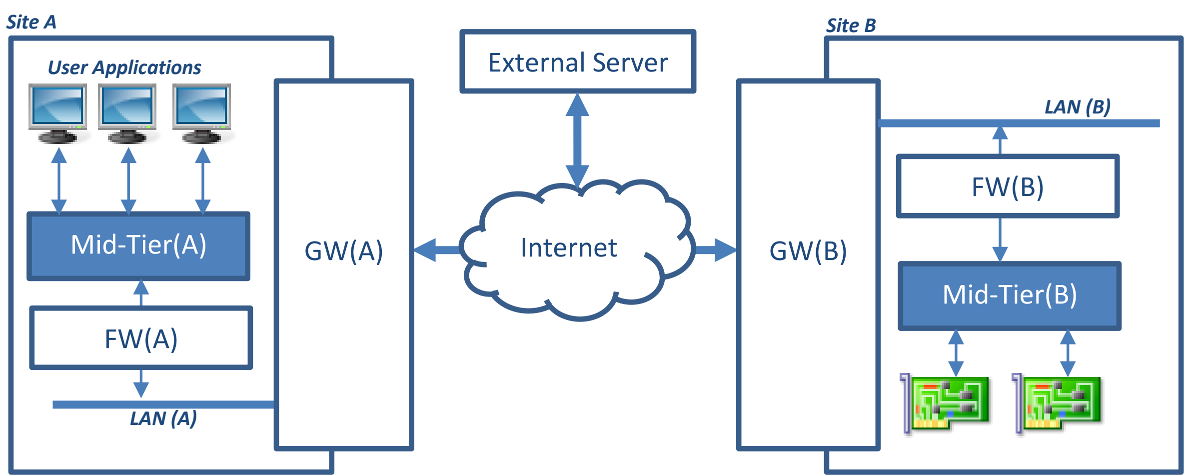


FIGURE 13 – INTERNETWORKING

In this common situations some scenarios are possible and some actions can be implemented

- Giving public access of Mid-Tier(B) by configuring GW(B) and FW(B).
  This operations must be performed by the network B administrator and is not recommended because it could expose to security risks the host Mid-Tier(B) is running on.
- Using a VPN software on the computer running Mid-Tier(B) to virtually connect it to private network behind FW(A). The VPN concentrator in Site A must be connected to the same private network of the machine running Mid-Tier(A).
- Using other tunnelling techniques, for example an additional server which accepts connections from the two Mid-Tiers. When the connection to the external server is established, one Mid-Tier acts as a Master (thus mimics a Server Module) while the other as a Slave (thus mimics a Client Module). This approach is widely used by chat application or remote desktop software (e.g. TeamViewer ™) which use HTTP to connect to the external server, thus allowing to pass through the local firewalls.
  A dedicated module for the Mid-Tier software needs to be developed.
- Using a dedicated private network, if available.

Possible Instruments Abstraction in Python

A different approach to the register-based access of the board resources is presented and an abstraction layer is introduced. Abstraction reduces the programming efforts to the end-user who needs to develop his/her application software. This part of the report describes an example of abstraction using the Python language.

At the software level, Figure 14 shows how the single instruments boards could be described as derived classes of a common generic board class. At a higher level, a set of boards can be organized into an object which describes the locally interconnected devices inside a rack. Finally, the top level of the system represents the interconnected racks through the Mid-Tier Software Node thus constituting a network. A separate object, implements the communication with the network.

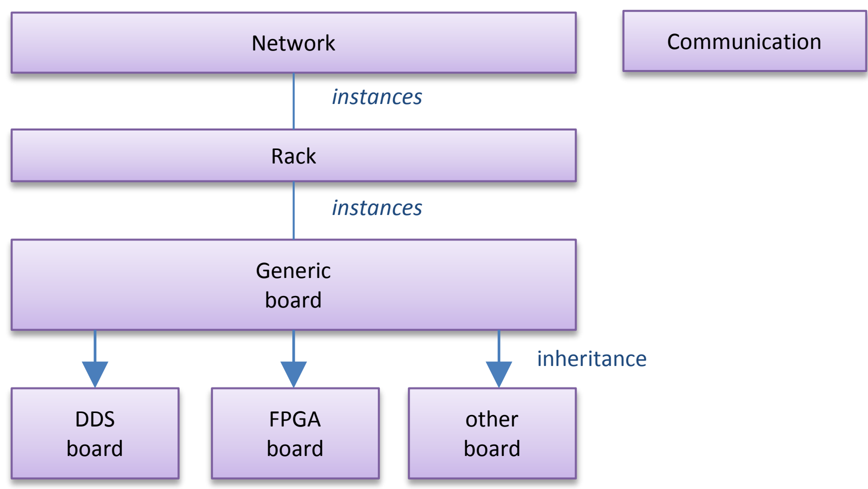


FIGURE 14 - INSTRUMENT NETWORK ABSTRACTION

The advantage of the described organization is that every object of the network has its own properties and methods and some operations in the application software could be described in a more readable form.

The generic base board can be described as base class , which provides all the common properties and methods of a non-specialized board. The *controller* object, passed in the constructor of the class implements the communication with the generic board.

```
class Board(object): # Generic instrument board.
        def __init__(self, controller, id):
                self.controller = controller
                self._id=id
        def GET(self, register): # Get register value
                return self.controller.GET(self._id,register)
        def SET(self, register, value): Set register value
                return self.controller.SET(self._id,register, value)
        def Reset(self): # Reset the DDS
                self.controller.RESET(self._id)
        @property
        def ID(self): # Get the board I2C address
                return self._id
        @property
        def Name(self): # Get / Set the board name
                return self.GET( STO_BOARD_NAME )
        @Name.setter
        def Name(self, Name):
                self.SET(STO_BOARD_NAME, Name)

# other code . . .
```

For example, the implementation of the derived class for a DDS board can start from the base `Board` class and then define the relevant properties like four *channel* objects. Each channel object represents one of the physical channels of the DDS chip which is present on the board.

```
class DDSBoard(Board): # An Arduino Board connected to a DDS
        def __init__(self, controller, id):
                super(DDSBoard, self).__init__(controller, id)
                self._channels = [Channel(self, i) for i in range(4) ]

        @property
        def Channels(self): # Get the DDS channels objects
                return self._channels
# other code . . .
```

The channel class provides all the properties and methods of a given channel. In the example below, only the implementation of the channel name and frequency are shown. Access to the DDS functions is achieved through the GET/SET methods implemented in the base `Board` class.

```
class Channel(object): # A DDS Channel - This class implements the properties of a DDS channel.
        def __init__(self, board, ch):
                """
                Constructor for the channel object.
                Parameters
                        board : board object
                        ch : DDS channel number (0-3)
                """
                self._ch=ch
                self._Board =board

        @property
        def Num(self):
                return self._ch
        @Num.setter
        def Num(self, n):
                self._ch = n

        @property
        def Name(self): # Get / Set the channel name
                if self._ch == 0:
                        return self._Board.GET( STO_CH0_NAME )
                elif self._ch == 1:
                        return self._Board.GET( STO_CH1_NAME )
                # other code . . .
        @Name.setter
        def Name(self, Name):
                if self._ch == 0:
                        self._Board.SET(STO_CH0_NAME, Name)
                elif self._ch == 1:
                        self._Board.SET(STO_CH1_NAME, Name)
                # other code . . .

        @property
        def Frequency(self): # Get / Set the DDS channel tuning word
                if self._ch == 0:
                        return self._Board.GET( REG_CH0_FREQ )
                elif self._ch == 1:
                        return self._Board.GET( REG_CH1_FREQ )
                # other code . . .
        @Frequency.setter
        def Frequency(self, Frequency):
                if self._ch == 0:
                        self._Board.SET(REG_CH0_FREQ, Frequency)
                elif self._ch == 1:
                        self._Board.SET(REG_CH1_FREQ, Frequency)
                # other code . . .
# other code . . .
```

At this point, once the `Channel`, `Board` and `DDSBoard` have been defined, a common operation like setting the frequency of channel 2 of a board to 1.5 MHz can be written as

```
board.Channels[2].frequency = 1.5
```

where `board` is the Python object associated to the given physical board.

The Rack object provides an array of locally connected boards, whatever the type they are. This is done by populating the array with the `Scan()` method of the class.

```
class Rack(object): # A set of Boards
    """
    This class implements a set of interconnected boards.
    One board acts as the bus controller.
    """
    def __init__(self, CommunicationObject):
        """
        Constructor for Rack object
        Parameters:
            CommunicationObject: Provides Board Communication
        """
        self.controller = CommunicationObject
        # other initialization code …

    def BoardClass(self, board): # Return the class of the board with given ID
        driver = self.controller.GET(board, REG_FW_DRIVER)
        if 'dds' == driver:
            import ddsboard
            return ddsboard.DDSBoard
        elif fpga' == driver:
            import fpgaboard
            return fpgabord.FPGABoard
        else:
            import board
            return board.Board

    def Scan(self): # Search all the boards. For each found board add the instance to the _Boards array
        self._Boards = {}
        for b in self.controller.LIST():
            driver = self.BoardClass(int(b))
            self._Boards[int(b)] = driver(self.controller, b)

# other code . . .
```

The `Scan()` method lists all the boards in the rack and for each one calls the method `BoardClass()`with its ID as a parameter. The `BoardClass()` method queries the specified board for its type by reading its `REG_FW_DRIVER` register and according to the returned value returns the correct board class to `Scan()`. In the end, the array `Boards` contains the objects describing the connected boards, each descendent of the base board class, each abstracting exactly that board.

In this way, the software abstracts the hardware resources which can be accessed through the `Boards` array property of the object `rack` and specific board methods and properties (although GET/SET are still available for special low-level access).

```
# controller is instantiated before
rack = Rack(controller)
rack.Scan()

for board in rack.Boards:
    BoardType = board.Firmware["driver"]

    print board.ID, " : ", BoardType ,board.Name

    if BoardType == "dds":
        for channel in board.Channels:
            print channel.Num, channel.Name, "  f=", channel.Frequency

    if BoardType == "fpga":
        for pin in board.Inputs:
            print "PIN", pin.num, ":", pin.value

# Other code ...
```

Software-defined instruments

An interesting by-product of the Mid-Tier Layer framework is the definition of *virtual instruments.* Such nodes behave like ordinary boards but the features are actually defined by the software.

The implementation of the virtual instruments lies of a the board simulator already present in the mid-tier layer framework and an extension of the simulator class which defines the desired behaviour. The aim of virtual instruments is a better integration of different instruments and data handlers into the end-user application.

As the scope is too wide to fit in this technical report it will not be discussed in detail here. A few examples of virtual instruments are given anyway for reference.

- **IEEE488 controller**
  The virtual board can control GPIB-connected instruments thus integrating third party instrumentation into the user application.
- **User interface**
  This node accesses parameters and/or data by accessing the network nodes. Some examples of implementations are data presentation in graphical format or in other formats useful for the end-user. Control panels with switches and knobs could be implemented as well, possibly using battery powered single board computers to run the *virtual instrument* software. The node could also be run into a mobile device to facilitate the access to the experimental setup in the field.
  Another application could be the real-time data presentation onto a dashboard mounted on the wall of the lab to show the operating conditions and health of a running experiment, key measurements and environmental parameters.
- **Software Interfacing**
  This node acts as an interface between the core of an existing user-software and the protocol of the boards network for general purposes.
- **Hardware simulator**
  This node mimics the behaviour of a hardware which is not available yet. This allows to develop and test in advance the overall network which controls an experiment.
- **Computing node**
  The node accesses other nodes of the network and performs intermediate calculations. The partial results are then made accessible thus reducing the network load.
- **Database node**
  A data storage used to save logs, configurations, measurements, results, etc. Data can be accessed in parallel by the boards network and other software. For example, this can be used for documentation, automatic report generation, log of operations for quality assurance, etc.
- **Software Repository**
  This node contains the software and firmware used by the experimental setup. The aim of this node is to provide a centralized storage to get the needed software to operate an experiment. For example, a user could connect to the Software Repository Node with an ordinary browser to get the required applications and libraries. Reliability of software revision tracking and quality assurance could benefit from the centralized software distribution point as well.

Bibliography

1. **Francese, Claudio.** *Flexible Arduino-Based board - Base Firmware R1.0.* INRIM. 2017. Technical Report. TR 24/2017.

2. —. *Flexible Arduino-Based board - Firmware Extension for DDS.* INRIM. 2017. Technical Report. TR 25/2017.

3. **Python Software Foundation.** Python Homepage. *Python Software Foundation Website.* [Online] 2017. http://www.python.org.

4. **The wxPython Team.** WxPython Homepage. [Online] 2017. http://www.wxpython.org.

5. **I2C Bus.** I2C Bus Organization. *I2C Bus Organization.* [Online] 2017. https://www.i2c-bus.org.

6. **Francese, Claudio.** *Templating and Automatic Code Generation.* INRiM. 2017. Tecnical Report. TR 27/2017.

7. **com0com.** Null Modem Emulator. [Online] http://com0com.sourceforge.net/.

8. **Arduino.** Arduino Nano. [Online] Arduino, 2017. https://store.arduino.cc/arduino-nano.

9. **Atmel Corporation / Microchip.** ATmega328P. *Microchip / Atmel Corporation.* [Online] 2017. http://www.microchip.com/wwwproducts/en/atmega328p.