Introduction

This article describes the mechanism of the shared memory wrapper, shared memory server, shared memory client and transportation protocols.

Accordingly, this article is divided to four separate parts, each of them describe on of the following features.

Shared Memory Server

**About**

The purpose of the shared memory server, is to provide easy and suitable interface, in order to control remote machine and handle with its shared memory.

For example, if we want to execute test on a remote machine that checks the shared memory, it will be easier to connect to the shared memory remote server over, using the remote machine itself.

**Protocol**

In order to communicate properly with the shared memory client, the shared memory client and server have a unique protocol.

The protocol is a simple stateless protocol, which each message is one of the following formats:

Request message:

1. Request code = integer
2. Arguments = tuple

The first property of the class is the request code, which represent which shared memory command we want to execute. The following commands are supported:

SMT\_INIT = 1

SMT\_VERSION = 2

SMT\_CREATE\_TOPIC = 3

SMT\_PUBLISH = 4

SMT\_GET\_BY\_COUNTER = 5

SMT\_GET\_LATEST = 6

SMT\_GET\_OLDEST = 7

SMT\_GET\_PUBLISH\_COUNT = 8

SMT\_CLEAR\_HISTORY = 9

EXIT = 999

The second property of the class is a tuple, which contains the arguments of the function the client want to execute.

For example, SMT\_PUBLISH function (request\_code = 4), will have one argument which is the structure we want to publish.

If the function does not receive any parameters, the property will contain empty tuple.

Response message:

1. Response\_code = integer
2. Response\_object = object

The first property of the class is the response code, which represent if the command the client tried to execute succeed or not.

It will be 0, if error was occurred, otherwise it will be 1.

The second property is the return object from the function. For example, the return object from the SMT\_GetByCounter function will be the object which was in the shared memory before.

If there is no return object from the function, this property will be none.

**Flow**

The flow of the shared memory server is pretty simple.

1. The init\_configuration function is called, in order to initialize dictionary of the configuration files.
2. Shared memory server object is created, and the function start\_server is being called
3. The server get the proper connection type from the connection factory, based on the configuration passed to him
4. The server is waiting for a client
5. The server is handle and executing function which received from the client in the Request class
6. The server send the client the output of the function with response code in the Response class

Shared Memory Client

**About**

The purpose of the shared memory client, is to provide easy and suitable interface, in order to communicate with remote machine.

**Protocol**

In order to communicate properly with the shared memory server, the shared memory client and server have a unique protocol.

The protocol is the same as described before.

**Flow**

The shared memory client have main goal to communicate and execute command on the remote machine.

But before that, he needs to execute the shared memory server and the remote machine.

1. First, the remote\_agent.py script need to be executed, with the proper configuration files as an argument   
   \*\*Attention\*\* - the remote\_agent.py script need to be run as administrator in order to use psexec to execute process in the remote machine.
2. After that, the start\_client.py need to be execute with the client configuration file as an argument
3. The client will initialize the configuration files into a dictionary
4. The client will get the proper shared memory object based on the configuration, if the method is set to remote, than it will get SharedMemoryClient object, and if the method is set to local, it will get the SharedMemoryWrapper object
5. The client will send the first package to the server at the beginning in order to initialize the structures defines in the configuration files
6. After that, the tester can execute any Shared Memory command he want to, on the local machine or on a remote one, both of them will have the same syntax

**Shared Memory Wrapper**

Shared memory wrapper is a python module (comes wth .pyd and .dll file), which its purpose is to give accessibility to the microblock functions such as publish and get by counter, throw python script.  
This module was built with pybind11 library, and have a specific interface.  
 Moreover, a parser comes with the shared memory wrapper, which is purpose is to parse topic header file, into couple of headrers + cpp files, and will build all of them into .pyd file. The .dll file is for microblocks function support.   
The module, is just the same as the microblocks function, and supported the following functions:

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Function Signature | Parameters | Return Value |
| SMT\_Version | const char\* SMT\_Version(); | None | Return the version as null terminated string. |
| SMT\_Init | bool SMT\_Init(); | None | true for initialization success, false otherwise |
| SMT\_Show | void SMT\_Show(const char\* topicName); | None | None |
| publish | bool publish(const char\* topicName, void\* ptr, uint32\_t size); | const char\* topicName, void\* ptr, uint32\_t counter, uint64\_t timeout\_us | true for initialization success, false otherwise |
| SMT\_CreateTopic | bool SMT\_CreateTopic(const char\* topicName, const uint32\_t maxDataSize, const uint32\_t historyDepth, const uint32\_t cellsCount); | const char\* topicName, const uint32\_t maxDataSize, const uint32\_t historyDepth, const uint32\_t cellsCount | true for initialization success, false otherwise |
| SMT\_getByCounter | bool SMT\_getByCounter(const char\* topicName, void\* ptr, uint32\_t counter, uint64\_t timeout\_us);  bool SMT\_getByCounter (const char\* topicName, void\* ptr, uint32\_t counter, uint64\_t timeout\_us, SMT\_DataInfo data\_info); | const char\* topicName, void\* ptr, uint32\_t counter, uint64\_t timeout\_us, , SMT\_DataInfo data\_info (optional) | true for publication existed and copied, false if timed out or topic not exist in history.  In datainfo object: Return info about the topic which got, in the data info struct |
| SMT\_getLatest | bool SMT\_getLatest(const char\* topicName, void\* ptr);  bool SMT\_getLatest (const char\* topicName, void\* ptr, SMT\_DataInfo data\_info); | const char\* topicName, void\* ptr, , SMT\_DataInfo data\_info (optional) | true for publication existed and copied, false if timed out or topic not exist in history.  In datainfo object: Return info about the topic which got, in the data info struct |
| SMT\_getOldest | bool SMT\_getOldest(const char\* topicName, void\* ptr);  bool SMT\_getOldest(const char\* topicName, void\* ptr, SMT\_DataInfo data\_info); | const char\* topicName, void\* ptr, SMT\_DataInfo data\_info (optional) | true for publication existed and copied, false if timed out or topic not exist in history.  In datainfo object: Return info about the topic which got, in the data info struct |
| SMT\_GetPublishCount | int SMT\_GetPublishCount(const char\* topicName); | const char\* topicName | amount of times topic has been published successfully. 0 if topic does not exist |
| SMT\_ClearHistory | bool SMT\_ClearHistory(const char\* topicName); | const char\* topicName | true if topic exists, false otherwise |

**Usage**

In order to use the module, all you need to do is to put the .pyd and .dll files in your working directory, and write import SharedMemoryWrapper at the top of your python file. After that simple steps, you will be able to use all the function I mention below.

Transportation Protocols

**About**

The purpose of the transportation protocols module, is to provide easy and suitable interface, in order to use transportation protocols in order to receive and send message such as TCP and UDP.

**Interface**

The transportation protocols module, contain base interface (abstract class) called CommDeviceInterface.

You can think about all the transportation protocols, as a wrapper class for specific connection (udp or tcp).

Each transportation protocol, and also future ones, need to implement the functions defined in the base class.

The function are:

1. Send(self, message) function which receive message and send it over the connection of the class.
2. Receive(self, timeout\_seconds, buffer\_size\_bytes) function which receive message from the connection of the class. The timeout and buffer\_size parameters are optional, they can be provide also at the configuration file.

The init function of the base class, just initialize the buffer size and the timeout for the receive function.

**TCP Device**This class is base class of all TCP connection classes. The TCP classes just inherited from this class and override the function in it. The functions in the class are:

1. Init(self, tcp\_connection, timeout, buffer\_size\_bytes) function which initialize the TCP connection of the class and also initialize the timeout and the buffer size of the parent class.
2. Getconnection(self) function which return the current TCP connection.
3. Set\_connection(self, tcp\_connection) function which set the TCP connection of the class.
4. Send(self, message, tcp\_connection) function which sends message to the tcp connection. If the tcp connection parameter is None, it will send to the tcp connection of the class.
5. Receive(self, buffer\_size\_bytes, timeout\_seconds, tcp\_connection) function will receive from the following connection tcp packat. If the tcp connection is None, it will receive from the tcp connection of the class.

**TCP Client**This class inherited from TCPDevice class, and have all its function plus the connect function:  
connect(self) function connect to the tcp connection of the class.

**TCP Server**This class inherited from TCPDevice class, and have all its function plus the accept function:  
accept(self) function accept a client, set the connection of the class to be the new connection created, and return the new connection.

**UDP Device**This class is base class of all UDP connection classes. The UDP classes just inherited from this class and override the function in it. The functions in the class are:

1. Init(self, udp\_connection, timeout, buffer\_size\_bytes) function which initialize the UDP connection of the class and also initialize the timeout and the buffer size of the parent class.
2. Send(self, message) function which sends message to a udp connection.
3. Receive(self, buffer\_size\_bytes, timeout\_seconds) function will receive packat from a udp connection

**UDP Responder**This class inherited from UDPDevice class, all its purpose is to receive messages, so it overrides all UDPDevice function's except of send:

1. Init(self, udp\_connection, timeout, buffer\_size\_bytes) function call the init of the UDPDevice with the buffer size and the timeout, and also initialize the udp connection of the class
2. Receive(self, timeout\_seconds, buffer\_size\_bytes) function receive message from the current UDP connection with the timeout and the buffer size.

**UDP Initiator**This class inherited from UDPDevice class, all its purpose is to send messages from specific port address, so it overrides all UDPDevice function's except of receive:

1. Init(self, udp\_connection, timeout, buffer\_size\_bytes) function call the init of the UDPDevice with the buffer size and the timeout, and also initialize the udp connection of the class
2. send(self, message) function send message to the destination ip and port of the class

**UDP Strict**This class inherited from UDPDevice class, and its purpose is to make specific udp connection, from one ip address and port to another ip address and port. So its override all the functions of the UDPDevice base class:

1. Init(self, udp\_connection, timeout, buffer\_size\_bytes) function call the init of the UDPDevice with the buffer size and the timeout, and also initialize the udp connection of the class
2. Receive (self, timeout\_seconds, buffer\_size\_bytes) function receive message from the current UDP connection with the timeout and the buffer size.
3. send(self, message) function send message to the destination ip and port of the class

**Connection Factory**this class, is a factory of all the connection mentioned before. Its main function, called get\_connection() is a static function, which produce the proper connection based on the configuration file. If you don't use any wrapper classes which use configuration file, you can initialize this singleton class by calling to the following functions:

Parseconfig.parseconfig(config\_files) function return dictionary of the configuration found in the configuration files. It's get list of configuration files as an argument.

In order to create each type of connection, you required to put different configuration in the config file:

TCP server: responder\_port = the port of the server  
  
TCP client: responder\_port = the port of the server, responder\_ip = the ip of the server  
  
UDP responder: responder\_port = the port of the server  
  
UDP client: responder\_port = the port of the server, responder\_ip = the ip of the server  
  
UDP strict: local\_port = the port of the current side, responder\_port = the port of the other side, responder\_ip = the ip of the other side. Local\_ip = the ip of the local side, this config is usefull for the other side in order to know which ip to bind.

Note that this configuration will changed from the two side. For example:

One side UDP strict configuration: local\_port = 1111, responder\_port = 2222, responder\_ip = "2.2.2.2"

Other side UDP strict configuration: local\_port = 2222, responder\_port = 1111, responder\_ip = "1.1.1.1"